6.858 Fall 2012 Lab 3: Server-side sandboxing

Handed out: Wednesday, October 3, 2012
Parts 1 and 2 due: Friday, October 12, 2012 (5:00pm)
All parts due: Friday, October 19, 2012 (5:00pm)

Introduction

In this lab, we will extend the zoobar web application to allow users to use Python code as their profiles. Whenever someone requests a user's profile, the server will execute that user's Python code to generate the resulting profile output. This will allow users to implement a variety of features in their profiles, such as:

- A profile that greets visitors by their user name.
- A profile that keeps track of the last several visitors to that profile.
- A profile that gives a zoobar to every visitor (limit 1 per minute).

Supporting this safely requires sandboxing the profile code on the server, so that it cannot perform arbitrary operations or access arbitrary files. On the other hand, this code may need to keep track of persistent data in some files, or to access existing zoobar databases, to function properly.

To fetch the new source code for this lab, use git to commit your lab 2 solutions, fetch the latest version of the course repository, and then create a local branch called `lab3` based on our lab3 branch, `origin/lab3`:

```
httpd@vm-6858:~$ cd lab
httpd@vm-6858:~/lab$ git commit -am 'my solution to lab2'
[lab2 f524ff8] my solution to lab2
 1 files changed, 1 insertions(+), 0 deletions(-)
httpd@vm-6858:~/lab$ git pull
Already up-to-date.
httpd@vm-6858:~/lab$ git checkout -b lab3 origin/lab3
Branch lab3 set up to track remote branch lab3 from origin.
Switched to a new branch 'lab3'
httpd@vm-6858:~/lab$
```

The new source code includes the following components, which you should familiarize yourself with:

- `profiles/` contains five Python-based profiles, which you will use as examples throughout this lab:
  - `profiles/hello-user.py` is a simple profile that prints back the name of the visitor when the profile code is executed, along with the current time.
  - `profiles/visit-tracker.py` keeps track of the last time that each visitor looked at the profile, and prints out the last visit time (if any).
  - `profiles/last-visits.py` records the last three visitors to the profile, and prints them out.
  - `profiles/xfer-tracker.py` prints out the last zoobar transfer between the profile owner and the visitor.
  - `profiles/granter.py` gives the visitor one zoobar, as long as the profile owner has any zoobars left, the visitor has less than 20 zoobars, and it has been at least a minute since the last time the visitor got a free zoobar.
- `zoobar/proflib.py` is a Python module imported by the Python-based profiles to provide an API for accessing zoobar state. For example `proflib.py` provides functions to get parameters passed to the Python profile from the zoobar web application, to look up a user's zoobar balance and profile, look up a list of transfers for a user, and to transfer zoobars.
- `zoobar/profile.py` contains a function to run a user's profile as sandboxed Python code. You will modify this file to switch it to the PyPy sandbox.
- `zoobar/nullsandbox.py` is a Python module that provides a `run` function to execute Python code and return its output. As suggested by its name, it does not provide any isolation for the code; it is the starting point for an (insecure) Python profile system.
- `zoobar/pypysandbox.py` provides some initial code for a module that uses the PyPy interpreter to implement a secure sandbox for Python code, which you will fully implement in this lab. We will discuss the PyPy sandbox more later.

To get started, verify that the lab 3 code you have checked out works, by setting up a user with each of the five Python profiles in the `profiles/` directory, and checking that the profile code works properly. To run the server, follow the same steps as before to set up the `/jail` directory and then run `zookld`:

```
httpd@vm-6858:~/lab$ make
cc -m32 -g -std=c99 -fno-stack-protector -Wall -Werror -D_GNU_SOURCE   -c -o zookld.o zookld.c
cc -m32 -g -std=c99 -fno-stack-protector -Wall -Werror -D_GNU_SOURCE   -c -o http.o http.c
cc -m32  zookld.o http.o  -lcrypto -o zookld
cc -m32 -g -std=c99 -fno-stack-protector -Wall -Werror -D_GNU_SOURCE   -c -o zookfs.o zookfs.c
cc -m32  zookfs.o http.o  -lcrypto -o zookfs
cc -m32 -g -std=c99 -fno-stack-protector -Wall -Werror -D_GNU_SOURCE   -c -o zookd.o zookd.c
cc -m32  zookd.o http.o  -lcrypto -o zookd
cc -m32 -g -std=c99 -fno-stack-protector -Wall -Werror -D_GNU_SOURCE   -c -o zooksvc.o zooksvc.c
cc -m32  zooksvc.o  -lcrypto -o zooksvc
httpd@vm-6858:~/lab$ sudo make setup
[sudo] password for httpd: 6858
./chroot-setup.sh
+ grep -qv uid=0
+ id
...
httpd@vm-6858:~/lab$ sudo ./zookld
zookld: Listening on port 8080
zookld: Launching zookd
...
```

You can run `make check` to run some basic tests and verify that the profile code is working properly (although, keep in mind that these tests are not exhaustive). At this point, the sandbox check will not pass. This is expected as the PyPy sandbox is not enabled yet.

If something doesn't seem to be working, try to figure out what went wrong, or contact the course staff, before proceeding further.

Part 1: Python profiles with privilege separation

The first part of this lab will require you to combine your privilege-separated design from lab 2 with the Python profiles from this lab. There are two main ways in which these features interact. First, the databases used to look up information are different (e.g., the zoobar balances are stored in a separate database in a privilege-separated design). Second, transferring zoobars between users in a privilege-separated design requires an authentication token for the sender.

To get started, you will next need to merge your solutions to achieve privilege separation for lab 2 into the `lab3` branch, by running:

```
httpd@vm-6858:~/lab$ git merge lab2
Merge made by recursive
...
httpd@vm-6858:~/lab$
```

At this point, if git reports any conflicts, you should resolve them first, and commit the resolved merge, before proceeding.

Exercise 1. Make Python profiles work in your privilege-separated design. Verify that the resulting system can correctly execute all five of the example profiles. In order to support the `granter.py` profile, which performs zoobar transfers, you may need to give the profile code an authentication token for the profile owner. Be sure that you do not create a way for an arbitrary user to get another user's authentication token. One way around this would be to extend the authentication service you implemented in lab 2, to perform an operation that runs a given user's profile with that user's current authentication token.

Run `sudo make check` to verify that your modified configuration passes our basic tests, except for the sandbox check.

hmm

Part 2: Initial sandboxing with PyPy

At this point, your web server can run user-supplied Python profiles. However, a malicious user may supply arbitrary Python code. Since the profile code is currently executed using `nullsandbox.py`, it can potentially perform arbitrary actions on the server, such as reading, writing, or deleting files accessible to the user ID under which the code is running.

To provide stronger isolation guarantees, we will use the PyPy sandbox. At a high level, PyPy is a Python interpreter, just like the standard CPython interpreter called `python` that you are used to using. One difference is that PyPy has a "sandbox" mode of execution. In this sandbox mode, whenever the PyPy interpreter wants to perform a system call (e.g., in order to open a file when it encounters a call to the Python `open()` function), it does not issue the system call directly, but instead sends the system call arguments over RPC to another process. It then waits for the RPC server to interpret the system call arguments, and send back the appropriate system call return values, before proceeding with its execution. Thus, the RPC server is in complete control of how the sandboxed code can interact with the outside world, and can implement different sandboxing policies.

For the purposes of this lab, the PyPy interpreter is fixed, but you will be responsible for implementing parts of the RPC server that interprets and executes "system calls" issued by the sandboxed interpreter, to support operations needed by our five Python profiles.

You can read more about the PyPy sandbox here:

- http://codespeak.net/pypy/dist/pypy/doc/sandbox.html

In the lab 3 source code, `zoobar/pypysandbox.py` implements an initial version of PyPy-based sandboxed execution using its `run` function. The `MySandboxedProc` class in `pypysandbox.py` implements the RPC server we described above. This class inherits from existing library code for implementing such an RPC server, which you can find in `/jail/zoobar/pypy-sandbox/pypy/translator/sandbox/pypy_interact.py` and `/jail/zoobar/pypy-sandbox/pypy/translator/sandbox/sandlib.py`. This library code invokes a method called `do_ll_os__ll_os_`*`syscall`*`()` to perform system call *`syscall`*; you can see a few examples in `pypysandbox.py` already. The library invokes the sandboxed PyPy interpreter binary, called `pypy-c`, from the `/zoobar/pypy-sandbox/pypy/translator/goal` directory (inside of a chroot to `/jail`).

To implement system calls related to the file system, the RPC server uses a Python-based representation of a file system, which you can see in `/jail/zoobar/pypy-sandbox/pypy/translator/sandbox/vfs.py`. You will be extending the file system parts of the RPC server in later exercises, but for now you may want to simply familiarize yourself with this code.

Exercise 2. Modify the zoobar web application to use `pypysandbox` instead of `nullsandbox` to execute Python profiles. For now, focus on supporting basic functionality working (namely, the `hello-user.py` profile). We will get to supporting other example profiles later.

To see system calls being issued by the sandboxed PyPy interpreter, set `self.debug` to `True` in the `MySandboxedProc` constructor `__init__()`.

You will need to provide a different version of `proflib.py` for Python profiles running inside of the sandbox, because the PyPy interpreter does not support some modules, such as the `sqlite` database. For now, you only need to implement the `get_param()` function in the sandboxed version of `proflib.py`. You will also need to expose your new version of `proflib.py` in the sandboxed file system, perhaps by modifying `pypysandbox.py`. Note that the sandboxed PyPy interpreter loads all files, including `proflib.py`, via calls to the RPC server in `pypysandbox.py`.

Run `sudo make check` to verify that your modified configuration passes our tests for `hello-user.py`. The sandbox check should also pass at this time. It is expected that the other profiles and `/tmp` check do not pass. If you run into problems from the `make check` tests, you can always check `/tmp/html.out` for the output html of the profiles. Similarly, you can also check the output of the server in `/tmp/zookld.out`. If there is an error in the server, they will usually display there.

Submit your answers to the first two parts of the lab assignment by running `make submit` to upload `lab3-handin.tar.gz` to the submission web site.

Part 3: Extending the PyPy sandbox

In this part of the lab, we will extend the PyPy sandbox implemented in `pypysandbox.py` to support operations needed for a profile to store persistent data, and to access zoobar application state.

So just code

Exercise 3. Implement a writable and persistent `/tmp` directory in the PyPy sandbox. This is needed by the `visit-tracker.py` and `last-visits.py` profiles to store their persistent information. Make sure that the `/tmp` directory seen by each user's profile is separate from other users, so that profiles of different users cannot tamper with each other's files. As in lab 2, remember to consider the possibility of usernames with special characters.

Is that the best place for it?

Ensure that the `visit-tracker` and `last-visits` profiles work correctly after you implement your changes. The `/tmp` check should also pass at this time.

Exercise 4. Since the standard Python SQLite module is implemented by calling into the native SQLite C/C++ library, it is not available in the PyPy sandbox (because the native library does not know how to forward its system calls via the RPC channel). In this exercise, your job is to support the `get_xfers()` function from `proflib.py` in the sandbox. A reasonable approach to do this is to extend the RPC server (`pypysandbox.py`) to perform the `get_xfers()` functionality on behalf of the sandboxed code. You will also need to modify `proflib.py` to invoke this new interface.

Hint: to create a new interface between code in the sandbox and the RPC server outside of the sandbox, such as for performing `get_xfers()` calls, consider overloading the file namespace by defining a special file name that corresponds to `get_xfers` calls. You can take a look at `VirtualizedSocketProc` in `.../pypy-sandbox/pypy/translator/sandbox/sandlib.py` to see an example of how the PyPy sandbox exposes access to TCP sockets in this manner.

Once you are finished with this exercise, the `xfer-tracker.py` should be functioning correctly in the sandbox.

Exercise 5. Implement the last remaining parts of `proflib.py` in the sandbox: `get_user()` and `xfer()`. Once you are done, `granter.py` should work from within the sandbox.

Challenge! (optional) For extra credit, allow sandboxed code to safely manipulate sub-directories under `/tmp` using `mkdir` and `rmdir`, to open files in those sub-directories, and to be able to `unlink` and `rename` files and sub-directories. Write an example Python profile that uses sub-directories and renames files.

How do I start? No instructions

Challenge! (optional) Allow sandboxed code to safely create and use symlinks inside of its `/tmp` directory.

You are done! Run `make submit` to upload `lab3-handin.tar.gz` to the submission web site.

6.858

Lab 3
in OH

10/10
5P

Update code

Download code

Look at profiles

Store visitor log in .tmp file

Sandbox is ~~not~~ a seperate process

Verify w/ o changes

'did those files change'

(X) Conflicts in merge run-shellcode.c

✓ Fixed

Exercise 1

Make profiles work in priv seperated design

(2)

Wait - test w/ browser 1st

(X) Error on 1st pg
my debug line

(✓) Fixed

(X) Can't log in
Oh logging in when not registered → ugly error

(✓) Fixed - its trying to pull toolbars

Able to type in profile text
? where do users view other users' profile?

Exercise 1

Make profiles work in pin seperated design
?so do they not work at all now?

Granted, my needs a token

(3)

So how do we get started to making these work

So set the users profile to something in the directory?

Yes ✓ - hellow user works!

And profile shows up on users tab

Last 3 users ✓ worked

both for me and others

Last visit ✓

Ok now get the transfer to work

---

† A; Note who needs token

Backend Fetch token in src call

(4)

So currently we need a token from sender

In our case sender = page owner

not page viewer

So this is the difficulty the TA spoke about

Lab: run users profile w/ users current token

TA: only user can change their profile

so only they can put in their token

"Setuid" process

Can't take recipent token

Don't return token!

Then attacker can do whatever

* Call to auth → run this users profile w/ his token

has run profile method

then that runs profile

└ See how users.py runs this ✓ sees

5

Ah so --run-profile

So instead have auth to run_profile
+ return result...

Try that
still need to run as that user...

- ⊗ Didn't import JSON
- ⊗ Run_profile not defined
- ⊗ User not defined
- ⊗ Not outputting anything
- ⊗ Auth has no field profile
  - So other db
    - Which anyone can read...
  - Switch to person.db

↓ stupid mistakes

(e)

(X) Working outside req context

Student trying to get flask outside

TA: Flask g is missing

run as other user
use other hooks
in proflib.py
run those fns as user
modify proflib.py

So add visitor to script in API call to auth
then rewrite obj -> not g.user

(X) Didn't change visitor

(7)

(✓) sweet profile works again

Now ~~check~~ fix transfer to use token
in get_xfer

I get it now

Any one can call profile
but people can do that anyway

(✗) Person object has no attr zookers

Oh must change the other ones
No the granter.py does that...

Adding calls to stderror

Oh that is working
but are typical zoobar change
in sep db

(6)

So make that then see what breaks

(X) Username error line 64

that was stupid error

(X) Must import JSON

(X) Permission denied

So auth user call transfer

set transfer grp to 6013 - which all share

TAs Didn't update

So see which grps can call socket for txt - in src-file

make sure auth is a member of that grp - in zookeeper

Work on Lab

10/11
6:50p

Git update from the TA
(✓) Done

Now remember where I was

(✗) still permission denied
I didn't update

(✓) Thanks for visiting I gave you 1 zoobar
But we don't know why profile is doing
Not erroring
But can't see demo

~~(✗)~~ Oh no it worked (✓)

Now make test
~~Oh it~~ Oh it logs out

(✓) Pass ex1
last try

Ex. 2 Initial sandboxing

currently in nullsandbox.py

So can run real code

- it's just executed
- which is really stupid!

They should have provided sample code that does that

Though it can't transfer - since service

So use PyPy sandbox

- sends system calls over RPC to another process
- which will inspect it and execute it
- and return outputs

We need to build part of this RPC server

(3)

My Sandboxed Rpc class in pypy sandbox implements

inherits existing lib code

do_ll_os__ll_os_syscall()

So some we in there

Lover~~rides~~ides existing

Lib implms sandboxed interpter pypy -c

L inside our sys call functions

Or is this what calls it

To implement calls to file system uses py-version of fs which can see in vfs.py

; How does this work?

; its some low level py code

(c)

Now the actual task:

Modify Foobar web app
Just support hello-user

Can turn on debug in My Sandboxed Proc Constructor
(✓) Done

Need a diff version of prof lib
: won't this break our lab 1 code?

Only need get_param() function
Expose new profile.py in sandboxed fs
So our sandboxed interpreter loads via call
grr - this will be annoying -
need to learn how all this sandbox code works

(5)

Ok so now what to actually do?

Wait nullsandbox slp process
- ? so could do other user id?

So in profile.py change import

So lots of mecurial error → no db in use
Seems to be false

(X) No module named proflib
So when we import proflib
Need to do that file check

So the file open
? in build virtual conf?

TA (Piazza @152): Yes

(6)

; where do we want to put it

; want to be able import w/ statement

; so in root

(X) still no

TA: put anywhere in chroot & so pypy sandbox can expose it

(If I was there - would be easy - last part of lab!)

But where to put it?

Don't get where files show up...

Jwang: Add to chroot-setup.pyy - pypy.py

(X) No module zoodb

Lso that works strip the rest out

(7)

except get_param()

stiped it out

(8) works

now check

(9) Sandbox check

Ex I died - but is that expected

Posted to Piazza

```
1   import os, sys, errno
2   from cStringIO import StringIO
3
4   pypy_sandbox_dir = '/zoobar/pypy-sandbox'
5   sys.path = [pypy_sandbox_dir] + sys.path
6
7   from pypy.translator.sandbox import pypy_interact, sandlib, vfs
8   from pypy.translator.sandbox.vfs import Dir, RealDir, RealFile
9   from pypy.rpython.module.ll_os_stat import s_StatResult
10  from pypy.tool.lib_pypy import LIB_ROOT
11
12  class WritableFile(vfs.RealFile):
13      def __init__(self, basenode):
14          self.path = basenode.path
15      def open(self):
16          try:
17              return open(self.path, 'wb')
18          except IOError, e:
19              raise OSError(e.errno, 'write open failed')
20
21  class MySandboxedProc(pypy_interact.PyPySandboxedProc):
22      def __init__(self, profile_owner, code, args):
23          super(MySandboxedProc, self).__init__(
24              pypy_sandbox_dir + '/pypy/translator/goal/pypy-c',
25              ['-S', '-c', code] + args
26          )
27          self.debug = False
28          self.virtual_cwd = '/'
29
30      ## Replacements for superclass functions
31      def get_node(self, vpath):
32          dirnode, name = self.translate_path(vpath)
33          if name:
34              node = dirnode.join(name)
35          else:
36              node = dirnode
37          if self.debug:
38              sandlib.log.vpath('%r => %r' % (vpath, node))
39          return node
40
41      def handle_message(self, fnname, *args):
42          if '__' in fnname:
43              raise ValueError("unsafe fnname")
44          try:
45              handler = getattr(self, 'do_' + fnname.replace('.', '__'))
46          except AttributeError:
47              raise RuntimeError("no handler for " + fnname)
48          resulttype = getattr(handler, 'resulttype', None)
49          return handler(*args), resulttype
50
51      def build_virtual_root(self):
52          # build a virtual file system:
```

So what do we have to change?

```
53          # * can access its own executable
54          # * can access the pure Python libraries
55          # * can access the temporary usession directory as /tmp
56          exclude = ['.pyc', '.pyo']
57          tmpdirnode = RealDir('/tmp/sandbox-root', exclude=exclude)
58          libroot = str(LIB_ROOT)
59
60          return Dir({
61              'bin':  Dir({'pypy-c':     RealFile(self.executable),
62                           'lib-python': RealDir(libroot + '/lib-python', exclude=
    exclude),
63                           'lib_pypy':   RealDir(libroot + '/lib_pypy',   exclude=
    exclude)
64                          }),
65              'proc': Dir({'cpuinfo': RealFile('/proc/cpuinfo'), }),
66              'tmp':  tmpdirnode,
67              })
68
69      ## Implement / override system calls
70      ##
71      ## Useful reference:
72      ##    pypy-sandbox/pypy/translator/sandbox/sandlib.py
73      ##    pypy-sandbox/pypy/translator/sandbox/vfs.py
74      ##
75      def do_ll_os__ll_os_geteuid(self):
76          return 0
77
78      def do_ll_os__ll_os_getuid(self):
79          return 0
80
81      def do_ll_os__ll_os_getegid(self):
82          return 0
83
84      def do_ll_os__ll_os_getgid(self):
85          return 0
86
87      def do_ll_os__ll_os_fstat(self, fd):
88          ## Limitation: fd's 0, 1, and 2 are not in open_fds table
89          f = self.get_file(fd)
90          try:
91              return os.fstat(f.fileno())
92          except:
93              raise OSError(errno.EINVAL)
94      do_ll_os__ll_os_fstat.resulttype = s_StatResult
95
96      def do_ll_os__ll_os_open(self, vpathname, flags, mode):
97          if flags & (os.O_CREAT):
98              dirnode, name = self.translate_path(vpathname)
99              ## LAB 3: handle file creation
100
101         node = self.get_node(vpathname)
102         if flags & (os.O_RDONLY|os.O_WRONLY|os.O_RDWR) != os.O_RDONLY:
```

So certain calls we are filtering

```
103                 ## LAB 3: handle writable files, by not raising OSError in some cases
104                 raise OSError(errno.EPERM, "write access denied")
105                 node = WritableFile(node)
106
107             f = node.open()
108             return self.allocate_fd(f)
109
110         def do_ll_os__ll_os_write(self, fd, data):
111             try:
112                 f = self.get_file(fd)
113             except:
114                 f = None
115
116             if f is not None:
117                 ## LAB 3: if this file should be writable, do the write,
118                 ##        and return the number of bytes written
119                 raise OSError(errno.EPERM, "write not implemented yet")
120
121             return super(MySandboxedProc, self).do_ll_os__ll_os_write(fd, data)
122
123     def run(profile_owner, code, args = [], timeout = None):
124         sandproc = MySandboxedProc(profile_owner, code, args)
125
126         if timeout is not None:
127             sandproc.settimeout(timeout, interrupt_main=True)
128         try:
129             code_output = StringIO()
130             sandproc.interact(stdout=code_output, stderr=code_output)
131             return code_output.getvalue()
132         finally:
133             sandproc.kill()
134
135
```

Will appear in the 2009 IEEE Symposium on Security and Privacy

Native Client: A Sandbox for Portable, Untrusted x86 Native Code

Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth,
Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar
Google Inc.

Abstract

This paper describes the design, implementation and evaluation of Native Client, a sandbox for untrusted x86 native code. Native Client aims to give browser-based applications the computational performance of native applications without compromising safety. Native Client uses software fault isolation and a secure runtime to direct system interaction and side effects through interfaces managed by Native Client. Native Client provides operating system portability for binary code while supporting performance-oriented features generally absent from web application programming environments, such as thread support, instruction set extensions such as SSE, and use of compiler intrinsics and hand-coded assembler. We combine these properties in an open architecture that encourages community review and 3rd-party tools.

1. Introduction

As an application platform, the modern web browser brings together a remarkable combination of resources, including seamless access to Internet resources, high-productivity programming languages such as JavaScript, and the richness of the Document Object Model (DOM) [64] for graphics presentation and user interaction. While these strengths put the browser in the forefront as a target for new application development, it remains handicapped in a critical dimension: computational performance. Thanks to Moore's Law and the zeal with which it is observed by the hardware community, many interesting applications get adequate performance in a browser despite this handicap. But there remains a set of computations that are generally infeasible for browser-based applications due to performance constraints, for example: simulation of Newtonian physics, computational fluid-dynamics, and high-resolution scene rendering. The current environment also tends to preclude use of the large bodies of high-quality code developed in languages other than JavaScript.

Modern web browsers provide extension mechanisms such as ActiveX [15] and NPAPI [48] to allow native code to be loaded and run as part of a web application. Such architectures allow plugins to circumvent the security mechanisms otherwise applied to web content, while giving them access to full native performance, perhaps as a secondary consideration. Given this organization, and the absence of effective technical measures to constrain these plugins, browser applications that wish to use native-code must rely on non-technical measures for security; for example, manual establishment of trust relationships through pop-up dialog boxes, or manual installation of a console application. Historically, these non-technical measures have been inadequate to prevent execution of malicious native code, leading to inconvenience and economic harm [10], [54]. As a consequence we believe there is a prejudice against native code extensions for browser-based applications among experts and distrust among the larger population of computer users.

While acknowledging the insecurity of the current systems for incorporating native-code into web applications, we also observe that there is no fundamental reason why native code should be unsafe. In Native Client, we separate the problem of safe native execution from that of extending trust, allowing each to be managed independently. Conceptually, Native Client is organized in two parts: a constrained execution environment for native code to prevent unintended side effects, and a runtime for hosting these native code extensions through which allowable side effects may occur safely.

The main contributions of this work are:

- an infrastructure for OS and browser-portable sandboxed x86 binary modules,
- support for advanced performance capabilities such as threads, SSE instructions [32], compiler intrinsics and hand-coded assembler,
- an open system designed for easy retargeting of new compilers and languages, and
- refinements to CISC software fault isolation, using x86 segments for improved simplicity and reduced overhead.

We combine these features in an infrastructure that supports safe side effects and local communication. Overall, Native Client provides sandboxed execution of native code and portability across operating systems, delivering native code performance for the browser.

The remainder of the paper is organized as follows. Section 1.1 describes our threat model. Section 2 develops some essential concepts for the NaCl[1] system architecture and

1. We use "NaCl" as an adjective reference to the Native Client system.

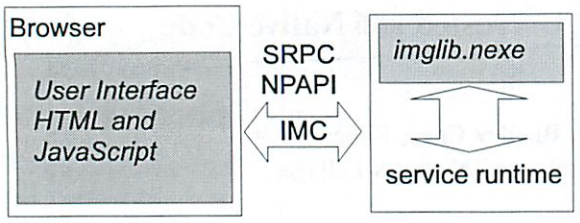


Figure 1: Hypothetical NaCl-based application for editing and sharing photos. Untrusted modules have a grey background.

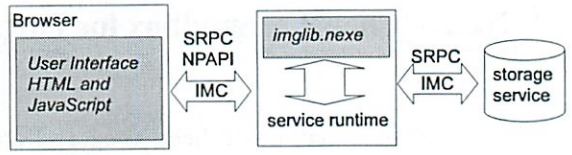


Figure 2: The hypothetical photo application of Figure 1 with a trusted storage service.

programming model. Section 3 gives additional implementation details, organized around major system components. Section 4 provides a quantitative evaluation of the system using more realistic applications and application components. In Section 5 we discuss some implications of this work. Section 6 discusses relevant prior and contemporary systems. Section 7 concludes.

1.1. Threat Model

Native Client should be able to handle untrusted modules from any web site with comparable safety to accepted systems such as JavaScript. When presented to the system, an untrusted module may contain *arbitrary* code and data. A consequence of this is that the NaCl runtime must be able to confirm that the module conforms to our validity rules (detailed below). Modules that don't conform to these rules are rejected by the system.

Once a conforming NaCl module is accepted for execution, the NaCl runtime must constrain its activity to prevent unintended side effects, such as might be achieved via unmoderated access to the native operating system's system call interface. The NaCl module may arbitrarily combine the entire variety of behaviors permitted by the NaCl execution environment in attempting to compromise the system. It may execute any reachable instruction block in the validated text segment. It may exercise the NaCl application binary interface to access runtime services in any way: passing invalid arguments, etc. It may also send arbitrary data via our intermodule communication interface, with the communicating peer responsible for validating input. The NaCl module may allocate memory and spawn threads up to resource limits. It may attempt to exploit race conditions in subverting the system.

We argue below that our architecture and code validity rules constrain NaCl modules within our sandbox.

2. System Architecture

A NaCl application is composed of a collection of trusted and untrusted components. Figure 1 shows the structure of a hypothetical NaCl-based application for managing and sharing photos. It consists of two components: A user interface, implemented in JavaScript and executing in the web browser, and an image processing library (imglib.nexe), implemented as a NaCl module. In this hypothetical scenario, the user interface and image processing library are part of the application and therefore untrusted. The browser component is constrained by the browser execution environment and the image library is constrained by the NaCl container. Both components are portable across operating systems and browsers, with native code portability enabled by Native Client. Prior to running the photo application, the user has installed Native Client as a browser plugin. Note that the NaCl browser plugin itself is OS and browser specific. Also note it is trusted, that is, it has full access to the OS system call interface and the user trusts it to not be abusive.

When the user navigates to the web site that hosts the photo application, the browser loads and executes the application JavaScript components. The JavaScript in turn invokes the NaCl browser plugin to load the image processing library into a NaCl container. Observe that the native code module is loaded silently—no pop-up window asks for permission. Native Client is responsible for constraining the behavior of the untrusted module.

Each component runs in its own private address space. Inter-component communication is based on Native Client's reliable datagram service, the IMC (Inter-Module Communications). For communications between the browser and a NaCl module, Native Client provides two options: a Simple RPC facility (SRPC), and the Netscape Plugin Application Programming Interface (NPAPI), both implemented on top of the IMC. The IMC also provides shared memory segments and shared synchronization objects, intended to avoid messaging overhead for high-volume or high-frequency communications.

The NaCl module also has access to a "service runtime" interface, providing for memory management operations, thread creation, and other system services. This interface is analogous to the system call interface of a conventional operating system.

In this paper we use "NaCl module" to refer to untrusted native code. Note however that applications can use multiple NaCl modules, and that both trusted and untrusted components may use the IMC. For example, the user of the photo

application might optionally be able to use a (hypothetical) trusted NaCl service for local storage of images, illustrated in Figure 2. Because it has access to local disk, the storage service must be installed as a native browser plugin; it can't be implemented as a NaCl module. Suppose the photo application has been designed to optionally use the stable storage service; the user interface would check for the stable storage plugin during initialization. If it detected the storage service plugin, the user interface would establish an IMC communications channel to it, and pass a descriptor for the channel to the image library, enabling the image library and the storage service to communicate directly via IMC-based services (SRPC, shared memory, etc.). In this case the NaCl module will typically be statically linked against a library that provides a procedural interface for accessing the storage service, hiding details of the IMC-level communications such as whether it uses SRPC or whether it uses shared memory. Note that the storage service must assume that the image library is untrusted. The service is responsible for insuring that it only services requests consistent with the implied contract with the user. For example, it might enforce a limit on total disk used by the photo application and might further restrict operations to only reference a particular directory.

Native Client is ideal for application components requiring pure computation. It is not appropriate for modules requiring process creation, direct file system access, or unrestricted access to the network. Trusted facilities such as storage should generally be implemented outside of Native Client, encouraging simplicity and robustness of the individual components and enforcing stricter isolation and scrutiny of all components. This design choice echoes microkernel operating system design [2], [12], [25].

With this example in mind we will now describe the design of key NaCl system components in more detail.

2.1. The Inner Sandbox

Native Client is built around an x86-specific intra-process "inner sandbox." We believe that the inner sandbox is robust; regardless, to provide defense in depth [13], [16] we have also developed a second "outer sandbox" that mediates system calls at the process boundary. The outer sandbox is substantially similar to prior structures (systrace [50] and Janus [24]) and we will not discuss it in detail in this paper.

The inner sandbox uses static analysis to detect security defects in untrusted x86 code. Previously, such analysis has been challenging for arbitrary x86 code due to such practices as self-modifying code and overlapping instructions. In Native Client we disallow such practices through a set of alignment and structural rules that, when observed, insure that the native code module can be disassembled reliably, such that all reachable instructions are identified during disassembly. With reliable disassembly as a tool, our validator can then insure that the executable includes only the subset of legal instructions, disallowing unsafe machine instructions.

The inner sandbox further uses x86 segmented memory to constrain both data and instruction memory references. Leveraging existing hardware to implement these range checks greatly simplifies the runtime checks required to constrain memory references, in turn reducing the performance impact of safety mechanisms.

This inner sandbox is used to create a security subdomain within a native operating system process. With this organization we can place a trusted service runtime subsystem within the same process as the untrusted application module, with a secure trampoline/springboard mechanism to allow safe transfer of control from trusted to untrusted code and vice-versa. Although in some cases a process boundary could effectively contain memory and system-call side effects, we believe the inner sandbox can provide better security. We generally assume that the operating system is not defect free, such that the process barrier might have defects, and further that the operating system might deliberately map resources such as shared libraries into the address space of all processes, as occurs in Microsoft Windows. In effect our inner sandbox not only isolates the system from the native module, but also helps to isolate the native module from the operating system.

2.2. Runtime Facilities

The sandboxes prevent unwanted side effects, but some side effects are often necessary to make a native module useful. For interprocess communications, Native Client provides a reliable datagram abstraction, the "Inter-Module Communications" service or IMC. The IMC allows trusted and untrusted modules to send/receive datagrams consisting of untyped byte arrays along with optional "NaCl Resource Descriptors" to facilitate sharing of files, shared memory objects, communication channels, etc., across process boundaries. The IMC can be used by trusted or untrusted modules, and is the basis for two higher-level abstractions. The first of these, the Simple Remote Procedure Call (SRPC) facility, provides convenient syntax for defining and using subroutines across NaCl module boundaries, including calls to NaCl code from JavaScript in the browser. The second, NPAPI, provides a familiar interface to interact with browser state, including opening URLs and accessing the DOM, that conforms to existing constraints for content safety. Either of these mechanisms can be used for general interaction with conventional browser content, including content modifications, handling mouse and keyboard activity, and fetching additional site content; substantially all the resources commonly available to JavaScript.

As indicated above, the service runtime is responsible for providing the container through which NaCl modules

interact with each other and the browser. The service runtime provides a set of system services commonly associated with an application programming environment. It provides sysbrk() and mmap() system calls, primitives to support malloc()/free() interface or other memory allocation abstractions. It provides a subset of the POSIX threads interface, with some NaCl extensions, for thread creation and destruction, condition variables, mutexes, semaphores, and thread-local storage. Our thread support is sufficiently complete to allow a port of Intel's Thread Building Blocks [51] to Native Client. The service runtime also provides the common POSIX file I/O interface, used for operations on communications channels as well as web-based read-only content. As the name space of the local file system is not accessible to these interfaces, local side effects are not possible.

To prevent unintended network access, network system calls such as connect() and accept() are simply omitted. NaCl modules can access the network via JavaScript in the browser. This access is subject to the same constraints that apply to other JavaScript access, with no net effect on network security.

The NaCl development environment is largely based on Linux open source systems and will be familiar to most Linux and Unix developers. We have found that porting existing Linux libraries is generally straightforward, with large libraries often requiring no source changes.

2.3. Attack Surface

Overall, we recognize the following as the system components that a would-be attacker might attempt to exploit:

- inner sandbox: binary validation
- outer sandbox: OS system-call interception
- service runtime binary module loader
- service runtime trampoline interfaces
- IMC communications interface
- NPAPI interface

In addition to the inner and outer sandbox, the system design also incorporates CPU and NaCl module black-lists. These mechanisms will allow us to incorporate layers of protection based on our confidence in the robustness of the various components and our understanding of how to achieve the best balance between performance, flexibility and security.

In the next section we hope to demonstrate that secure implementations of these facilities are possible and that the specific choices made in our own implementation work are sound.

3. Native Client Implementation

3.1. Inner Sandbox

In this section we explain how NaCl implements software fault isolation. The design is limited to explicit control flow,

C1	Once loaded into the memory, the binary is not writable, enforced by OS-level protection mechanisms during execution.
C2	The binary is statically linked at a start address of zero, with the first byte of text at 64K.
C3	All indirect control transfers use a `nacljmp` pseudo-instruction (defined below).
C4	The binary is padded up to the nearest page with at least one `hlt` instruction (0xf4).
C5	The binary contains no instructions or pseudo-instructions overlapping a 32-byte boundary.
C6	All *valid* instruction addresses are reachable by a fall-through disassembly that starts at the load (base) address.
C7	All direct control transfers target valid instructions.

Table 1: Constraints for NaCl binaries.

expressed with calls and jumps in machine code. Other types of control flow (e.g. exceptions) are managed in the NaCl service runtime, external to the untrusted code, as described with the NaCl runtime implementation below.

Our inner sandbox uses a set of rules for reliable disassembly, a modified compilation tool chain that observes these rules, and a static analyzer that confirms that the rules have been followed. This design allows for a small trusted code base (TCB) [61], with the compilation tools outside the TCB, and a validator that is small enough to permit thorough review and testing. Our validator implementation requires less than 600 C statements (semicolons), including an x86 decoder and `cpuid` decoding. This compiles into about 6000 bytes of executable code (Linux optimized build) of which about 900 bytes are the `cpuid` implementation, 1700 bytes the decoder, and 3400 bytes the validator logic.

To eliminate side effects the validator must address four sub-problems:

- Data integrity: no loads or stores outside of data sandbox
- Reliable disassembly
- No unsafe instructions
- Control flow integrity

To solve these problems, NaCl builds on previous work on CISC fault isolation. Our system combines 80386 segmented memory [14] with previous techniques for CISC software fault isolation [40]. We use 80386 segments to constrain data references to a contiguous subrange of the virtual 32-bit address space. This allows us to effectively implement a data sandbox without requiring sandboxing of load and store instructions. VX32 [20], [21] implements its data sandbox in a similar fashion. Note that NaCl modules are 32-bit x86 executables. The more recent 64-bit executable model is not supported.

Table 1 lists the constraints Native Client requires of untrusted binaries. Together, constraints C1 and C6 make disassembly reliable. With reliable disassembly as a tool, detection of unsafe instructions is straightforward. A partial list of opcodes disallowed by Native Client includes:

- `syscall` and `int`. Untrusted code cannot invoke the

operating system directly.

- all instructions that modify x86 segment state, including `lds`, far calls, etc.
- `ret`. Returns are implemented with a sandboxing sequence that ends with an indirect jump.

Apart from facilitating control sandboxing, excluding `ret` also prevents a vulnerability due to a race condition if the return address were checked on the stack. A similar argument requires that we disallow memory addressing modes on indirect `jmp` and `call` instructions. Native Client does allow the `hlt` instruction. It should never be executed by a correct instruction stream and will cause the module to be terminated immediately. As a matter of hygiene, we disallow all other privileged/ring-0 instructions, as they are never required in a correct user-mode instruction stream. We also constrain x86 prefix usage to only allow known useful instructions. Empirically we have found that this eliminates certain denial-of-service vulnerabilities related to CPU errata.

The fourth problem is control flow integrity, insuring that all control transfers in the program text target an instruction identified during disassembly. For each direct branch, we statically compute the target and confirm it is a valid instruction as per constraint C6. Our technique for indirect branches combines 80386 segmented memory with a simplified sandboxing sequence. As per constraint C2 and C4, we use the `CS` segment to constrain executable text to a zero-based address range, sized to a multiple of 4K bytes. With the text range constrained by segmented memory, a simple constant mask is adequate to ensure that the target of an indirect branch is aligned mod 32, as per constraints C3 and C5:

```
and         %eax, 0xffffffe0
jmp         *%eax
```

We will refer to this special two instruction sequence as a `nacljmp`. Encoded as a three-byte `and` and a two-byte `jmp` it compares favorably to previous implementations of CISC sandboxing [40], [41], [56]. Without segmented memory or zero-based text, sandboxed control flow typically requires two six-byte instructions (an `and` and an `or`) for a total of fourteen bytes.

Considering the pseudo-code in Figure 3, we next assert and then prove the correctness of our design for control-flow integrity. Assuming the text in question was validated without errors, let S be the set of instructions addresses from the list StartAddr.

Theorem: S contains all addresses that can be reached from an instruction with address in S.

Proof: By contradiction. Suppose an address IP not in S is reached during execution from a predecessor instruction A with address in S. Because execution is constrained by x86 segmentation, IP must trivially be in [0:TextLimit). So IP can only be reached in one of three ways.

```
// TextLimit = the upper text address limit
// Block(IP) = 32-byte block containing IP
// StartAddr = list of inst start addresses
// JumpTargets = set of valid jump targets

// Part 1: Build StartAddr and JumpTargets
IP = 0; icount = 0; JumpTargets = { }
while IP <= TextLimit:
  if inst_is_disallowed(IP):
    error "Disallowed instruction seen"
  StartAddr[icount++] = IP
  if inst_overlaps_block_size(IP):
    error "Block alignment failure"
  if inst_is_indirect_jump_or_call(IP):
    if !is_2_inst_nacl_jmp_idiom(IP) or
      icount < 2 or
      Block(StartAddr[icount-2]) != Block(IP):
      error "Bad indirect control transfer"
  else
    // Note that indirect jmps are inside
    // a pseudo-inst and bad jump targets
    JumpTargets = JumpTargets + { IP }
  // Proceed to the fall-through address
  IP += InstLength(IP)

// Part 2: Detect invalid direct transfers
for I = 0 to length(StartAddr)-1:
  IP = StartAddr[I]
  if inst_is_direct_jump_or_call(IP):
    T = direct_jump_target(IP)
    if not(T in [0:TextLimit))
       or not(T in JumpTargets):
      error "call/jmp to invalid address"
```

Figure 3: Pseudo-code for the NaCl validator.

case 1: IP is reached by falling through from A. This implies that IP is InstAddr(A) + InstLength(A). But this address would have been in S from part 1 of the construction. Contradiction.

case 2: IP is reached by a direct jump or call from an instruction A in S. Then IP must be in JumpTargets, a condition checked by part 2 of the construction. Observe that JumpTargets is a subset of S, from part 1 of the construction. Therefore IP must be in S. Contradiction.

case 3: IP is reached by an indirect transfer from an instruction at A in S. Since the instruction at A is an indirect call or jump, any execution of A always immediately follows the execution of an `and`. After the `and` the computed address is aligned 0 mod 32. Since no instruction can straddle a 0 mod 32 boundary, every 0 mod 32 address in [0, TextLimit) must be in S. Hence IP is in S. Contradiction.

Hence any instruction reached from an instruction in S is also in S. ■

Note that this analysis covers explicit, synchronous control flow only. Exceptions are discussed in Section 3.2.

If the validator were excessively slow it might discourage people from using the system. We find our validator can check code at approximately 30MB/second (35.7 MB in 1.2

seconds, measured on a MacBook Pro with MacOS 10.5, 2.4GHz Core 2 Duo CPU, warm file-system cache). At this speed, the compute time for validation will typically be very small compared to download time, and so is not a performance issue.

We believe this inner sandbox needs to be extremely robust. We have tested it for decoding defects using random instruction generation as well as exhaustive enumeration of valid x86 instructions. We also have used "fuzzing" tests to randomly modify test executables. Initially these tests exposed critical implementation defects, although as testing continues no defects have been found in the recent past. We have also tested on various x86 microprocessor implementations, concerned that processor errata might lead to exploitable defects [31], [38]. We did find evidence of CPU defects that lead to a system "hang" requiring a power-cycle to revive the machine. This occurred with an earlier version of the validator that allowed relatively unconstrained use of x86 prefix bytes, and since constraining it to only allow known useful prefixes, we have not been able to reproduce such problems.

3.2. Exceptions

Hardware exceptions (segmentation faults, floating point exceptions) and external interrupts are not allowed, due in part to distinct and incompatible exception models in Linux, MacOS and Windows. Both Linux and Windows rely on the x86 stack via `%esp` for delivery of these events. Regrettably, since NaCl modifies the `%ss` segment register, the stack appears to be invalid to the operating system, such that it cannot deliver the event and the corresponding process is immediately terminated. The use of x86 segmentation for data sandboxing effectively precludes recovery from these types of exceptions. As a consequence, NaCl untrusted modules apply a failsafe policy to exceptions. Each NaCl module runs in its own OS process, for the purpose of exception isolation. NaCl modules cannot use exception handling to recover from hardware exceptions and must be correct with respect to such error conditions or risk abrupt termination. In a way this is convenient, as there are very challenging security issues in delivering these events safely to untrusted code.

Although we cannot currently support hardware exceptions, Native Client does support C++ exceptions [57]. As these are synchronous and can be implemented entirely at user level there are no implementation issues. Windows Structured Exception Handling [44] requires non-portable operating support and is therefore not supported.

3.3. Service Runtime

The service runtime is a native executable invoked by an NPAPI plugin that also supports interaction between the service runtime and the browser. It supports a variety of web browsers on Windows, MacOS and Linux. It implements the dynamic enforcement that maintains the integrity of the inner sandbox and provides resource abstractions to isolate the NaCl application from host resources and operating system interface. It contains trusted code and data that, while sharing a process with the contained NaCl module, are accessible only through a controlled interface. The service runtime prevents untrusted code from inappropriate memory accesses through a combination of x86 memory segment and page protection.

Platform	"null" Service Runtime call time
Linux, Ubuntu 6.06 Intel™ Core™ 2 6600 2.4 GHz	156
Mac OSX 10.5 Intel™ Xeon™ E5462 2.8 GHz	148
Windows XP Intel™ Core™ 2 Q6600 2.4 GHz	123

Table 2: Service runtime context switch overhead. The runtimes are measured in nanoseconds. They are obtained by averaging the measurements of 10 runs of a NaCl module which measured the time required to perform 10,000,000 "null" service runtime calls.

When a NaCl module is loaded, it is placed in a segment-isolated 256MB region within the service runtime's address space. The first 64 KB of the NaCl module's address space (NaCl "user" address space) is reserved for initialization by the service runtime. The first 4 KB is read and write protected to detect NULL pointers. The remaining 60 KB contains trusted code that implements our "trampoline" call gate and "springboard" return gate. Untrusted NaCl module text is loaded immediately after this 64 KB region. The `%cs` segment is set to constrain control transfers from the zero base to the end of the NaCl module text. The other segment registers are set to constrain data accesses to the 256 MB NaCl module address space.

Because it originates from and is installed by the trusted service runtime, trampoline and springboard code is allowed to contain instructions that are forbidden elsewhere in untrusted executable text. This code, patched at runtime as part of the NaCl module loading process, uses segment register manipulation instructions and the `far call` instruction to enable control transfers between the untrusted user code and the trusted service runtime code. Since every *0 mod 32* address in the first 64 KB of the NaCl user space is a potential computed control flow target, these are our entry points to a table of system-call trampolines. One of these entry points is blocked with a `hlt` instruction, so that the remaining space may be used for code that can only be invoked from the service runtime. This provides space for the springboard return gate.

Invocation of a trampoline transfers control from untrusted code to trusted code. The trampoline sequence resets `%ds` and then uses a `far call` to reset the `%cs` segment register and transfer control to trusted service handlers, reestablishing the conventional flat addressing model expected by the code in the service runtime. Once outside the NaCl user address space, it resets other segment registers such as `%fs`, `%gs`, and `%ss` to re-establish the native-code threading environment, fully disabling the inner sandbox for this thread, and loads the stack register `%esp` with the location of a trusted stack for use by the service runtime. Note that the per-thread trusted stack resides outside the untrusted address space, to protect it from attack by other threads in the untrusted NaCl module.

Just as trampolines permit crossing from untrusted to trusted code, the springboard enables crossing in the other direction. The springboard is used by the trusted runtime

- to transfer control to an arbitrary untrusted address,
- to start a new POSIX-style thread, and
- to start the main thread.

Alignment ensures that the springboard cannot be invoked directly by untrusted code. The ability to jump to an arbitrary untrusted address is used in returning from a service call. The return from a trampoline call requires popping an unused trampoline return addresses from the top of the stack, restoring the segment registers, and finally aligning and jumping to the return address in the NaCl module.

Table 2 shows the overhead of a "null" system call. The Linux overhead of 156 ns is slightly higher than that of the Linux 2.6 getpid syscall time, on the same hardware, of 138 ns (implemented via the vsyscall table and using the `sysenter` instruction). We note that the user/kernel transfer has evolved continuously over the life of the x86 architecture. By comparison, the segment register operations and far calls used by the NaCl trampoline are somewhat less common, and may have received less consideration over the history of the x86 architecture.

3.4. Communications

The IMC is the basis of communications into and out of NaCl modules. The implementation is built around a *NaCl socket*, providing a bi-directional, reliable, in-order datagram service similar to Unix domain sockets [37]. An untrusted NaCl module receives its first NaCl socket when it is created, accessible from JavaScript via the Document-Object Model object used to create it. The JavaScript uses the socket to send messages to the NaCl module, and can also share it with other NaCl modules. The JavaScript can also choose to connect the module to other services available to it by opening and sharing NaCl sockets as NaCl descriptors. NaCl descriptors can also be used to create shared memory segments.

Number of Descriptor	Linux	OSX	Windows
1	3.3	31.5	38
2	5.3	38.6	51
3	6.6	47.9	64
4	8.2	50.9	77
5	9.7	54.1	90
6	11.1	60.0	104
7	12.6	63.7	117
8	14.2	66.2	130

Table 3: NaCl resource descriptor transfer cost. The times are in microseconds. In this test, messages carrying zero data bytes and a varying number of I/O descriptors are transferred from a client NaCl module to a server NaCl module. On OSX, a request/ack mechanism is needed as a bug workaround in the OSX implementation of sendmsg. On Windows, a DuplicateHandle() system call is required per I/O object transferred.

Using NaCl messages, Native Client's SRPC abstraction is implemented entirely in untrusted code. SRPC provides a convenient syntax for declaring procedural interfaces between JavaScript and NaCl modules, or between two NaCl modules, supporting a few basic types (int, float, char) as well as arrays in addition to NaCl descriptors. More complex types and pointers are not supported. External data representation strategies such as XDR [18] or Protocol Buffers [26] can easily be layered on top of NaCl messages or SRPC.

Our NPAPI implementation is also layered on top of the IMC and supports a subset of the common NPAPI interface. Specific requirements that shaped the current implementation are the ability read, modify and invoke properties and methods on the script objects in the browser, support for simple raster graphics, provide the createArray() method and the ability to open and use a URL like a file descriptor. The currently implemented NPAPI subset was chosen primarily for expedience, although we will likely constrain and extend it further as we improve our understanding of related security considerations and application requirements.

3.5. Developer Tools

3.5.1. Building NaCl Modules. We have modified the standard GNU tool chain, using version 4.2.2 of the gcc collection of compilers [22], [29] and version 2.18 of binutils [23] to generate NaCl-compliant binaries. We have built a reference binary from newlib[2] using the resulting tool chain, rehosted to use the NaCl trampolines to implement system services (e.g., read(), brk(), gettimeofday(), imc_sendmsg()). Native Client supports an insecure "debug" mode that allows additional file-system interaction not otherwise allowed for secure code.

We modified gcc for Native Client by changing the alignment of function entries (`-falign-functions`) to

2. See http://sourceware.org/newlib/

32 bytes and by changing the alignment of the targets branches (`-falign-jumps`) to 32 bytes. We also changed gcc to use `nacljmp` for indirect control transfers, including indirect calls and all returns. We made more significant changes to the assembler, to implement Native Client's block alignment requirements. To implement returns, the assembler ensures that call instructions always appear in the final bytes of a 32 byte block. We also modified the assembler to implement indirect control transfer sequences by expanding the `nacljmp` pseudo-instruction as a properly aligned consecutive block of bytes. To facilitate testing we added support to use a longer `nacljmp` sequence, align the text base, and use an `and` and `or` that uses relocations as masks. This permits testing applications by running them on the command line, and has been used to run the entire gcc C/C++ test suite. We also changed the linker to set the base address of the image as required by the NaCl loader (64K today).

Apart from their direct use the tool chain also serves to document by example how to modify an existing tools chain to generate NaCl modules. These changes were achieved with less than 1000 lines total to be patched in gcc and binutils, demonstrating the simplicity of porting a compiler to Native Client.

3.5.2. Profiling and Debugging. Native Client's open source release includes a simple profiling framework to capture a complete call trace with minimal performance overhead. This support is based on gcc's `-finstrument-functions` code generation option combined with the `rdtsc` timing instruction. This profiler is portable, implemented entirely as untrusted code. In our experience, optimized builds profiled in this framework have performance somewhere between `-O0` and `-O2` builds. Optionally, the application programmer can annotate the profiler output with methods similar to printf, with output appearing in the trace rather than stdout.

Native Client does not currently support interactive debugging of NaCl binary modules. Commonly we debug NaCl module source code by building with standard tools and a library that exports all the interfaces to the NaCl service runtime, allowing us to build debug and NaCl modules from identical source. Over time we hope to improve our support for interactive debugging of release NaCl binaries.

4. Experience

Unless otherwise noted, performance measurements in this section are made without the NaCl outer sandbox. Sandbox overhead depends on how much message-passing and service runtime activity the application requires. At this time we do not have realistic applications of Native Client to stress this aspect of the system.

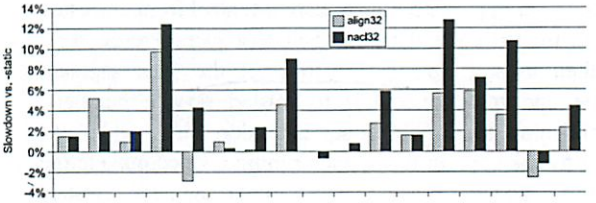


Figure 4: SPEC2000 performance. "Static" results are for statically linked binaries; "align32" results are for binaries aligned in 32-byte blocks, and "nacl32" results are for NaCl binaries.

	static	aligned	NaCl	increase
ammp	200	203	203	1.5%
art	46.3	48.7	47.2	1.9%
bzip2	103	104	104	1.9%
crafty	113	124	127	12%
eon	79.2	76.9	82.6	4.3%
equake	62.3	62.9	62.5	0.3%
gap	63.9	64.0	65.4	2.4%
gcc	52.3	54.7	57.0	9.0%
gzip	149	149	148	-0.7%
mcf	65.7	65.7	66.2	0.8%
mesa	87.4	89.8	92.5	5.8%
parser	126	128	128	1.6%
perlbmk	94.0	99.3	106	13%
twolf	154	163	165	7.1%
vortex	112	116	124	11%
vpr	90.7	88.4	89.6	-1.2%

Table 4: SPEC2000 performance. Execution time is in seconds. All binaries are statically linked.

4.1. SPEC2000

A primary goal of Native Client is to deliver substantially all of the performance of native code execution. NaCl module performance is impacted by alignment constraints, extra instructions for indirect control flow transfers, and the incremental cost of NaCl communication abstractions.

We first consider the overhead of making native code side effect free. To isolate the impact of the NaCl binary constraints (Table 1), we built the SPEC2000 CPU benchmarks using the NaCl compiler, and linked to run as a standard Linux binary. The worst case for NaCl overhead is CPU bound applications, as they have the highest density of alignment and sandboxing overhead. Figure 4 and Table 4 show the overhead of NaCl compilation for a set of benchmarks from SPEC2000. The worst case performance overhead is crafty at about 12%, with other benchmarks averaging about 5% overall. Hardware performance counter measurements indicate that the largest slowdowns are due to instruction cache misses. For crafty, the instruction fetch unit is stalled during 83% of cycles for the NaCl build, compared to 49% for the default build. Gcc and vortex are also significantly impacted by instruction cache misses.

As our current alignment implementation is conservative,

	static	aligned	NaCl	increase
ammp	657	759	766	16.7%
art	469	485	485	3.3%
bzip2	492	525	526	7.0%
crafty	756	885	885	17.5%
eon	1820	2016	2017	10.8%
equake	465	475	475	2.3%
gap	1298	1836	1882	45.1%
gcc	2316	3644	3646	57.5%
gzip	492	537	537	9.2%
mcf	439	452	451	2.8%
mesa	1337	1758	1769	32.3%
parser	641	804	802	25.2%
perlbmk	1167	1752	1753	50.2%
twolf	773	937	936	21.2%
vortex	1019	1364	1351	32.6%
vpr	668	780	780	16.8%

Table 5: Code size for SPEC2000, in kilobytes.

aligning some instructions that are not indirect control flow targets, we hope to make incremental code size improvement as we refine our implementation. "NaCl" measurements are for statically linked binaries, 32-byte block alignment, and using the `nacljmp` instruction for indirect control flow transfers. To isolate the impact of these three constraints, Figure 4 also shows performance for static linking only, and for static linking and alignment. These comparisons make it clear that alignment is the main factor in cases where overhead is significant. Impact from static linking and sandboxing instruction overhead is small by comparison.

The impact of alignment is not consistent across the benchmark suite. In some cases, alignment appears to improve performance, and in others it seems to make things worse. We hypothesize that alignment of branch targets to 32-byte boundaries sometimes interacts favorably with caches, instruction prefetch buffers, and other facets of processor microarchitecture. These effects are curious but not large enough to justify further investigation. In cases where alignment makes performance worse, one possible factor is code size, as mentioned above. Table 5 shows that increases in NaCl code size due to alignment can be significant, especially in benchmarks like gcc with a large number of static call sites. Similarly, benchmarks with a large amount of control flow branching (e.g., crafty, vortex) have a higher code size growth due to branch target alignment. The incremental code size increase of sandboxing with `nacljmp` is consistently small.

Overall, the performance impact of Native Client on these benchmarks is on average less than 5%. At this level, overhead compares favorably to untrusted native execution.

4.2. Compute/Graphics Performance Tests

We implemented three simple compute+animation benchmarks to test and evaluate our CPU performance for threaded code.[3] They are:

- Earth: a ray-tracing workload, projecting a flat image of the earth onto a spinning globe
- Voronoi: a brute force Voronoi tessellation[4]
- Life: cellular automata simulation of Conway's Game of Life

These workloads have helped us refine and evaluate our thread implementation, in addition to providing a benchmark against standard native compilation.

Sample	Native Client	Linux Executable
Voronoi	12.4	13.9
Earth	14.4	12.6
Life	21.9	19.4

Table 6: Compute/graphics performance tests. Times are user time in seconds.

Executable	1 thread	2 threads	4 threads
Native Client	42.16	22.04	12.4
Linux Binary	46.29	24.53	13.9

Table 7: Voronoi thread performance. Times are user time in seconds.

We used the Linux `time` command to launch and time standalone vs. NaCl release build executables. All measurements are for a Ubuntu Dapper Drake Linux system with a 2.4GHz Intel Q6600 quad core processor. VSYNC was disabled.[5] The normal executables were built using g++ version 4.0.3, the NaCl versions with nacl-g++ version 4.2.2. All three samples were built with `-O3 -mfpmath=sse -msse -fomit-frame-pointer`.

Voronoi used four worker threads and ran for 1000 frames. Earth ran with four worker threads for 1000 frames. Life ran as a single thread, for 5000 frames. Table 6 shows the average for three consecutive runs.

Voronoi ran faster as a NaCl application than as a normal executable. The other two tests, Earth and Life, ran faster as normal executables than their Native Client counterparts. Overall these preliminary measurements suggest that, for these simple test cases, the NaCl thread implementation behaves reasonably compared to Linux. Table 7 shows a comparison of threaded performance between Native Client and a normal Linux executable, using the Voronoi demo. Comparing Native Client to Linux, performance scales comparably with increased thread count.

4.3. H.264 Decoder

We ported an internal implementation of H.264 video decoding to evaluate the difficulty of the porting effort.

3. These benchmarks will be included in our open source distribution.
4. See http://en.wikipedia.org/wiki/Voronoi
5. It is important to disable VSYNC when benchmarking rendering applications. If VSYNC is enabled, the application's rendering thread may be put to sleep until the next vertical sync occurs on the display.

The original application converted H.264 video into a raw file format, implemented in about 11K lines of C for the standard GCC environment on Linux. We modified it to play video. The port required about twenty lines of additional C code, more than half of which was error checking code. Apart from rewriting the Makefile, no other modifications were required. This experience is consistent with our general experience with Native Client; legacy Linux libraries that don't inherently require network and disk generally port with minimal effort. Performance of the original and NaCl versions were comparable and limited by video frame-rate.

4.4. Bullet

Bullet [8] is an open source physics simulation system. It has accuracy and modeling features that make it appropriate for real-time applications like computer games. As a complex, performance sensitive legacy code base it is representative of a type of system that we would like to support with Native Client.

The effort required to build Bullet for Native Client was non-trivial but generally straightforward. We used Bullet v2.66 for our experiments which is configurable via autotools [5], allowing us specify use of the NaCl compiler. We also had to build the Jam build system [35], as it is required by the Bullet build. A few #defines also had to be adjusted to eliminate unsupported profiling system calls and other OS specific code. Overall it took a couple of hours of effort to get the library to build for Native Client.

Our performance test used the HelloWorld demo program from the Bullet source distribution, a simulation of a large number of spheres falling and colliding on a flat surface. We compared two builds using GCC v4.2.2 capable of generating NaCl compliant binaries. Measuring 100,000 iterations, we observed 36.5 seconds for the baseline build (-static) vs. 32-byte aligned blocks (as required by Native Client) at 36.1 seconds, or about a 1% speedup for alignment. Incorporating the additional opcode constraints required by Native Client results in runtime of 37.3 seconds, or about a 2% slowdown overall. These numbers were obtained using a two processor dual-core Opteron 8214 with 8GB of memory.

4.5. Quake

We profiled sdlquake-1.0.9 (from www.libsdl.org) using the built-in "timedemo demo1" command. Quake was run at 640x480 resolution on a Ubuntu Dapper Drake Linux box with a 2.4GHz Intel Q6600 quad core CPU. The video system's vertical sync (VSYNC) was disabled. The Linux executable was built using gcc version 4.0.3, and the Native Client version with nacl-gcc version 4.2.2, both with `-O2` optimization.

With Quake, the differences between Native Client and the normal executable are, for practical purposes, indistinguishable. See Table 8 for the comparison. We observed very little non-determinism between runs. The test plays the same sequence of events regardless of frame rate. Slight variances in frame rate can still occur due to the OS thread scheduler and pressure applied to the shared caches from other processes. Although Quake uses software rendering, the performance of the final bitmap transfer to the user's desktop may depend on how busy the video device is.

Run #	Native Client	Linux Executable
1	143.2	142.9
2	143.6	143.4
3	144.2	143.5
Average	143.7	143.3

Table 8: Quake performance comparison. Numbers are in frames per second.

5. Discussion

As described above, Native Client has inner and outer sandboxes, redundant barriers to protect native operating system interfaces. Additional measures such as a CPU blacklist and NaCl module blacklist will also be deployed, and we may deploy whitelists if we determine they are needed to secure the system. We have also considered more elaborate measures, although as they are speculative and unimplemented we don't describe them here. We see public discussion and open feedback as critical to hardening this technology, and informing our decisions about what security mechanisms to include in the system.

We expect Native Client to be well suited to simple, computationally intensive extensions for web applications, specifically in domains such as physical simulation, language processing, and high-performance graphics rendering. Over time, if we can provide convenient DOM access, we hope to enable web-based applications that run primarily in native code, with a thin JavaScript wrapper. There are also applications of this technology outside of the browser; these are beyond our current focus.

We have developed and tested Native Client on Ubuntu Linux, MacOS and Microsoft Windows XP. Overall we are satisfied with the interaction of Native Client with these operating systems. That being said, there are a few areas where operating system support might helpful. Popular operating systems generally require all threads to use a flat addressing model in order to deliver exceptions correctly. Use of segmented memory prevents these systems from interpreting the stack pointer and other essential thread state. Better segment support in the operating system might allow us to resolve this problem and allow for better hardware exception support in untrusted code. If the OS recognized a distinguished thread to receive all exceptions, that would allow Native Client to receive exceptions in a trusted thread.

Native Client would also benefit from more consistent enabling of LDT access across popular x86 operating sys-

tems. As an interesting alternative to maintaining system call access as provided by most current systems, a system call for mapping the LDT directly into user space would remove a kernel system call from the path for NaCl thread creation, relevant for modules with a large number of threads.

With respect to programming languages and language implementations, we are encouraged by our initial experience with Native Client and the GNU tool chain, and are looking at porting other compilers. We have also ported two language interpreters, Lua and awk, and are aware of efforts to port other popular interpreted languages. While it would be challenging to support JITted languages such as Java, we are hopeful that Native Client might someday allow developers to use their language of choice in the browser rather than being restricted to only JavaScript.

6. Related Work

Techniques for safely executing 3rd-party code generally fall into three categories: system request moderation, fault isolation (including virtualization), and trust with authentication.

6.1. System Request Moderation

Kernel-based mechanisms such as user-id based access control, systrace [50] and ptrace [60] are familiar facilities on Unix-like systems. Many previous projects have explored use of these mechanisms for containing untrusted code [24], [33], [34], [36], [52], most recently Android [9], [27] from Google and Xax [17] from Microsoft Research. Android uses a sandbox for running 3rd party applications. Each Android application is run as a different Linux user, and a containment system partitions system call activity into permission groups such as "Network communication" and "Your personal information". User acknowledgment of required permissions is required prior to installing a 3rd party application. User separation inherently denies potentially useful intercommunication. To provide intercommunication, Android formed a permissions model atop the Binder interprocess communication mechanism, the Intent sytem and ContentProvider data access model. [9]

Xax is perhaps the most similar work to Native Client in terms of goals, although their implementation approach is quite different, using system call interception based on ptrace on Linux and a kernel device driver on Windows. We considered such a kernel-based approach very early in our work but rejected it as impractical due to concerns about supportability. In particular we note that the Xax Windows implementation requires a kernel-mode device driver that must be updated for each supported Windows build, a scheme we imagine onerous even if implemented by the OS vendor themselves. There are known defects in ptrace containment[6] that Xax does not address. Although the Xax authors do recognize one such issue in their paper, a simple search at Mitre's Common Vulnerabilities and Exposures site[7] documents forty-one different ptrace-related issues. Because of its pure user-space inner sandbox, Native Client is less vulnerable to these difficult kernel issues. Xax is also vulnerable to denial-of-service attacks based on x86 errata that can cause a machine to hang or reboot [31], [38]. Because Native Client examines every instruction and rejects modules with instructions it finds suspect, it significantly reduces the attack surface with respect to invalid instructions, and further it includes relevant mechanism for defending against new exploits should they be found.

Because the Xax sandbox functions at the process boundary, it fails to isolate untrusted code when shared application components such as DLLs are involuntarily injected by the operating system, an issue both for security and for portability of untrusted code. In contrast, the Native Client inner sandbox creates a security sub-domain within a native operating system process. Apart from these security differences we note that Xax does not support threading, which we considered essential given the trend towards multicore CPUs.

The Linux seccomp[8] facility also constrains Linux processes at the system call interface, allowing a process to enter a mode where only _exit(), read(), and write() system calls are permitted.

6.2. Fault Isolation

Native Client applies concepts of software fault isolation and proof-carrying code that have been extensively discussed in the research literature. Our data integrity scheme is a straightforward application of segmented memory as implemented in the Intel 80386 [14]. Our current control flow integrity technique builds on the seminal work by Wahbe, Lucco, Anderson and Graham [62]. Like Wahbe et al., Native Client expresses sandboxing constraints directly in native machine instructions rather than using a virtual machine or other ISA-portable representation. Native Client extends this previous work with specific mechanisms to achieve safety for the x86 [4], [14], [32] ring-3 instruction set architecture (ISA), using several techniques first described by McCamant and Morrisett [40]. Native Client uses a static validator rather than a trusted compiler, similar to validators described for other systems [19], [40], [41], [49], applying the concept of proof-carrying code [46].

After the notion of software fault isolation was popularized by Wahbe et al., researchers described complementary and alternative systems. A few [1], [19], [40], [41],

6. http://www.linuxhq.com/kernel/v2.4/36-rc1/Documentation/ptrace.txt

7. For example, see http://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=ptrace

8. See linux/kernel/seccomp.c

[49], [56] work directly with x86 machine code. Others are based on intermediate program representations, such as type-safe languages [28], [45], [47], [59], abstract virtual machines [3], [20], [21], [39], or compiler intermediate representations [53]. They use a portable representation, allowing ISA portability but creating a performance obstacle that we avoid by working directly with machine code. A further advantage of expressing sandboxing directly in machine code is that it does not require a trusted compiler. This greatly reduces the size of the trusted computing base [61], and obviates the need for cryptographic signatures from the compiler. Apart from simplifying the security implementation, this has the further benefit in Native Client of opening the system to 3rd-party tool chains.

Compared to Native Client, CFI [1] provides finer-grained control flow integrity. Whereas our system only guarantees indirect control flow will target an aligned address in the text segment, CFI can restrict a specific control transfer to a fairly arbitrary subset of known targets. While this more precise control is possibly useful in some scenarios, such as insuring integrity of translation from a high-level language, it is not useful for Native Client, since we intend to permit quite arbitrary control flow, even hand-coded assembler, as long as execution remains in known text and targets are aligned. At the same time, CFI overhead is a factor of three higher on average (15% vs. 5% on SPEC2000), not surprising since its instrumentation sequences are much longer than Native Client's, both in terms of size and instruction count. XFI [19] adds data sandboxing to CFI control flow checks, with additional overhead. By contrast Native Client gets data integrity for free from x86 segments.

Other recent systems have explored mechanisms for enabling safe side effects with measured trust. NaCl resource descriptors are analogous to capabilities in systems such as EROS [55]. Singularity channels [30] serve an analogous role. DTrace [11], Systemtap [49] and XFI [19] have related mechanisms.

A number of projects have explored isolating untrusted kernel extensions. SPIN and VINO take different approaches to implementing safety. SPIN chose a type-safe language, Modula-3 [47], together with a trusted compiler tool chain, for implementing extensions. VINO, like Native Client and the original work by Wahbe et al., used software fault isolation based on sandboxing of machine instructions. Like Native Client, VINO used a modified compilation toolchain to add sandboxing instructions to x86 machine code, using C++ for implementing extensions. Unlike Native Client, VINO had no binary validator, relying on a trusted compiler. We note that a validator for VINO would be more difficult than that of Native Client, as its validator would have had to enforce data reference integrity, achieved in Native Client with 80386 segments.

The Nooks system [58] enhances operating system kernel reliability by isolating trusted kernel code from untrusted device driver modules using a transparent OS layer called the Nooks Isolation Manager (NIM). Like Native Client, NIM uses memory protection to isolate untrusted modules. As the NIM operates in the kernel, x86 segments are not available. The NIM instead uses a private page table for each extension module. To change protection domains, the NIM updates the x86 page table base address, an operation that has the side effect of flushing the x86 Translation Lookaside Buffer (TLB). In this way, NIM's use of page tables suggests an alternative to segment protection as used by Native Client. While a performance analysis of these two approaches would likely expose interesting differences, the comparison is moot on the x86 as one mechanism is available only within the kernel and the other only outside the kernel. A critical distinction between Nooks and Native Client is that Nooks is designed only to protect against unintentional bugs, not abuse. In contrast, Native Client must be resistant to attempted deliberate abuse, mandating our mechanisms for reliable x86 disassembly and control flow integrity. These mechanisms have no analog in Nooks.

There are many environments based on a virtual-machine architecture that provide safe execution and some fraction of native performance [3], [6], [7], [20], [28], [39], [53], [63]. While recognizing the excellent fault-isolation provided by these systems, we made a deliberate choice against virtualization in Native Client, as it is generally inconsistent with, or irrelevant to, our goals of OS neutrality, browser neutrality, and peak native performance.

More recently, kernel extensions have been used for operating system monitoring. DTrace [11] incorporated a VM interpreter into the Solaris kernel for safe execution, and provided a set of kernel instrumentation points and output facilities analogous to Native Client's safe side effects. Systemtap [49] provides similar capabilities to DTrace, but uses x86 native code for extensions rather than an interpreted language in a VM.

6.3. Trust with Authentication

Perhaps the most prevalent example of using native code in web content is Microsoft's ActiveX [15]. ActiveX controls rely on a trust model to provide security, with controls cryptographically signed using Microsoft's proprietary Authenticode system [43], and only permitted to run once a user has indicated they trust the publisher. This dependency on the user making prudent trust decisions is commonly exploited. ActiveX provides no guarantee that a trusted control is safe, and even when the control itself is not inherently malicious, defects in the control can be exploited, often permitting execution of arbitrary code. To mitigate this issue, Microsoft maintains a blacklist of controls deemed unsafe [42]. In contrast, Native Client is designed to prevent such exploitation, even for flawed NaCl modules.

7. Conclusions

This paper has described Native Client, a system for incorporating untrusted x86 native code into an application that runs in a web browser. In addition to creating a barrier against undesirable side effects, NaCl modules are portable both across operating systems and across web browsers, and supports performance-oriented features such as threading and vectorization instructions. We believe the NaCl inner sandbox is extremely robust; regardless we provide additional redundant mechanisms to provide defense-in-depth.

In our experience we have found porting existing Linux/gcc code to Native Client is straightforward, and that the performance penalty for the sandbox is small, particularly in the compute-bound scenarios for which the system is designed.

By describing Native Client here and making it available as open source, we hope to encourage community scrutiny and contributions. We believe this feedback together with our continued diligence will enable us to create a system that achieves a superior level of safety than previous native code web technologies.

Acknowledgments

Many people have contributed to the direction and the development of Native Client; we acknowledge a few of them here. The project was conceived based on an idea from Matt Papakipos. Jeremy Lau, Brad Nelson, John Grabowski, Kathy Walrath and Geoff Pike have made valuable contributions to the implementation and evaluation of the system. Thanks also to Danny Berlin, Chris DiBona, and Rebecca Ward. We thank Sundar Pichai and Henry Bridge for their role in shaping the project direction. We'd also like to thank Dick Sites for his thoughtful feedback on an earlier version of this paper.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity: Principles, implementations, and applications. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, November 2005.

[2] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. pages 93–112, 1986.

[3] A. Adl-Tabatabai, G. Langdale, S. Lucco, and R. Wahbe. Efficient and language-independent mobile programs. *SIGPLAN Not.*, 31(5):127–136, 1996.

[4] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual, Volume 1: Application Programming*. Advanced Micro Devices, September 2007. Publication number: 24592.

[5] Autoconf. http://www.gnu.org/software/autoconf/.

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *19th ACM Symposium on Operating Systems Principles*, pages 164–177, 2003.

[7] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems*, 15(4):412–447, November 1997.

[8] Bullet physics SDK. http://www.bulletphysics.com.

[9] J. Burns. Developing secure mobile applications for android. http://isecpartners.com/files/iSEC_Securing_Android_Apps.pdf, 2008.

[10] K. Campbell, L. Gordon, M. Loeb, and L. Zhou. The economic cost of publicly announced information security breaches: empirical evidence from the stock market. *Journal of Computer Security*, 11(3):431–448, 2003.

[11] B. Cantrill, M. Shapiro, and A. Leventhal. Dynamic instrumentation of production systems. In *2004 USENIX Annual Technical Conference*, June 2004.

[12] D. R. Cheriton. The V distributed system. *Communications of the ACM*, 31:314–333, 1988.

[13] F. B. Cohen. Defense-in-depth against computer viruses. *Computers and Security*, 11(6):565–584, 1993.

[14] J. Crawford and P. Gelsinger. *Programming 80386*. Sybex Inc., 1991.

[15] A. Denning. *ActiveX Controls Inside Out*. Microsoft Press, May 1997.

[16] Directorate for Command, Control, Communications and Computer Systems, U.S. Department of Defense Joint Staff. Information assurance through defense-in-depth. Technical report, Directorate for Command, Control, Communications and Computer Systems, U.S. Department of Defense Joint Staff, February 2000.

[17] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch. Leveraging legacy code to deploy desktop applications on the web. In *Proceedings of the 2008 Symposium on Operating System Design and Implementation*, December 2008.

[18] M. Eisler (editor). XDR: External data representation. Internet RFC 4506, 2006.

[19] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. Necula. XFI: Software guards for system address spaces. In *OSDI '06: 7th Symposium on Operating Systems Design And Implementation*, pages 75–88, November 2006.

[20] B. Ford. VXA: A virtual architecture for durable compressed archives. In *USENIX File and Storage Technologies*, December 2005.

[21] B. Ford and R. Cox. Vx32: Lightweight user-level sandboxing on the x86. In *2008 USENIX Annual Technical Conference*, June 2008.

[22] The GNU compiler collection. http://gcc.gnu.org.

[23] GNU binutils. http://www.gnu.org/software/binutils/.

[24] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure enviroment for untrusted helper applications. In *Proceedings of the 6th USENIX Security Symposium*, 1996.

[25] D. Golub, A. Dean, R. Forin, and R. Rashid. UNIX as an application program. In *Proceedings of the Summer 1990 USENIX Conference*, pages 87–95, 1990.

[26] Google Inc. Protocol buffers. http://code.google.com/p/protobuf/.

[27] Google Inc. Android—an open handset alliance project. http://code.google.com/android, 2007.

[28] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, 2000.

[29] B. Gough and R. Stallman. *An Introduction to GCC*. Network Theory, Ltd., 2004.

[30] G. Hunt, J. Larus, M. Abadi, M. Aiken, P. Barham, M. Fahndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill. An overview of the Singularity project. Technical Report MSR-TR-2005-135, Microsoft Research, October 2005.

[31] Intel Corporation. Intel Pentium processor invalid instruction errata overview. http://support.intel.com/support/processor/pentium/ppiie/index.html.

[32] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developers Manual, Volume 1: Basic Architecture*. Intel Corporation, August 2007. Order Number: 253655-024US.

[33] S. Ioannidis and S. M. Bellovin. Building a secure web browser. In *USENIX Annual Technical Conference, FREENIX Track*, pages 127–134, 2001.

[34] S. Ioannidis, S. M. Bellovin, and J. M. Smith. Sub-operating systems: A new approach to application security. In *Proceedings of the 10th ACM SIGOPS European Workshop*, 2002.

[35] Jam 2.1 user's guide. http://javagen.com/jam/.

[36] C. Jensen and D. Hagimont. Protection wrappers: a simple and portable sandbox for untrusted applications. In *Proceedings of the 8th ACM SIGOPS European Systems Conference*, pages 104–110, 1998.

[37] W. Joy, E. Cooper, R. Fabry, S. Leffler, K. McKusick, and D. Mosher. 4.2 BSD system manual. Technical report, Computer Systems Research Group, University of California, Berkeley, 1983.

[38] K. Kaspersky and A. Chang. Remote code execution through Intel CPU bugs. In *Hack In The Box (HITB) 2008 Malaysia Conference*.

[39] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Prentice Hall, 1999.

[40] S. McCamant and G. Morrisett. Efficient, verifiable binary sandboxing for a CISC architecture. Technical Report MIT-CSAIL-TR-2005-030, 2005.

[41] S. McCamant and G. Morrisett. Evaluating SFI for a CISC architecture. In *15th USENIX Security Symposium*, August 2006.

[42] Microsoft Corporation. The kill-bit faq - part 1 of 3. *Microsoft Security Vulnerability Research and Defense (Blog)*.

[43] Microsoft Corporation. Signing and checking code with Authenticode. http://msdn.microsoft.com/en-us/library/ms537364(VS.85).aspx.

[44] Microsoft Corporation. Structured exception handling. http://msdn.microsoft.com/en-us/library/ms680657(VS.85).aspx, 2008.

[45] G. Morrisett, K. Crary, N. Glew, and D. Walker. Stack-based typed assembly language. *Journal of Functional Programming*, 12(1):43–88, 2002.

[46] G. Necula. Proof carrying code. In *Principles of Programming Languages*, 1997.

[47] G. Nelson (editor). *System Programming in Modula-3*. Prentice-Hall, 1991.

[48] Netscape Corporation. Gecko plugin API reference. http://developer.mozilla.org/en/docs/Gecko_Plugin_API_Reference.

[49] V. Prasad, W. Cohen, F. Eigler, M. Hunt, J. Keniston, and J. Chen. Locating system problems using dynamic instrumentation. In *2005 Ottawa Linux Symposium*, pages 49–64, July 2005.

[50] N. Provos. Improving host security with system call policies. In *USENIX Security Symposium*, August 2003.

[51] J. Reinders. *Intel Thread Building Blocks*. O'Reilly & Associates, 2007.

[52] J. G. Richard West. User-level sandboxing: a safe and efficient mechanism for extensibility. Technical Report TR-2003-014, Boston University, Computer Science Department, Boston, MA, 2003.

[53] J. Richter. *CLR via C#, Second Edition*. Microsoft Press, 2006.

[54] M. Savage. Cost of computer viruses top $10 billion already this year. *ChannelWeb*, August 2001.

[55] J. Shapiro, J. Smith, and D. Farber. EROS: a fast capability system. In *Symposium on Operating System Principles*, pages 170–185, 1999.

[56] C. Small. MiSFIT: A tool for constructing safe extensible C++ systems. In *Proceedings of the Third USENIX Conference on Object-Oriented Technologies*, June 1997.

[57] B. Stroustrup. *The C++ Programming Language: Second Edition*. Addison-Wesley, 1997.

[58] M. Swift, M. Annamalai, B. Bershad, and H. Levy. Recovering device drivers. In *6th USENIX Symposium on Operating Systems Design and Implementation*, December 2004.

[59] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: a type-directed optimizing compiler for ML. In *PLDI '96: Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, pages 181–192, New York, NY, USA, 1996. ACM.

[60] W. Tarreau. ptrace documentation. http://www.linuxhq.com/kernel/v2.4/36-rc1/Documentation/ptrace.txt, 2007.

[61] U. S. Department of Defense, Computer Security Center. Trusted computer system evaluation criteria, December 1985.

[62] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, December 1993.

[63] C. Waldspurger. Memory resource management in VMware ESX Server. In *5th Symposium on Operating Systems Design and Implementation*, December 2002.

[64] *Document Object Model (DOM) Level 1 Specification*. Number REC-DOM-Level-1-19981001. World Wide Web Consortium, October 1998.

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 11

Suppose an adversary discovers a bug in NaCl where the checker incorrectly determines the length of a particular x86 instruction. How could an adversary exploit this to escape the inner sandbox?

hmm
buffer overflow
builds an instruction that redirects execution elsewhere

Q1 is NaCl ready for prime time

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 Friday, 12-Oct-2012 23:31:47 EDT*

10/15

Paper Question 11

Michael Plasmeier

If an instruction length is not detected, this could cause the instruction length to overflow and insert unchecked code into the execution flow. This has the potential to allow arbitrary code execution.

6858

10/15

Lecture 11

Native Client

Pape by Google

Ships w/ Chrome

Couple of changes, but core idea is in paper

Before: could only run JS + Flash

limited capabilities

Slow

- better, but still an order of magnitude worse

no C, Haskell

no legacy C, C++ app

could run native code - but had to trust devs

How to sandbox x86 code?

- OS sandbox in process or VM
  - OKWS
  - Seccomp
  - Capsicum
- ~~ActiveX~~ Active X - plugins signed
- Browser plugins - shared lib that runs in browser process

No guarantees/security

(2)

Heavy weight:

- does not really need to be
  - could just run app
- though outer sandbox does use a number of these mechanisms
- also they don't want OS-specific mechanisms
  - wanted cross platform portability
- must provide some access to trusted object
  - IPC channel
    - but could do through any mechanism

Paper: don't trust OS vendors to do it right
- think they can do it better

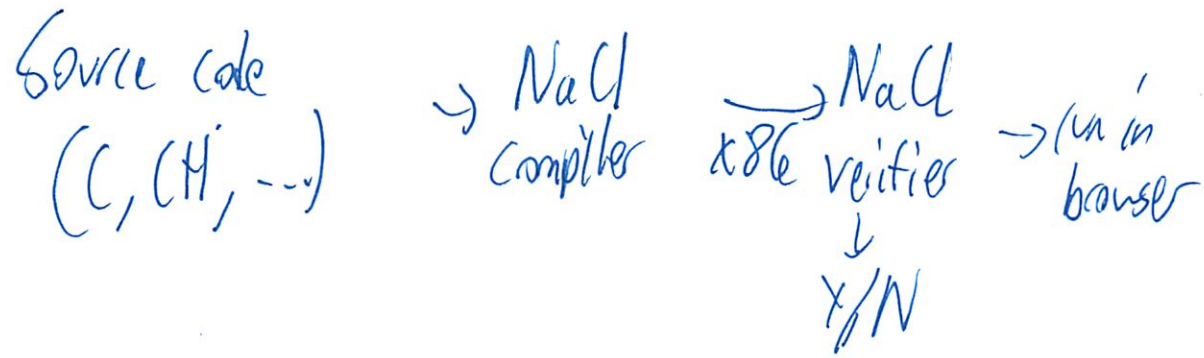
CPU bugs that let untrusted code hang your machine
- the others are not safe from CPU bugs
- NaCl wanted to prevent this issue

(3)

Xen also couldn't have done nested virtualization
So this approach is better

Software fault isolation - SFI

looks like JS rewriting paper we read
run through whole thing
rewrite
then run rewritten code



internal disassembly of instructions

So they don't have to trust the compiler
dev runs compiler
dev could even write assembly
gcc is very big

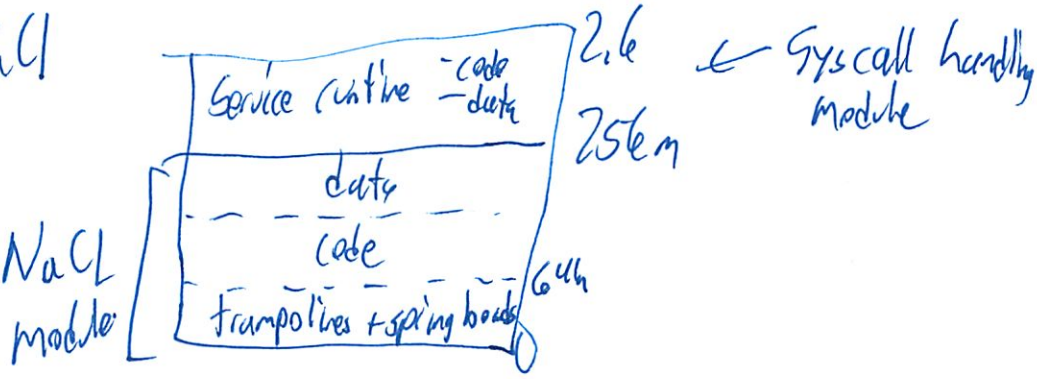
(9)

Verifier is only 600 LoC
Know several thousand
- but still pretty small

You could run this code directly now
They want to provide addl safety w/ the outer layer

Verifier's Safety Goals

1. No disallowed instructions
   - sys calls
   - int instructions
2. Restrict memory addresses
   - Linux - addr you can generally use your addr as you want
   - NaCl



So NaCl can only touch within this
0 → 256 m region

So after verifying code, we can let it run

How do we know what code will be run
if we jump to the middle?

But it could modify code?
Or execute data?

- Code section marked non writable
- Data section " non executable
  ↳ mprotect in linux

Nice Scan instructions
but x86 allows variable lenght instructions
25 00 80 00 00
AND %eax, $0x00008000

(6)

But if decode starting at CD

CD 80

↓

int 0x80 = syscall

(So should we check at each possible start?
But this would get unworkably complicated

So try to enforce can only jump to
start/end at 32 bit gap

Reliable Dissusembly

Start from 64k

Disassemble

- record the addr of all instr
- if jump + must go to instr
  - Static - constant in code
  - Computed jumps - jmp * %eax
    - statically impossible to tell

(9)

So what do they do?

Like FPGS array indexing paper

Must add some runtime instrumentation

So it checks it at runtime

NaCL makes sure jumps have a special seq

```
AND $ 0xffffffe0, % eax
JMP * % eax
```

So 0 out low order bits

So target is 32 bytes alligned

Rule that no instruction spans 32 byte boundry

Inst can only jump to 32 byte multiple

Each byte must be one they checked

Do 2 passes to be sure

(8)

32 byte is sweet spot
Some ins 15-16 bytes long

So job of compiler that it jmps to 32 bit boundry

So they modified gcc to add nops to pad it

The ffffffe0
high bits preserved
low bits remark
5 low bits of 0

1111...111160000
5 0s
gets up to 32 bit
Since removes this part of offset

(9)

A return is a computed jump
So this is disallowed
Compiler must load ret value into registe
Clears the lower 5 bits
Then jumps there

So that it jmps to a verified instruction location

Whole bunch of rules in table 1

C1 - executable memory non writable
C2 - linked at 0/64k
C3 - all computed jmps use the 2-instr
AND and JMP to only Call on 32 bit

C4 binary padded w/ 1 or more HLT instructions
to prevent you from running into data
- next is non writable, but this is extra protection

(10)

C5 No instruction can span 32 byte boundry

C5b No 2 instruction AND + JMP pair spans boundry

↳ So could JMP to a unchecked JMP!
(JMPs do not all need to use same register)

C6/C7 all instructions / jump targets reachable

HW Q: What will break?

Offset wrong

So verifier gets boundry messed up

So could possibly jump to wrong place

This has kinda happened in NaCl

Also need to make sure it does not jump to the service runtime: (above 256m)

Could 0 out higher bits

0000 1...1 10 0000
stays in low 256 mb

(1)

But must be able to call the library

Must control reads + writes up there

So use segmentation

Since ~~can~~ otherwise compated bytes bigger

<11110000 <01...100000

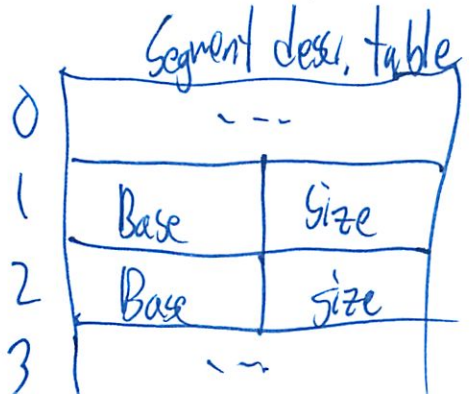
AND	3	5
JMP	2	2
	5	7

2 bytes more

too expensive!

Segmentation

Window into memory

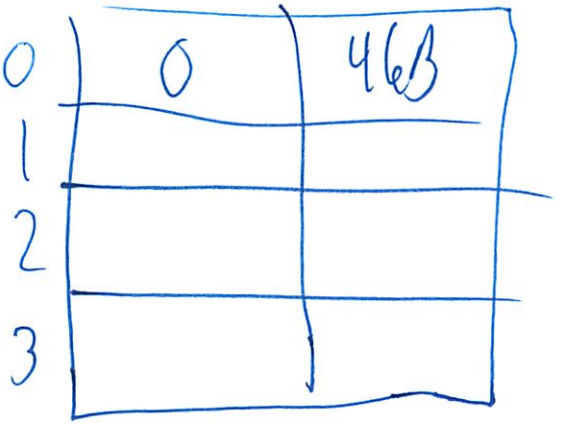


(12)

Segment selector register

integer indexes into table

%cs → 1
%ds → 2
%es → 2
%fs → 2
%gs → 2
%ss → 2



every instruction that refs memory address does it w/ reference to its segment

%ds is default seg

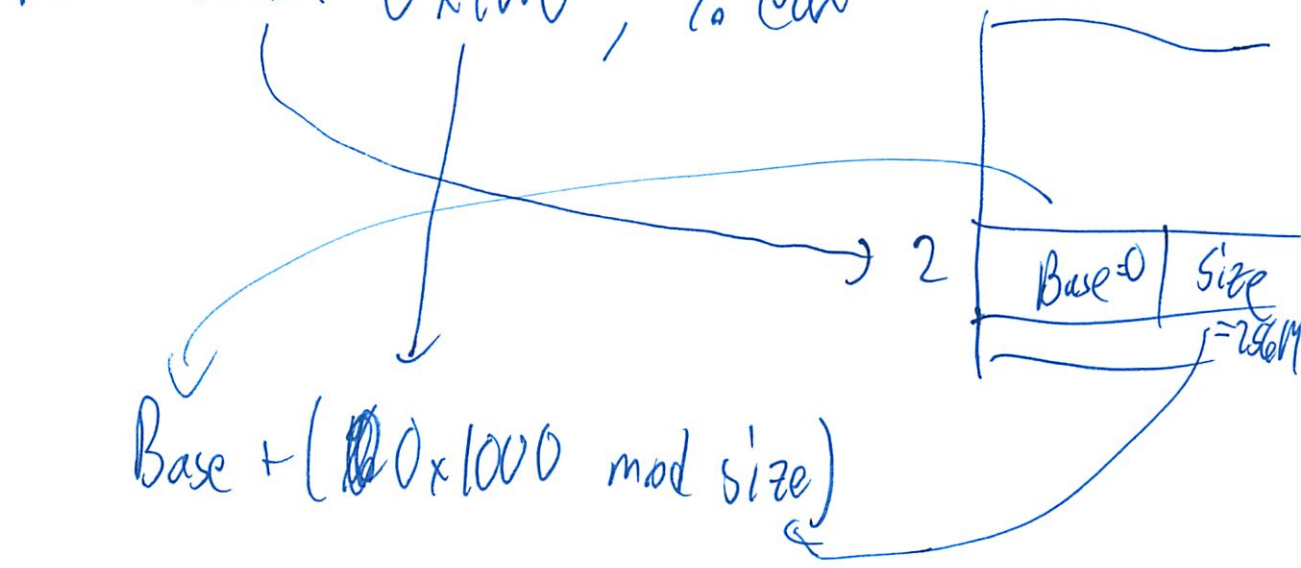
every object refers to a segment

mov %ds:0x1000, %eax

usually don't see

(13)

mov %ds: 0x1000, %eax



(missed explination)

%cs = %ds = %es

So no addr can escape this section of memory

Could make it work w/o halts
But make it clear
Segments wrap around in x86
Would need to check in validate

So could we do w/o segmentation?

What about data accesses w/o segmentation hw?

Would have to apply AND trick to every memory addr w/ a check

But that is a lot of instructions!

x64 turned off segmentation since legacy

So NaCl masks at low and high end bits

Segmentation from Multics - 72 bit pointers

Intel copied this w/ x86

Virtual Mem + page protection better

This was just slowing down CPU

So they got rid of it to make things faster

VM were used the same trick

but then real hw support came out for them

(15)

So what if buffer overflow in a module?

Can't inject shell code since flags

but could you return to libC

but will on AND, JMP

would have to JMP to unexpected place in module

But then every next JMP is verified

So can't JMP out of sandbox

Speed

Actually faster in some cases!

But generally slows down

NOPs fill up icache - needs to be flushed more often

flushing icache in right place

no real reason why just luck

(14)

Outside World

Code can just compute

But how does it report back, store, dam, etc

Trampoline

Sandbox → Service runtime

Must undo sandbox & JMP to service runtime

undo segmentation protection →

```
MOV $3, %ds, %ss
long jmp %cs, $0x7fff ffff(%rax)
```

JMP * %rax

(verifier does not verify this code - comes w/ NaCl assumes its correct

Put msg want to send in registers

Jmps to service runtime

Springboard

Service runtime → Sandbox

Generate sandbox

JMP to inside code

```
mov $7, %es/ds
```

(17)

Why can't regular code do sure thing?

disallowed
- Changing segment selectors

Springboard can only be invoked from service runtime
But halt instr is 1st chunk
Service runtime can jump to un-aligned addr

NPAPI

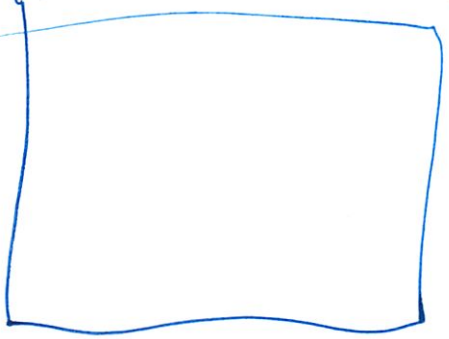
Lots of trusted components
- verifier
- trampoline, spring board
- svc runtime
- browser

Service runtime's instead of normal return returns to spring board
- it knows that (hard coded)

(18)

modify_ldt()

↳ lets ya set segmentation description table



Google uses this for running Python in App Engine
on their servers
to extra protect the Py interpreter

x86 memory segmentation

From Wikipedia, the free encyclopedia

x86 memory segmentation refers to the implementation of memory segmentation on the x86 architecture. Certain portions of the memory may be addressed by a single index register without changing a 16-bit **segment selector**. In real mode or V86 mode, a segment is always 65,536 bytes in size (using 16-bit offsets). In protected mode, a segment can have variable length. Segments can overlap.

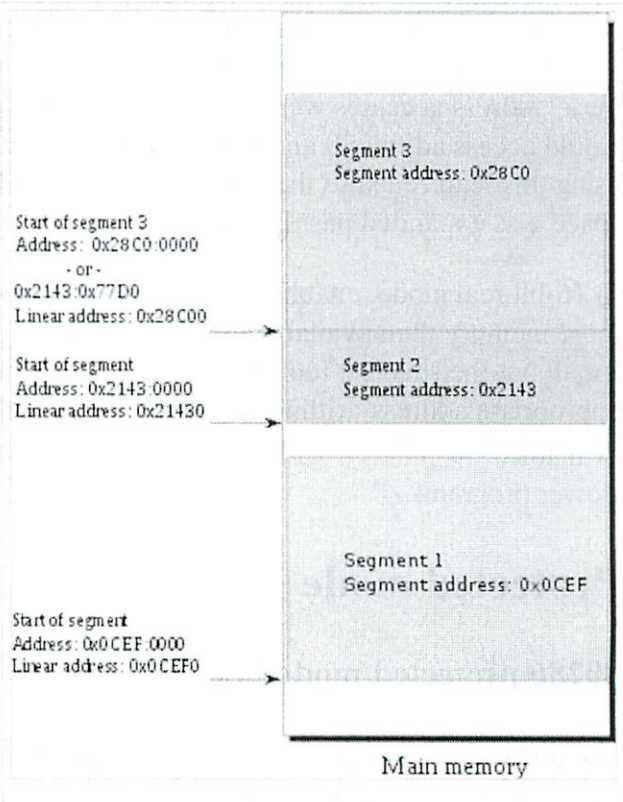
Contents

Read 10/22 opt

what is it first of all

Real mode

In real mode, the 16-bit segment selector is interpreted as the most significant 16 bits of a linear 20-bit address, called a segment address, of which the remaining four least significant bits are all zeros. The segment address is always added with a 16-bit offset to yield a *linear* address, which is the same as physical address in this mode. For instance, the segmented address 06EFh:1234h has a segment selector of 06EFh, representing a segment address of 06EF0h, to which we add the offset, yielding the linear address 06EF0h + 1234h = 08124h (hexadecimal).



Because of the way the segment address and offset are added, a single linear address can be mapped to up to 4096 distinct segment:offset pairs. For example, the linear address 08124h can have the segmented addresses 06EFh:1234h, 0812h:0004h, 0000h:8124h, etc. This could be confusing to programmers accustomed to unique addressing schemes, but it can also be used to advantage, for example when addressing multiple nested data structures. While real mode segments are always

Three segments in real mode memory (click on image to enlarge). Note the overlap between segment 2 and segment 3; the bytes in the turquoise area can be used from both segment selectors.

64 KiB long, the practical effect is only that no segment can be longer than 64 KiB, rather than that every segment *must* be 64 KiB long. Because there is no protection or privilege limitation in real mode, even if a segment could be defined to be smaller than 64 KiB, it would still be entirely up to the programs to coordinate and keep within the bounds of their segments, as any program can always access any memory (since it can arbitrarily set segment selectors to change segment addresses with absolutely no supervision). Therefore, real mode can just as well be imagined as having a variable length for each segment, in the range 1 to 65536 bytes, that is just not enforced by the CPU.

(Note that the leading zeros of the linear address, segmented addresses, and the segment and offset fields, which are usually neglected, were shown here for clarity.)

The effective 20-bit address space of real mode limits the addressable memory to 2^{20} bytes, or 1,048,576 bytes. This derived directly from the hardware design of the Intel 8086 (and, subsequently, the closely related 8088), which had exactly 20 address pins. (Both were packaged in 40-pin DIP packages; even with only 20 address lines, the address and data buses were multiplexed to fit all the address and data lines within the limited pin count.)

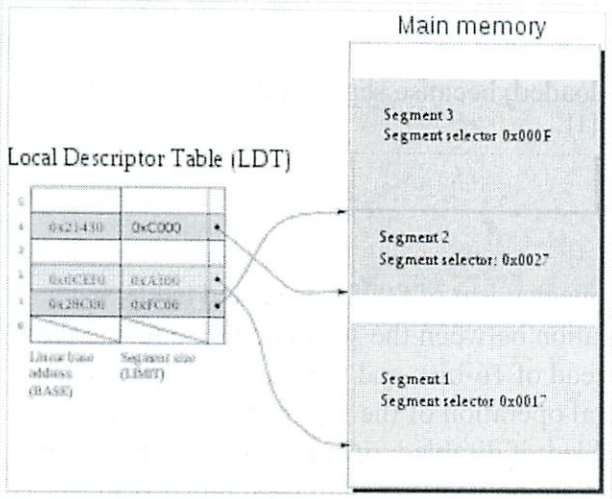
Each segment begins at a multiple of 16 bytes, from the beginning of the linear (flat) address space. That is, at 16 byte intervals. Since all segments are 64 KiB long, this explains the huge overlap that can occur between segments and that any location in the linear memory address space can be accessed with many segment:offset pairs. The actual location of the beginning of a segment in the linear address space can be calculated with segment×16. A segment value of 0Ch (12) would give an linear address at C0h (192) in the linear address space. The address offset can then be added to this number. 0Ch:0Fh (12:15) would be C0h+0Fh=CFh (192+15=207), CFh (207) being the linear address. Such address translations are carried out by the segmentation unit of the CPU. The last segment, FFFFh (65535), begins at linear address FFFF0h (1048560), 16 bytes before the end of the 20 bit address space, and thus, can access, with an offset of up to 65,536 bytes, up to 65,520 (65536−16) bytes past the end of the 20 bit 8088 address space. On the 8088, these address accesses were wrapped around to the beginning of the address space such that 65535:16 would access address 0 and 65533:1000 would access address 952 of the linear address space. Programmers using this feature led to the Gate A20 compatibility issues in later CPU generations, where the linear address space was expanded past 20 bits.

In 16-bit real mode, enabling applications to make use of multiple memory segments (in order to access more memory than available in any one 64K-segment) is quite complex, but was viewed as a necessary evil for all but the smallest tools (which could do with less memory). The root of the problem is that no appropriate address-arithmetic instructions suitable for flat addressing of the entire memory range are available.[*citation needed*] Flat addressing is possible by applying multiple instructions, which however leads to slower programs.

Protected mode

80286 protected mode

The 80286's protected mode extends the processor's address space to 2^{24} bytes (16 megabytes), but not by adjusting the shift value. Instead, the 16-bit segment registers now contain an index into a table of segment



Three segments in protected mode memory (click on image to enlarge), with the **local descriptor table**.

descriptors containing 24-bit base addresses to which the offset is added. To support old software, the processor starts up in "real mode", a mode in which it uses the segmented addressing model of the 8086. There is a small difference though: the resulting physical address is no longer truncated to 20 bits, so real mode pointers (but not 8086 pointers) can now refer to addresses between 100000_h and $10FFEF_h$. This roughly 64-kilobyte region of memory was known as the High Memory Area (HMA), and later versions of MS-DOS could use it to increase the available "conventional" memory (i.e. within the first MiB). With the addition of the HMA, the total address space is approximately 1.06 MiB. Though the 80286 does not truncate real-mode addresses to 20 bits, a system containing an 80286 can do so with hardware external to the processor, by gating off the 21st address line, the A20 line. The IBM PC AT provided the hardware to do this (for full backward compatibility with software for the original IBM PC and PC/XT models), and so all subsequent "AT-class" PC clones did as well.

The protected mode segmentation system, present in the 80286 and later x86 CPUs, can be used to enforce separation of unprivileged processes, but most 32-bit operating systems uses the paging mechanism introduced with the 80386 for this purpose instead. Such systems set all segment registers to point to a segment descriptor with offset=0 and limit=2^{32}, giving an application full access to a 32-bit flat virtual address space through any segment register. By this method, normal application code does not have to deal with segment registers at all. This was possible because the 80386 widened the general purpose registers (i.e. the offset registers) to 32 bits. Naturally, the base addresses in the descriptors were also widened to 32 bits.

Detailed Segmentation Unit Workflow

A logical address consists of a 16-bit segment selector (supplying 13+1 address bits) and a 16-bit offset. The segment selector must be located in one of the segment registers. That selector consists of a 2-bit Requested Privilege Level (RPL), a 1-bit Table Indicator (TI), and a 13-bit index.

When attempting address translation of a given logical address, the processor reads the 64-bit segment descriptor structure from either the Global Descriptor Table when TI=0 or the Local Descriptor Table when TI=1. It then performs the privilege check:

$$\max(\mathrm{CPL}, \mathrm{RPL}) \leq \mathrm{DPL}$$

where CPL is the current privilege level (found in the lower 2 bits of the CS register), RPL is the requested privilege level from the segment selector, and DPL is the descriptor privilege level of the segment (found in the descriptor). All privilege levels are integers in the range 0–3, where the lowest number corresponds to the highest privilege.

If the inequality is false, the processor generates a general protection (GP) fault. Otherwise, address translation continues. The processor then takes the 32-bit or 16-bit offset and compares it against the segment limit specified in the segment descriptor. If it is larger, a GP fault is generated. Otherwise, the processor adds the 24-bit segment base, specified in descriptor, to the offset, creating a linear physical

address.

The privilege check is done only when the segment register is loaded, because segment descriptors are cached in hidden parts of the segment registers.[*citation needed*][1]

80386 protected mode

In the 386 and later, protected mode retains the segmentation mechanism of 80286 protected mode, but a paging unit has been added as a second layer of address translation between the segmentation unit and the physical bus. Also, importantly, address offsets are 32-bit (instead of 16-bit), and the segment base in each segment descriptor is also 32-bit (instead of 24-bit). The general operation of the segmentation unit is otherwise unchanged. The paging unit may be enabled or disabled; if disabled, operation is the same as on the 80286. If the paging unit is enabled, addresses in a segment are now virtual addresses, rather than physical addresses as they were on the 80286. That is, the segment starting address, the offset, and the final 32-bit address the segmentation unit derives by adding the two are all virtual (or logical) addresses when the paging unit is enabled. When the segmentation unit generates and validates these 32-bit virtual addresses from a program's logical (46-bit[2]) addresses, the enabled paging unit finally translates these virtual addresses into physical addresses. The physical addresses are 32-bit on the 386, but can be larger on newer processors which support Physical Address Extension.

The 80386 also introduced two new general-purpose data segment registers, FS and GS, to the original set of four segment registers (CS, DS, ES, and SS).

Later developments

The x86-64 architecture does not use segmentation in long mode (64-bit mode). Four of the segment registers: CS, SS, DS, and ES are forced to 0, and the limit to 2^{64}. The segment registers FS and GS can still have a nonzero base address. This allows operating systems to use these segments for special purposes.

For instance, Microsoft Windows on x86-64 uses the GS segment to point to the Thread Environment Block, a small data structure for each thread, which contains information about exception handling, thread-local variables, and other per-thread state. Similarly, the Linux kernel uses the GS segment to store per-CPU data.

Practices

Logical addresses can be explicitly specified in x86 assembly language, e.g. (AT&T syntax):

```
movl $42, %fs:(%eax) ; Equivalent to M[fs:eax]<-42) in RTL
```

However, segment registers are usually used implicitly.

- All CPU instructions are implicitly fetched from the *code segment* specified by the segment selector held in the CS register.
- Most memory references come from the *data segment* specified by the segment selector held in the DS register. These may also come from the extra segment specified by the segment selector held in the ES register, if a segment-override prefix precedes the instruction that makes the memory reference.

Most, but not all, instructions that use DS by default will accept an ES override prefix.

- Processor stack references, either implicitly (e.g. **push** and **pop** instructions) or explicitly (memory accesses using the (E)SP or (E)BP registers) use the *stack segment* specified by the segment selector held in the SS register.
- String instructions (e.g. **stos**, **movs**), along with data segment, also use the *extra segment* specified by the segment selector held in the ES register.

Segmentation cannot be turned off on x86-32 processors (this is true for 64-bit mode as well, but beyond the scope of discussion), so many 32-bit operating systems simulate a flat memory model by setting all segments' bases to 0 in order to make segmentation neutral to programs. For instance, the Linux kernel sets up only 4 general purpose segments:

```
* __KERNEL_CS (Kernel code segment, base=0, limit=4GB, DPL=0)
* __KERNEL_DS (Kernel data segment, base=0, limit=4GB, DPL=0)
* __USER_CS   (User code segment,   base=0, limit=4GB, DPL=3)
* __USER_DS   (User data segment,   base=0, limit=4GB, DPL=3)
```

Since the base is set to 0 in all cases and the limit 4 GiB, the segmentation unit does not affect the addresses the program issues before they arrive at the paging unit. (This, of course, refers to 80386 and later processors, as the earlier x86 processors do not have a paging unit.)

Current Linux also uses GS to point to thread-local storage.

Segments can be defined to be either code, data, or system segments. Additional permission bits are present to make segments read only, read/write, execute, etc.

Note that, in protected mode, code may always modify all segment registers *except* CS (the code segment selector). This is because the current privilege level (CPL) of the processor is stored in the lower 2 bits of the CS register. The only way to raise the processor privilege level (and reload CS) is through the **lcall** (far call) and **int** (interrupt) instructions. Similarly, the only way to lower the privilege level (and reload CS) is through **lret** (far return) and **iret** (interrupt return) instructions. In real mode, code may also modify the CS register by making a far jump (or using an undocumented `POP CS` instruction on the 8086 or 8088)[3]). Of course, in real mode, there are no privilege levels; all programs have absolute unchecked access to all of memory and all CPU instructions.

For more information about segmentation, see the IA-32 manuals freely available on the AMD or Intel websites.

Notes and References

1. ^ "Intel 64 and IA-32 Architectures Software Developer's Manual", Volume 3, "System Programming Guide", published in 2011, Page "Vol. 3A 3-11", the book is written: "*Every segment register has a "visible" part and a "hidden" part. (The hidden part is sometimes referred to as a "descriptor cache" or a "shadow register.") When a segment selector is loaded into the visible part of a segment register, the processor also loads the hidden part of the segment register with the base address, segment limit, and access control information from the segment descriptor pointed to by the segment selector. The information cached in the segment register (visible and hidden) allows the processor to translate addresses without taking extra bus cycles to read the base address and limit from the segment descriptor.*"

2. ^ The 46 bits are 14 bits from a 16-bit segment register (the other two bits are the privilege level) plus a 32-bit offset.
3. ^ `POP CS` must be used with extreme care and has limited usefulness, because it immediately changes the effective address that will be computed from the instruction pointer to fetch the next instruction. Generally, a far jump is much more useful. The existence of `POP CS` is probably an accident, as it follows a pattern of PUSH and POP instruction opcodes for the four segment registers on the 8086 and 8088.

See also

- Intel Memory Model
- THE multiprogramming system

External links

- Home of the IA-32 Intel Architecture Software Developer's Manual (http://www.intel.com/products/processor/manuals/index.htm)
- The Segment:Offset Addressing Scheme (http://mirror.href.com/thestarman/asm/debug/Segments.html)

Retrieved from "http://en.wikipedia.org/w/index.php?title=X86_memory_segmentation&oldid=518434163"
Categories: X86 memory management

CIS-77 Home http://www.c-jump.com/CIS77/CIS77syllabus.htm

Modes of Memory Addressing on x86

Read 10/22 opt

1. Two Real modes of addressing on 80x86

- **Real mode flat model** means

  strictly converting ***one*** *address value* into a *physically meaningful location* in the RAM.

- **Real mode segmented model** means

  strictly converting ***two*** *address values* into a physically meaningful memory location.

  gives access to one megabyte (1,048,576 bytes) of directly addressable memory, known as *real mode memory*.

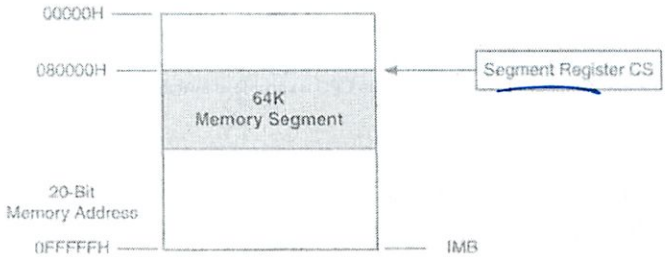
2. Segment Registers

Segment registers are basically *memory pointers* located inside the CPU.

Segment registers point to a place in memory where one of the following things begin:

1. Data storage
2. Code execution.

Example: *code segment register* CS points to a 64K region of memory:



3. Real Mode Segmented Model

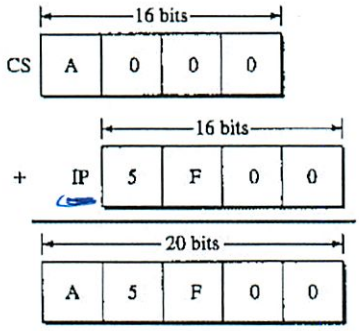
- Segmented organization
  - 16-bit wide segments
- Two components
  - Base (16 bits)
  - Offset (16 bits)
- Two-component specification is called *logical address*, also called *effective address*.
- Logical address translates to a 20-bit *physical address*.

4. Real Mode Segmented Model, Cont.

- Addresses are limited to 20 bits:

  2^{20}=1,048,576 bytes.
- Physical address is generated by adding a
  - 16-bit segment register, shifted left four bits
  - plus a 16 bit-offset.

Generating 20-bit **physical address** in Real Mode:



base

offset

5. Problems Related to Segmentation

- Segmentation often caused grief for programmers who tried to access large data structures:
  - Since an offset cannot exceed 16 bits, you cannot increment beyond 64k.
  - Instead, program must watch out for a 64k boundary and then play games with the segment register.
- This nightmare was originally created to support CP/M-80 programs ported from 8080 chip to 8086.
  - Successful short-term thinking;
  - Catastrophically bad long-term thinking that resulted in never-ending Windows 9x problems!

this sounds silly
What was the original purpose of this

6. Address Space in Real Mode

- Address space in **real mode segmented model** runs from

  00000h to 0FFFFFh,

  within *one megabyte of memory*.
- For compatibility reasons, Pentium CPU is capable of switching itself into real mode segmented model, is effectively *becoming* a good old 8086 chip!

7. Collective Terms for Memory

Size	Decimal	Hex

Byte	1	01H
Word	2	02H
Double word	4	04H
Quad word	8	08H
Ten byte	10	0AH
Paragraph (*)	16	10H
Page, or *page frame*, (almost never used)	256	100H
Segment	65,536	10000H

(*) Paragraphs are almost never used, except in connection with the places where segments may begin.

8. Memory Paragraphs in Real Mode

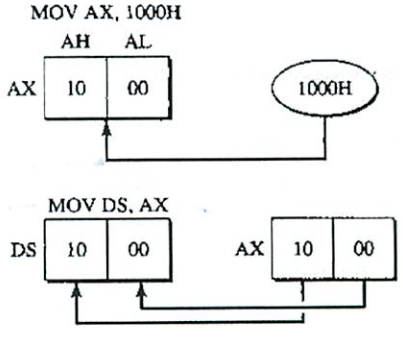
- Any memory address evenly divisible by 16 is called a *paragraph boundary*.
  - The first paragraph boundary is address 0.
  - The second is address 10H (10H is equal to decimal 16.)
  - The third address is 20H, and so on.
- Any paragraph boundary may be considered the *start of a segment*.
- There are 64K different paragraph boundaries where a segment may begin.
- Each paragraph boundary has a number.
- The numbers range from **0** to **64K minus one** (decimal 65,535 or hex 0FFFFh.)
- Any segment may begin at any paragraph boundary.
- The number of the paragraph boundary is called the *segment address*.
- Assembly language program can have up to **four** or **five** segments.
- Segment address is 16 bytes in size.

9. Segment of Memory in Real Mode

- Every byte of memory, accessible by program, is assumed to reside in a segment.
- Segment size varies and can range from 1 byte to 64 Kbytes.
- Nothing is protected within a segment in Real Mode.
- Segments can overlap.
- Initializing the data segment register in 16-bit real mode:
- Numerical (immediate) values cannot be moved directly into the segment register. It is a 2-step process:

```
mov ax, 1000h
mov ds, ax
```

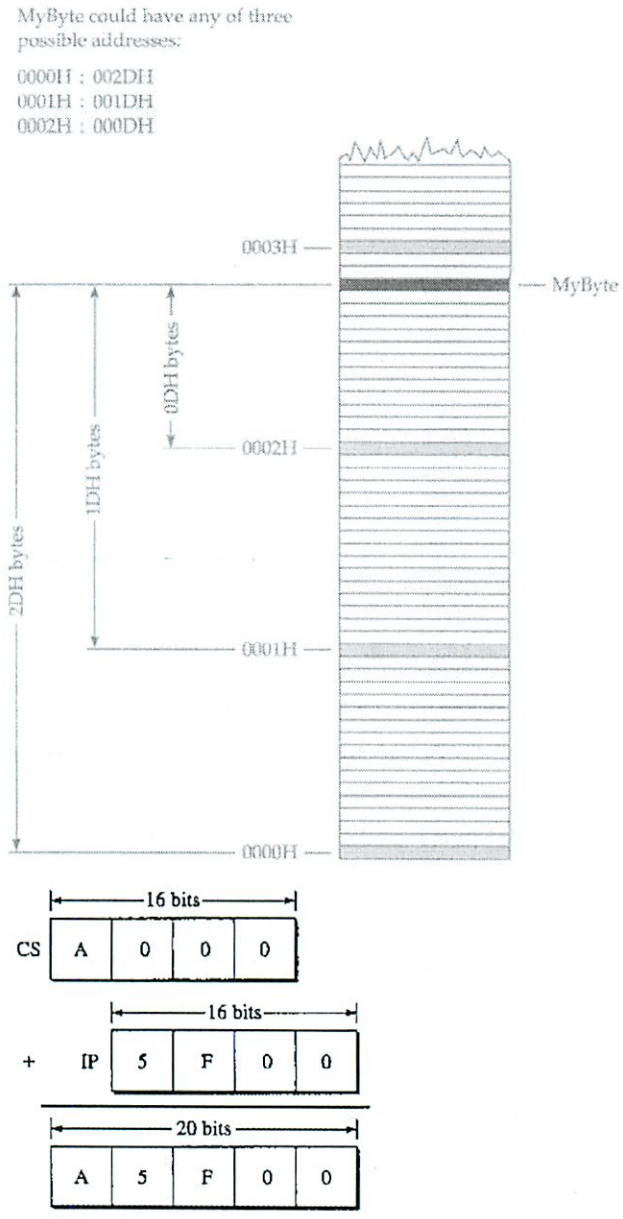
Initializing the data segment register:



10. Memory Access in Real Mode

- Recall that 8086 and 8088 CPUs had 20 address pins, limiting a program to 1 megabyte of memory.

- To express a 20-bit address, **two** 16-bit registers are used:
  - **segment address** in one 16-bit register,
  - and the **offset address** in another 16-bit register.
- The memory location of a particular byte from one megabyte of memory is calculated as
  - segment start *address*
  - **plus** *distance* between the byte and the segment start.
- The byte's distance from the start of the segment is referred to as the byte's *offset address*.
- `SEGMENT:OFFSET` addresses are always written in hexadecimal notation.
  - For example, an address of one byte of data **MyByte** is given as 0001:001D.
  - This means that **MyByte** is in segment 0001H and is located 001DH bytes from the start of that segment.
- Since segments can overlap, same byte could also be located by `SEGMENT:OFFSET` combinations 0002:000D or 0001:001D.



11. Segment Registers in Real Mode

- The 8088, 8086, and 80286 CPUs have **four** segment registers to hold segment addresses.
- The 386 and later CPUs have **two more**, also available in real mode.
- **Note**: the 386 and later Intel x86 CPUs still use 16-bit size segment registers.
- Each segment register is 16-bit in size.
- No matter how location in memory is accessed, the segment address of that location must be present in one of the six segment registers:

  CS, DS, SS, ES, FS and GS.

12. Segment Register Names

All x86 segment registers are 16 bits in size, irrespective of the CPU:

- CS, *code segment*. Machine instructions exist at some offset into a code segment. The segment address of the code segment of the currently executing instruction is contained in CS.

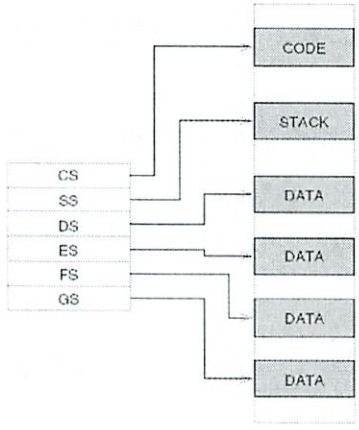
- DS, *data segment*. Variables and other data exist at some offset into a data segment. There may be many data segments, but the CPU may only use one at a time, by placing the segment address of that segment in register DS.
- SS, *stack segment*. The stack is a very important component of the CPU used for temporary storage of data and addresses. Therefore, *the stack has a segment address*, which is contained in register SS.
- ES, *extra segment*. The extra segment is exactly that: a spare segment that may be used for specifying a location in memory.

13. Segment Register Names, Cont.

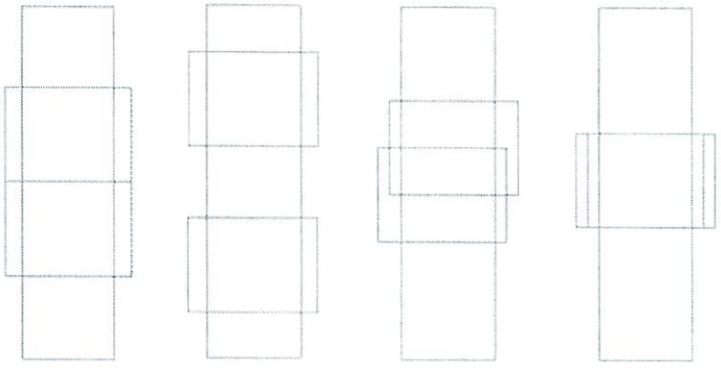
- FS and GS are clones of ES, the extra segment.
- FS and GS both are just additional segments, no specialty here.
- Names FS and GS come from the fact that they were created after ES: *E, F, G*.
- They exist only in the 386 and later x86 CPUs.
- Extra segments ES, FS, and GS can be used for both data or code.

We're supposed to use those names...

The six segments of the memory system:



14. Segment Positions in Real Mode

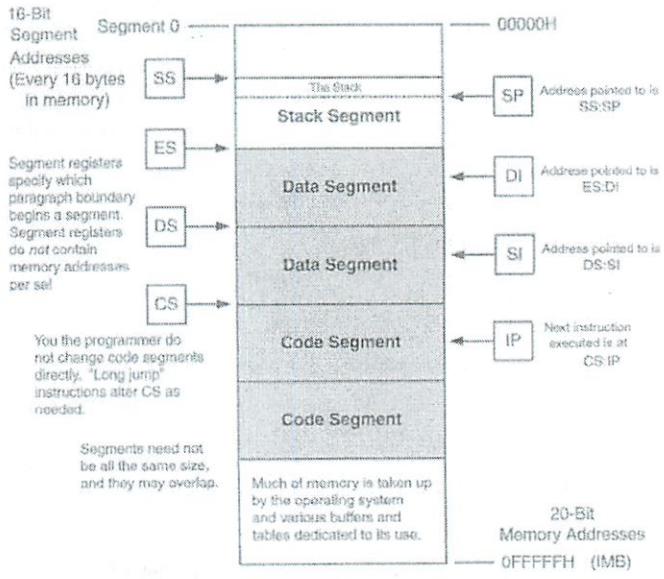


(a) Adjacent (b) Disjoint (c) Partially overlapped (d) Fully overlapped

15. General-Purpose Registers in Real mode

- General-purpose registers may hold
  - data values
  - offset addresses that must be paired with segment addresses to locate data in memory.
- The customary notation is to separate the segment register and the offset register by a colon. For example:

```
DS : AX
SS : SP
SS : BP
ES : DI
DS : SI
CS : BX
```



16. Segmentation Models Summary

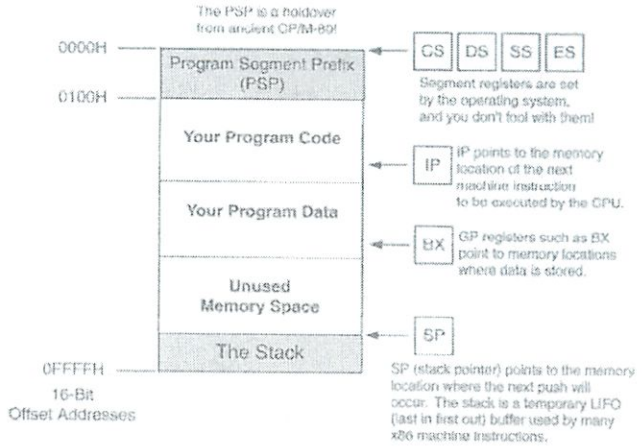
- Pentium can turn off segmentation.
- Flat model
  - Consists of one segment of 4GB
  - Used by linux
- Multisegment model
  - Up to six active segments
  - Can have more than six segments (all segment descriptors are in the descriptor table.)
  - A segment becomes active by loading its descriptor into one of the segment registers.

17. Real Mode Flat Model Summary

- CPU can see only 1 megabyte (1,048,576) of memory.
- **segment:offset** pairs form a 20-bit addresses out of **two** 16-bit addresses.
- In **real mode flat model** a program *and all its data* must exist within a single 64K block of memory.
- The real mode flat model is similar to *protected mode flat model*, the code model used on Linux and Windows XP/Vista.

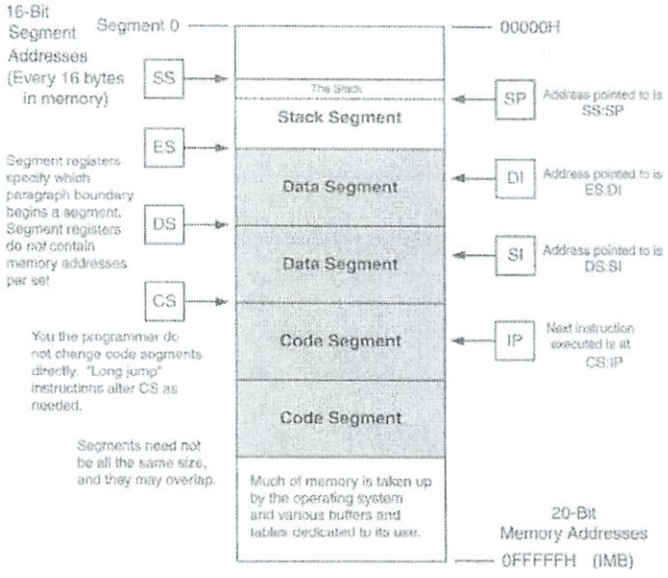
18. Real Mode Flat Model Diagram

- The segment registers are all set to point to the beginning of the 64K block of memory.
- The operating system sets segment registers when it loads the program.
- All segment registers point to that same place.
- Physical segment assignments **never change** as long as the program is running.
- The segment registers are still functioning, but no work with segments is required.



19. Real Mode Segmented Model

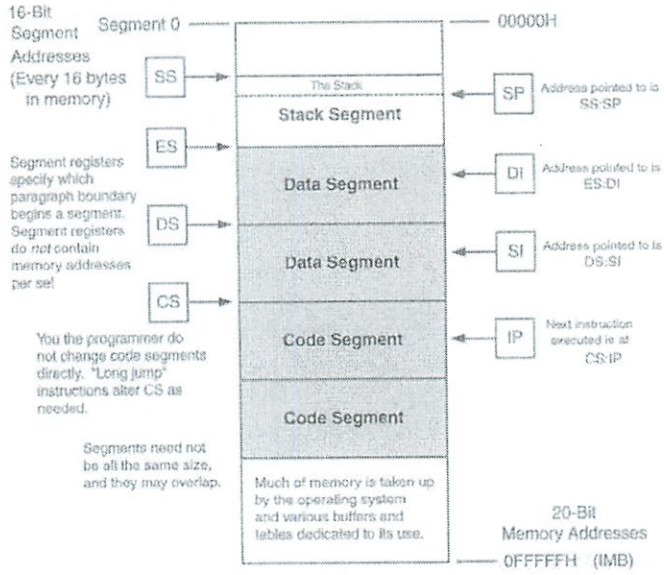
- **Real mode segmented model** was mainstream programming model throughout the MS-DOS era.
- Used when Windows 9x machine is booted into MS-DOS mode.
- Good choice to write code to run under MS-DOS.
- Program has access to 1MB of memory.
- The CPU handles transformations of **segment:offset** combinations into a full 20-bit address.
- CS always points to the current **code segment**



20. Real Mode Segmented Model, Cont.

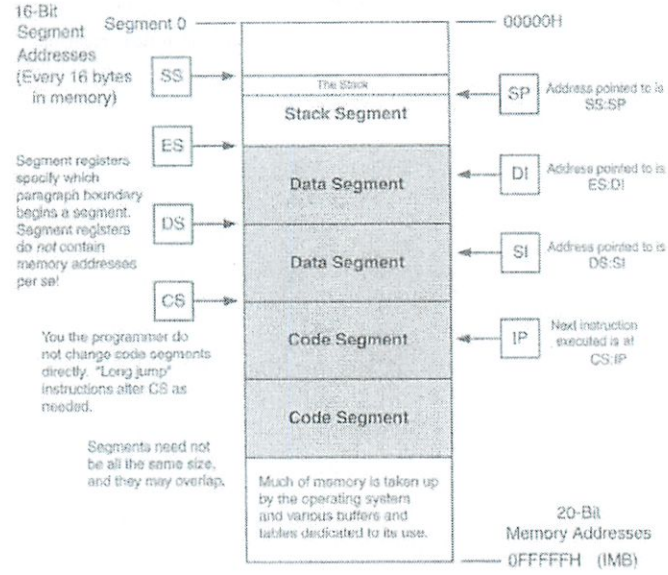
- The next instruction to be executed is pointed to by the CS:IP register pair.
- Machine instructions called **jumps** can change CS to another code segment if necessary.
- The program can span several code segments.
- There is no direct CS manipulation to change from one code segment to another:

  when a **jump** instruction needs to take execution into a different code segment, it changes CS value for you.



21. Real Mode Segmented Model, Cont.

- There is only **one** stack segment for any single program.
- A program has potential to *destroy portions of memory* that does not belong to its process.
- Careless use of segment registers will cause the operating system to crash.



22. x86 Protected Mode Flat Memory Model

- On 32-bit processors, Windows and Linux use the so-called *protected mode flat memory model*.
- Under flat memory model,
  - entire address space is described by a 32-bit segment, which provides $2^{32} = 4$ gigabytes of address space.
  - program can (in theory) access up to 4 gigabytes of virtual or physical memory.
- In protected mode,
  - segment registers contain *selector values* rather than actual physical segment addresses.
  - **Selector values** cannot be calculated by the program; they must be obtained by calling the operating system.
  - Programs that update segment values or attempt to address memory *directly* do not work in protected mode.

23. Advantages of Flat Memory Model

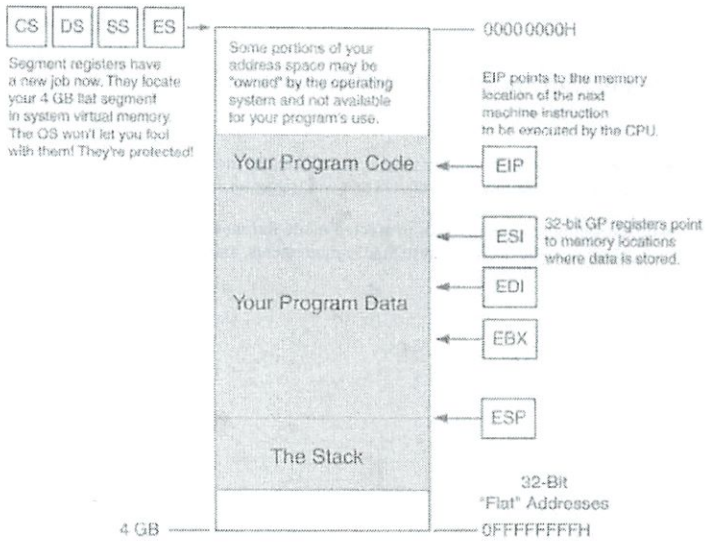
- 32-bit **Protected Mode** supports *much larger data structures* than **Real mode**.
- Because code, data, and stack reside in the *same segment*, each segment register can hold the same value that *never needs to change*.
- Rather than using a formula (such as CS:IP) to determine the physical address, protected mode processors use a *look up table*.
- Segment registers simply point to OS data structures that contain the information needed to access a location.
- **Protected mode** uses *privilege levels* to maintain system integrity and security.
- Programs cannot access data or code that is in a higher privilege level.

  (Segment value exchanges at the same privilege level are allowed.)

is this how most things work today?

24. The Protected Mode Flat Model Diagram

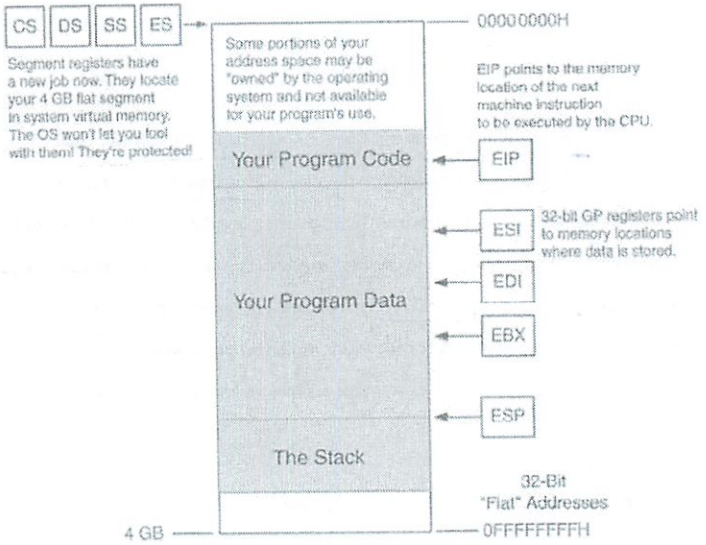
- The instruction pointer is 32 bits in size
- EIP can indicate any machine instruction anywhere in the 4 GB of memory.
- The segment registers still exists and define where 4 GB of program-accessible memory resides in *physical* or *virtual memory*
- The segment registers are now considered part of the operating system, you can neither read nor change them directly.



25. The Protected Mode Flat Model Diagram, Cont.

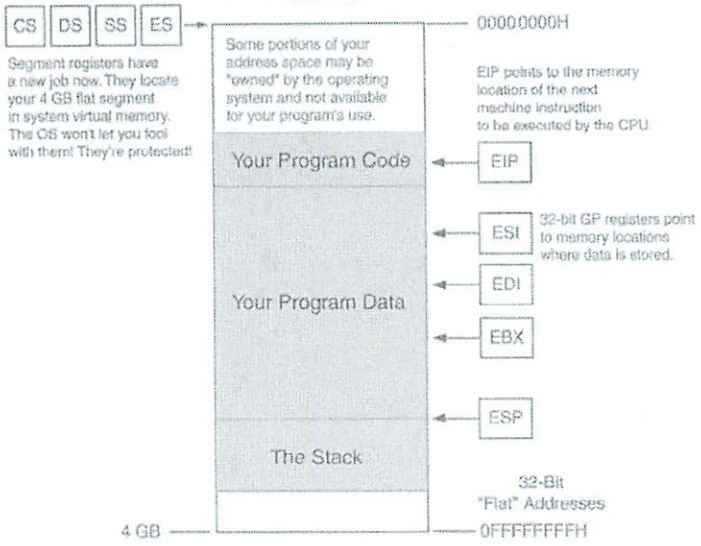
- When 32-bit program executes, it has access to 4-gig address space.
- Any general-purpose register by itself can specify any memory location in the entire memory address space of the 4 billion memory locations

  (except certain operating system-specific parts of the program that *belong to the operating system.*)



26. The Protected Mode Flat Model Diagram, Cont.

- Attempting to actually read or write certain locations in your own program can be forbidden by the OS and will trigger an error.
- Challenges in programming for **protected mode flat model** are based on understanding the operating system, its requirements, and restrictions.



27. Protected Mode Flat Model Summary

- Application programs cannot make use of protected mode by themselves.
- The operating system must set up and manage a protected mode.
- Capability provided by Linux and Windows NT/2000/XP/Vista systems.
- Each address is a 32-bit quantity.
- All of the general-purpose registers are 32 bits in size.

28. Console Applications

- An easiest way to write protected mode assembly program under Windows is to create *console application*.

- Console application is text-mode program that runs in a text-mode window called a *console*.
- The console is controlled by a user through a *command line interface*,

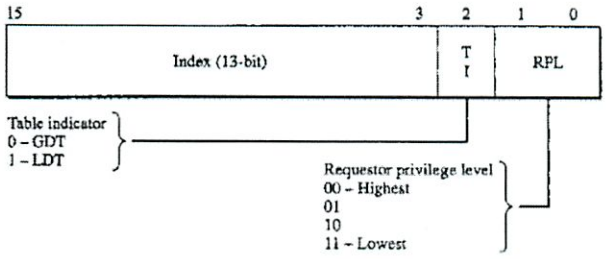
  (almost identical to the MS-DOS command window.)
- Console applications use **protected mode flat model** and are fairly straightforward.
- The default memory mode for text console app under Linux is also protected mode.

---

29. Segment Registers in Protected Mode

- Segment registers are called *selectors* when operating in **protected mode**.
- In protected mode, segment registers simply point to data structures called *segment descriptors* that contain the information needed to access a physical memory location.

Segment selector is a segment register, containing the *selector value:*



13-bit index field selects one of **8,192** *segment descriptors.*

TI (table indicator) specifies segment descriptor, which can be in

GDT - global descriptor table (one for all programs)

LDT - local descriptor table (typically one for each program, more can exist.)

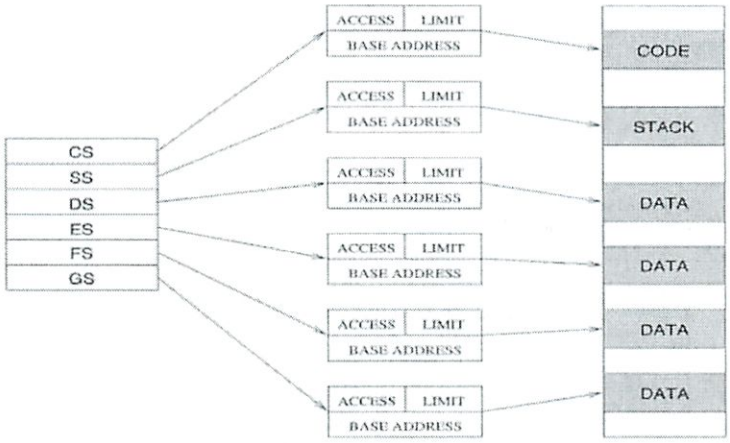
IDT - interrupt descriptor table.

RPL requestor privilege level - 2-bit field, specifies if the access to the segment is allowed.

---

30. Protected Mode Architecture

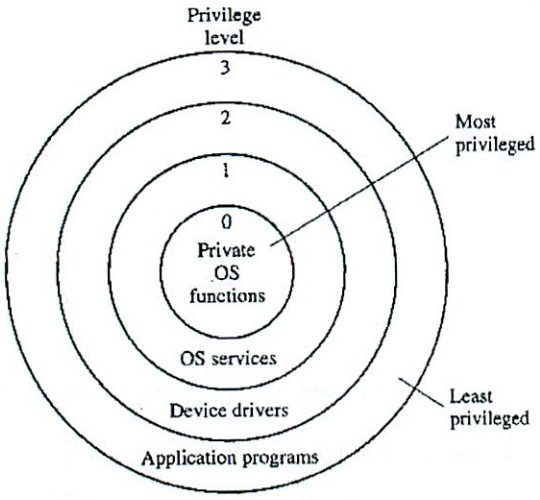
Segment register (segment selector) -> segment descriptor -> physical memory

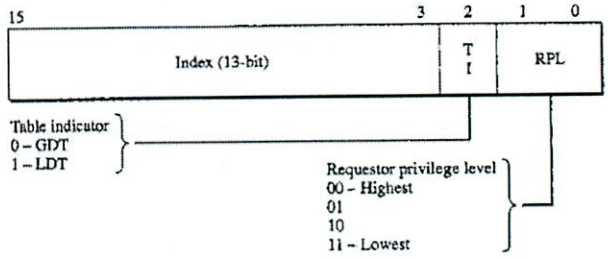


---

31. Rings of protection, four levels of security

The 2-bit requestor privilege level field, the RPL, specifies *segment protection level:*





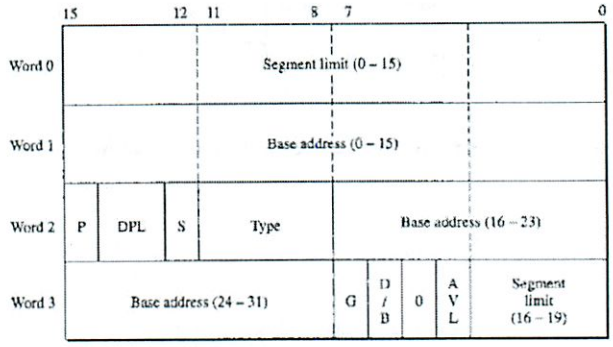
Segment register (segment selector)

Some instructions that directly access ports or affect interrupts (such as CLI, STI, IN, and OUT) are available at privilege levels normally used only by systems programs.

32. Three types of segment descriptor tables

1. Global descriptor table **GDT**
   - Only one in the system
   - Contains OS code and data
   - Available to all tasks
2. Local descriptor table **LDT**
   - Several LDTs may exist for a program.
   - Contains descriptors of a program

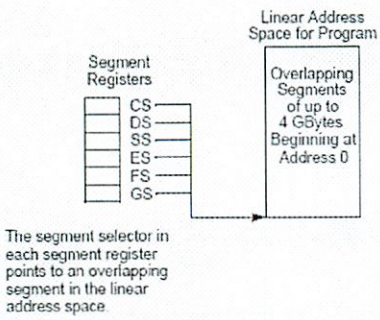
Segment descriptor format:



3. Interrupt descriptor table **IDT**, used by interrupt processing.

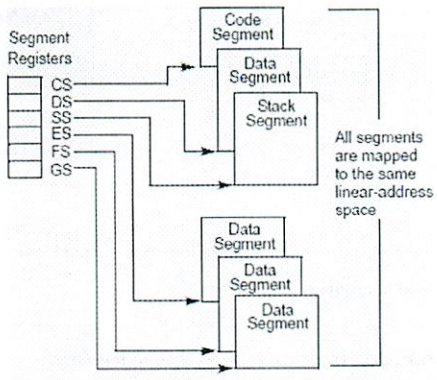
33. Flat vs. Segmented Memory Model

32 and 64-bit Intel systems using a flat memory model:



Use of Segment Registers for Flat Memory Model

Segmented memory is preferred model on 16-bit systems as it allows for flexibility in memory allocation and can also prevent buffer overruns:



Use of Segment Registers in Segmented Memory Model

34. Differences between 16-bit and 32-bit Memory Modes

- Understanding segments is an essential part of programming in assembly language.
- In the family of 8086-based processors, the term **segment** has two meanings:
  1. A block of memory of discrete size, called a *physical segment*. The number of bytes in a physical memory segment is

     (a) 64K for 16-bit processors

     (b) 4 gigabytes for 32-bit processors.
  2. A variable-sized block of memory, called a *logical segment* occupied by a program's code or data.

35. Differences, cont.

- Segmented architecture presents certain hurdles for 16-bit assembly-language program.
- For small 16-bit flat model programs, the limitations lose importance:
  - code and data each occupy less than 64K and reside in individual segments.
  - a simple offset locates each variable or instruction within a segment.

36. Differences, cont.

- Larger 16-bit programs, however, must contend with problems of segmented memory areas:
  - If data occupies two or more segments, the program must specify both segment and offset to access a variable.
  - When the data forms a continuous stream across segments, such as the text in a word processor's workspace, the problems become more acute.
  - Whenever it adds or deletes text in the first segment, the word processor must seamlessly move data back and forth over the boundaries of each following segment.
- The problem of segment boundaries disappears in flat address space of 32-bit protected mode:
  - Although segments still exist, they easily hold all the code and data of the largest programs.
  - Even a very large program becomes, in effect, a small application, capable to reach all code and all data with a *single offset address*.

37. Older OS and Addressing Modes Compared

The MS-DOS and Older Windows Operating Systems					
Operating System	**System Access**	**Available Active Processes**	**Addressable Memory**	**Contents of Segment Register**	**Word Length**
MS-DOS and Windows real mode	Direct to hardware and OS call	One	1 megabyte	Actual address	16 bits
Windows 3.x virtual-86 mode	Operating system call	Multiple	1 megabyte	Segment selectors	16 bits
Windows 3.x protected mode	Operating system call	Multiple	16 megabytes	Segment selectors	16 bits
Windows NT 3.x	Operating system call	Multiple	512 megabytes	Segment selectors	32 bits

10/15

```
Native Client
=============

What's the goal of this paper?
  At the time, browsers allowed any web page to run only JS (+Flash) code.
  Want to allow web apps to run native (e.g., x86) code on user's machine.
    Don't want to run complex code on server.
    Requires lots of server resources, incurs high latency for users.
  Why is this useful?
    Performance.
    Languages other than JS.
    Legacy apps.
  Actually being used in the real world.
    Ships as part of Google Chrome: the NaCl runtime is a browser extension.
    Web page can run a NaCl program much like a Flash program.
    Javascript can interact with the NaCl program by passing messages.
    NaCl also provides strong sandboxing for some other use cases.
  Core problem: sandboxing x86 code.

What are some options for safely running x86 code?

Approach 0: trust the code developer.
  ActiveX, browser plug-ins, Java, etc.
  Developer signs code with private key.
  Asks user to decide whether to trust code from some developer.
  Users are bad at making such decisions (e.g., with ActiveX code).
    Works for known developers (e.g., Windows Update code, signed by MS).
    Unclear how to answer for unknown web applications (other than "no").
  Native Client's goal is to enforce safety, avoid asking the user.

Approach 1: hardware protection / OS sandboxing.
  Similar plan to some ideas we've already read: OKWS, Capsicum, VMs, ..
  Run untrusted code as a regular user-space program or a separate VM.
  Need to control what system calls the untrusted code can invoke.
    Linux: seccomp.
    FreeBSD: Capsicum.
    MacOSX: Seatbelt.
    Windows: unclear what options exist.
  Native client uses these techniques, but only as a backup plan.
  Why not rely on OS sandboxing directly?
    Each OS may impose different, sometimes incompatible requirements.
      System calls to allocate memory, create threads, etc.
      Virtual memory layout (fixed-address shared libraries in Windows?).
    OS kernel vulnerabilities are reasonably common.
      Allows untrusted code to escape sandbox.
    Not every OS might have a sufficient sandboxing mechanism.
      E.g., unclear what to do on Windows, without a special kernel module.
      Some sandboxing mechanisms require root: don't want to run Chrome as root.
    Hardware might have vulnerabilities (!).
      Authors claim some instructions happen to hang the hardware.
      Would be unfortunate if visiting a web site could hang your computer.

Approach 2: software fault isolation (Native Client's primary sandboxing plan).
  In principle, similar to the rewriting idea from the "JS subsets" paper.
    Some high-level differences, though.
    "Rewriting" done by a modified compiler, not x86-to-x86 (hard).
    User's browser runs a verifier to check if "rewriting" was done correctly.
    Advantage: verifier much smaller than the compiler/rewriter -> small TCB.
  Overall plan:
    Given an x86 binary to run in Native Client, verify that it's safe.
    After verifying, can safely run it in same process as other trusted code.
    Allow the sandbox to call into trusted "service runtime" code.
```

```
    [ Figure 2 from paper. ]

What does safety mean for a Native Client module?
  Goal #1: does not execute any disallowed instructions (e.g., syscall, int).
    Ensures module does not perform any system calls.
  Goal #2: does not access memory or execute code outside of module boundary.
    Ensures module does not corrupt service runtime data structures.
    Ensures module does not jump into service runtime code, ala return-to-libc.
    As described in paper, module code+data live within [0..256MB) virt addrs.
      Need not populate entire 256MB of virtual address space.
    Everything else should be protected from access by the NaCl module.

How to check if the module can execute a disallowed instruction?
  Strawman: scan the executable, look for "int" or "syscall" opcodes.
    If check passes, can start running code.
    Of course, need to also mark all code as read-only.
    And all writable memory as non-executable.
  Complication: x86 has variable-length instructions.
    "int" and "syscall" instructions are 2 bytes long.
    Other instructions could be anywhere from 1 to 15 bytes.
  Suppose program's code contains the following bytes:
    25 CD 80 00 00
  If interpreted as an instruction starting from 25, it is a 5-byte instr:
    AND %eax, $0x000080cd
  But if interpreted starting from CD, it's a 2-byte instr:
    INT $0x80    # Linux syscall
  Could try looking for disallowed instructions at every offset..
    Likely will generate too many false alarms.
    Real instructions may accidentally have some "disallowed" bytes.

Reliable disassembly.
  Plan: ensure code only executes instructions verifier knows about.
  How can we guarantee this?  Table 1 and Figure 3 in paper.
  Scan forward through all instructions, starting at the beginning.
  If we see a jump instruction, make sure it's jumping to address we saw.
  Easy to ensure for static jumps (constant addr).
  Cannot ensure statically for computed jumps (jump to addr from register).
  Similar to the problem in FBJS with variable-based array indexing.

Computed jumps.
  Much as with FBJS, idea is to rely on runtime instrumentation.
  For computed jump to %eax, NaCl requires the following code:
    AND $0xffffffe0, %eax
    JMP *%eax
  This will ensure jumps go to multiples of 32 bytes.
  NaCl also requires that no instructions span a 32-byte boundary.
  Compiler's job is to ensure both of these rules.
    Replace every computed jump with the two-instruction sequence above.
    Add NOP instructions if some other instruction might span 32-byte boundary.
    Add NOPs to pad to 32-byte multiple if next instr is a computed jump target.
    Always possible because NOP instruction is just one byte.
  Verifier's job is to check these rules.
    During disassembly, make sure no instruction spans a 32-byte boundary.
    For computed jumps, ensure it's in a two-instruction sequence as above.
  What will this guarantee?
    Verifier checked all instructions starting at 32-byte-multiple addresses.
    Computed jumps can only go to 32-byte-multiple addresses.
  How does NaCl deal with RET instructions?
    Prohibited -- effectively a computed jump, with address stored on stack.
    Instead, compiler must generate explicit POP + computed jump code.

Why are the rules from Table 1 necessary?
  C1: executable code in memory is not writable.
```

```
  C2: binary is statically linked at zero, code starts at 64K.
  C3: all computed jumps use the two-instruction sequence above.
  C4: binary is padded to a page boundary with one or more HLT instruction.
  C5: no instructions, or our special two-instruction pair, can span 32 bytes.
  C6/C7: all jump targets reachable by fall-through disassembly from start.

Homework Q: what happens if verifier gets some instruction length wrong?

How to prevent NaCl module from jumping to 32-byte multiple outside its code?
  Could use additional checks in the computed-jump sequence.
  E.g.:
    AND $0x0fffffe0, %eax
    JMP *%eax
  Why don't they use this approach?
    Longer instruction sequence for computed jumps.
    Their sequence is 3+2=5 bytes, above sequence is 5+2=7 bytes.
    An alternative solution is pretty easy: segmentation.

Segmentation.
  x86 hardware provides "segments".
  Each memory access is with respect to some "segment".
    Segment specifies base + size.
  Segments are specified by a segment selector: ptr into a segment table.
    %cs, %ds, %ss, %es, %fs, %gs
    Each instruction can specify what segment to use for accessing memory.
    Code always fetched using the %cs segment.
  Translation: (segment selector, addr) -> (segbase + addr % segsize).
  Typically, all segments have base=0, size=max, so segmentation is a no-op.
  Can change segments: in Linux, modify_ldt() system call.
  Can change segment selectors: just "MOV %ds", etc.

Limiting code/data to module's size.
  Add a new segment with offset=0, size=256MB.
  Set all segment selectors to that segment.
  Modify verifier to reject any instructions that change segment selectors.
  Ensures all code and data accesses will be within [0..256MB).

What would be required to run Native Client on a system without segmentation?
  For example, AMD/Intel decided to drop segment limits in their 64-bit CPUs.
  One practical possibility: run in 32-bit mode.
    AMD/Intel CPUs still support segment limits in 32-bit mode.
    Can run in 32-bit mode even on a 64-bit OS.
  Would have to change the computed-jump code to limit target to 256MB.
  Would have to add runtime instrumentation to each memory read/write.
  See the paper in additional references below for more details.

Why doesn't Native Client support exceptions for modules?
  What if module triggers hardware exception: null ptr, divide-by-zero, etc.
  OS kernel needs to deliver exception (as a signal) to process.
  But Native Client runs with an unusual stack pointer/segment selector.
  Some OS kernels refuse to deliver signals in this situation.
  NaCl's solution is to prohibit hardware exceptions altogether.
  Language-level exceptions (e.g., C++) do not involve hardware: no problem.

What would happen if the NaCl module had a buffer overflow?
  Any computed call (function pointer, return address) has to use 2-instr jump.
  As a result, can only jump to validated code in the module's region.
  Buffer overflows might allow attacker to take over module.
  However, can't escape NaCl's sandbox.

Limitations of the original NaCl design?
  Static code: no JIT, no shared libraries.
  Dynamic code supported in recent versions (see additional refs at the end).
```

```
Invoking trusted code from sandbox.
  Short code sequences that transition to/from sandbox located in [4KB..64KB).
  Trampoline undoes the sandbox, enters trusted code.
    Starts at a 32-byte multiple boundary.
    Loads unlimited segment into %cs, %ds segment selectors.
    Jumps to trusted code that lives above 256MB.
    Slightly tricky: must ensure trampoline fits in 32 bytes.
    (Otherwise, module could jump into middle of trampoline code..)
    Trusted code first switches to a different stack: why?
  Springboard (re-)enters the sandbox on return or initial start.
    Re-set segment selectors, jump to a particular address in NaCl module.
    Springboard slots (32-byte multiples) start with HLT.
    Prevents computed jumps into springboard by module code.

What's provided by the service runtime?  NaCl's "system call" equivalent.
  Memory allocation: sbrk/mmap.
  Thread operations: create, etc.
  IPC: initially with Javascript code on page that started this NaCl program.
  Browser interface via NPAPI: DOM access, open URLs, user input, ..
  No networking: can use Javascript to access network according to SOP.

How secure is Native Client?
  List of attack surfaces: start of section 2.3.
  Inner sandbox: validator has to be correct (had some tricky bugs!).
  Outer sandbox: OS-dependent plan.
  Service runtime: initial loader, runtime trampoline interfaces.
  IMC interface + NPAPI: complex code, can (and did) have bugs.

How well does it perform?
  CPU overhead seems to be dominated by NaCl's code alignment requirements.
    Larger instruction cache footprint.
    But for some applications, NaCl's alignment works better than gcc's.
  Minimal overhead for added checks on computed jumps.
  Call-into-service-runtime performance seems comparable to Linux syscalls.

How hard is it to port code to NaCl?
  For computational things, seems straightforward: 20 LoC change for H.264.
  For code that interacts with system (syscalls, etc), need to change them.
    E.g., Bullet physics simulator (section 4.4).

Additional references.
  Native Client for 64-bit x86 and for ARM.
    http://static.usenix.org/events/sec10/tech/full_papers/Sehr.pdf
  Native Client for runtime-generated code (JIT).
    http://research.google.com/pubs/archive/37204.pdf
  Native Client without hardware dependence.
    http://css.csail.mit.edu/6.858/2012/readings/pnacl.pdf
  Other software fault isolation systems w/ fine-grained memory access control.
    http://css.csail.mit.edu/6.858/2012/readings/xfi.pdf
    http://research.microsoft.com/pubs/101332/bgi-sosp.pdf
```

Read 10/17

The Quest to Replace Passwords: A Framework for Comparative Evaluation of Web Authentication Schemes*

Joseph Bonneau
University of Cambridge
Cambridge, UK
jcb82@cl.cam.ac.uk

Cormac Herley
Microsoft Research
Redmond, WA, USA
cormac@microsoft.com

Paul C. van Oorschot
Carleton University
Ottawa, ON, Canada
paulv@scs.carleton.ca

Frank Stajano†
University of Cambridge
Cambridge, UK
frank.stajano@cl.cam.ac.uk

***Abstract*—We evaluate two decades of proposals to replace text passwords for general-purpose user authentication on the web using a broad set of twenty-five usability, deployability and security benefits that an ideal scheme might provide. The scope of proposals we survey is also extensive, including password management software, federated login protocols, graphical password schemes, cognitive authentication schemes, one-time passwords, hardware tokens, phone-aided schemes and biometrics. Our comprehensive approach leads to key insights about the difficulty of replacing passwords. Not only does no known scheme come close to providing all desired benefits: none even retains the full set of benefits that legacy passwords already provide. In particular, there is a wide range from schemes offering minor security benefits beyond legacy passwords, to those offering significant security benefits in return for being more costly to deploy or more difficult to use. We conclude that many academic proposals have failed to gain traction because researchers rarely consider a sufficiently wide range of real-world constraints. Beyond our analysis of current schemes, our framework provides an evaluation methodology and benchmark for future web authentication proposals.**

***Keywords*-authentication; computer security; human computer interaction; security and usability; deployability; economics; software engineering.**

I. Introduction

The continued domination of passwords over all other methods of end-user authentication is a major embarrassment to security researchers. As web technology moves ahead by leaps and bounds in other areas, passwords stubbornly survive and reproduce with every new web site. Extensive discussions of alternative authentication schemes have produced no definitive answers.

Over forty years of research have demonstrated that passwords are plagued by security problems [2] and openly hated by users [3]. We believe that, to make progress, the community must better systematize the knowledge that we have regarding both passwords and their alternatives [4]. However, among other challenges, unbiased evaluation of password replacement schemes is complicated by the diverse interests of various communities. In our experience, security experts focus more on security but less on usability and practical issues related to deployment; biometrics experts focus on analysis of false negatives and naturally-occurring false positives rather than on attacks by an intelligent, adaptive adversary; usability experts tend to be optimistic about security; and originators of a scheme, whatever their background, downplay or ignore benefits that their scheme doesn't attempt to provide, thus overlooking dimensions on which it fares poorly. As proponents assert the superiority of their schemes, their objective functions are often not explicitly stated and differ substantially from those of potential adopters. Targeting different authentication problems using different criteria, some address very specific environments and narrow scenarios; others silently seek generic solutions that fit all environments at once, assuming a single choice is mandatory. As such, consensus is unlikely.

These and other factors have contributed to a long-standing lack of progress on how best to evaluate and compare authentication proposals intended for practical use. In response, we propose a standard benchmark and framework allowing schemes to be rated across a common, broad spectrum of criteria chosen objectively for relevance in wide-ranging scenarios, without hidden agenda.[1] We suggest and define 25 properties framed as a diverse set of benefits, and a methodology for comparative evaluation, demonstrated and tested by rating 35 password-replacement schemes on the same criteria, as summarized in a carefully constructed comparative table.

Both the rating criteria and their definitions were iteratively refined over the evaluation of these schemes. Discussion of evaluation details for passwords and nine representative alternatives is provided herein to demonstrate the process, and to provide evidence that the list of benefits suffices to illuminate the strengths and weaknesses of a wide universe of schemes. Though not cast in stone, we believe that the list of benefits and their specific definitions provide an excellent basis from which to work; the framework and

*An extended version of this paper is available as a University of Cambridge technical report [1].

†Frank Stajano was the lead author who conceived the project and assembled the team. All authors contributed equally thereafter.

[1]The present authors contributed to the definition of the following schemes: URRSA [5], MP-Auth [6], PCCP [7] and Pico [8]. We invite readers to verify that we have rated them impartially.

In *Proc. IEEE Symp. on Security and Privacy 2012* ("Oakland 2012").

evaluation process that we define are independent of them, although our comparative results naturally are not. From our analysis and comparative summary table, we look for clues to help explain why passwords remain so dominant, despite frequent claims of superior alternatives.

In the past decade our community has recognized a tension between security and usability: it is generally easy to provide more of one by offering less of the other. But the situation is much more complex than simply a linear trade-off: we seek to capture the multi-faceted, rather than one-dimensional, nature of both usability and security in our benefits. We further suggest that "deployability", for lack of a better word, is an important third dimension that deserves consideration. We choose to examine all three explicitly, complementing earlier comparative surveys (e.g., [9]–[11]).

Our usability-deployability-security ("UDS") evaluation framework and process may be referred to as *semi-structured evaluation of user authentication schemes*. We take inspiration from inspection methods for evaluating user interface design, including *feature inspections* and Nielsen's *heuristic analysis* based on usability principles [12].

Each co-author acted as a domain expert, familiar with both the rating framework and a subset of the schemes. For each scheme rated, the evaluation process involved one co-author studying the scheme and rating it on the defined benefits; additional co-authors reviewing each rating score; and iteratively refining the ratings as necessary through discussion, as noted in Section V-D.

Our focus is *user authentication on the web*, specifically from unsupervised end-user client devices (e.g., a personal computer) to remote verifiers. Some schemes examined involve mobile phones as auxiliary devices, but logging in directly from such constrained devices, which involves different usability challenges among other things, is not a main focus. Our present work does not directly examine schemes designed exclusively for machine-to-machine authentication, e.g., cryptographic protocols or infrastructure such as client public-key certificates. Many of the schemes we examine, however, are the technologies proposed for the human-to-machine component that may precede machine-to-machine authentication. Our choice of web authentication as target application also has significant implications for specific schemes, as noted in our results.

II. Benefits

The benefits we consider encompass three categories: usability, deployability and security, the latter including privacy aspects. The benefits in our list have been refined to a set we believe highlights important evaluation dimensions, with an eye to limiting overlap between benefits.

Throughout the paper, for brevity and consistency, each benefit is referred to with an italicized mnemonic title. This title should not be interpreted too literally; refer instead to our actual definitions below, which are informally worded to aid use. Each scheme is rated as either offering or not offering the benefit; if a scheme *almost* offers the benefit, but not quite, we indicate this with the *Quasi-* prefix. Section V-D discusses pros and cons of finer-grained scoring.

Sometimes a particular benefit (e.g., *Resilient-to-Theft*) just doesn't apply to a particular scheme (e.g., there is nothing physical to steal in a scheme where the user must memorize a secret squiggle). To simplify analysis, instead of introducing a "not applicable" value, we rate the scheme as offering the benefit—in the sense that nothing can go wrong, for that scheme, with respect to the corresponding problem.

When rating password-related schemes we assume that implementers use best practice such as salting and hashing (even though we know they often don't [13]), because we assess what the scheme's design can potentially offer: a poor implementation could otherwise kill any scheme. On the other hand, we assume that ordinary users won't necessarily follow the often unreasonably inconvenient directives of security engineers, such as never recycling passwords, or using randomly-generated ones.

A. Usability benefits

- U1 *Memorywise-Effortless*: Users of the scheme do not have to remember *any* secrets at all. We grant a *Quasi-Memorywise-Effortless* if users have to remember *one* secret for everything (as opposed to one per verifier).
- U2 *Scalable-for-Users*: Using the scheme for hundreds of accounts does not increase the burden on the user. As the mnemonic suggests, we mean "scalable" only from the user's perspective, looking at the cognitive load, not from a system deployment perspective, looking at allocation of technical resources.
- U3 *Nothing-to-Carry*: Users do not need to carry an additional physical object (electronic device, mechanical key, piece of paper) to use the scheme. *Quasi-Nothing-to-Carry* is awarded if the object is one that they'd carry everywhere all the time anyway, such as their mobile phone, but not if it's their computer (including tablets).
- U4 *Physically-Effortless*: The authentication process does not require physical (as opposed to cognitive) user effort beyond, say, pressing a button. Schemes that don't offer this benefit include those that require typing, scribbling or performing a set of motions. We grant *Quasi-Physically-Effortless* if the user's effort is limited to speaking, on the basis that even illiterate people find that natural to do.
- U5 *Easy-to-Learn*: Users who don't know the scheme can figure it out and learn it without too much trouble, and then easily recall how to use it.
- U6 *Efficient-to-Use*: The time the user must spend for each authentication is acceptably short. The time

learn for our paper on tests for $5085

required for setting up a new association with a verifier, although possibly longer than that for authentication, is also reasonable.

U7 *Infrequent-Errors*: The task that users must perform to log in usually succeeds when performed by a legitimate and honest user. In other words, the scheme isn't so hard to use or unreliable that genuine users are routinely rejected.[2]

U8 *Easy-Recovery-from-Loss*: A user can conveniently regain the ability to authenticate if the token is lost or the credentials forgotten. This combines usability aspects such as: low latency before restored ability; low user inconvenience in recovery (e.g., no requirement for physically standing in line); and assurance that recovery will be possible, for example via built-in backups or secondary recovery schemes. If recovery requires some form of re-enrollment, this benefit rates its convenience.

B. Deployability benefits

D1 *Accessible*: Users who can use passwords[3] are not prevented from using the scheme by disabilities or other physical (not cognitive) conditions.

D2 *Negligible-Cost-per-User*: The total cost per user of the scheme, adding up the costs at both the prover's end (any devices required) and the verifier's end (any share of the equipment and software required), is negligible. The scheme is plausible for startups with no per-user revenue.

D3 *Server-Compatible*: At the verifier's end, the scheme is compatible with text-based passwords. Providers don't have to change their existing authentication setup to support the scheme.

D4 *Browser-Compatible*: Users don't have to change their client to support the scheme and can expect the scheme to work when using other machines with an up-to-date, standards-compliant web browser and no additional software. In 2012, this would mean an HTML5-compliant browser with JavaScript enabled. Schemes fail to provide this benefit if they require the installation of plugins or any kind of software whose installation requires administrative rights. Schemes offer *Quasi-Browser-Compatible* if they rely on non-standard but very common plugins, e.g., Flash.

D5 *Mature*: The scheme has been implemented and deployed on a large scale for actual authentication purposes beyond research. Indicators to consider for granting the full benefit may also include whether the scheme has undergone user testing, whether the standards community has published related documents, whether open-source projects implementing the scheme exist, whether anyone other than the implementers has adopted the scheme, the amount of literature on the scheme and so forth.

D6 *Non-Proprietary*: Anyone can implement or use the scheme for any purpose without having to pay royalties to anyone else. The relevant techniques are generally known, published openly and not protected by patents or trade secrets.

C. Security benefits

S1 *Resilient-to-Physical-Observation*: An attacker cannot impersonate a user after observing them authenticate one or more times. We grant *Quasi-Resilient-to-Physical-Observation* if the scheme could be broken only by repeating the observation more than, say, 10–20 times. Attacks include shoulder surfing, filming the keyboard, recording keystroke sounds, or thermal imaging of keypad.

S2 *Resilient-to-Targeted-Impersonation*: It is not possible for an acquaintance (or skilled investigator) to impersonate a specific user by exploiting knowledge of personal details (birth date, names of relatives etc.). Personal knowledge questions are the canonical scheme that fails on this point.

S3 *Resilient-to-Throttled-Guessing*: An attacker whose rate of guessing is constrained by the verifier cannot successfully guess the secrets of a significant fraction of users. The verifier-imposed constraint might be enforced by an online server, a tamper-resistant chip or any other mechanism capable of throttling repeated requests. To give a quantitative example, we might grant this benefit if an attacker constrained to, say, 10 guesses per account per day, could compromise at most 1% of accounts in a year. Lack of this benefit is meant to penalize schemes in which it is frequent for user-chosen secrets to be selected from a small and well-known subset (low min-entropy [14]).

S4 *Resilient-to-Unthrottled-Guessing*: An attacker whose rate of guessing is constrained only by available computing resources cannot successfully guess the secrets of a significant fraction of users. We might for example grant this benefit if an attacker capable of attempting up to 2^{40} or even 2^{64} guesses per account could still only reach

blacklist

[2]We could view this benefit as "low false reject rate". In many cases the scheme designer could make the false reject rate lower by making the false accept rate higher. If this is taken to an extreme we count it as cheating, and penalize it through a low score in some of the security-related benefits.

[3]Ideally a scheme would be usable by everyone, regardless of disabilities like zero-vision (blindness) or low motor control. However, for any given scheme, it is always possible to identify a disability or physical condition that would exclude a category of people and then no scheme would be granted this benefit. We therefore choose to award the benefit to schemes that do at least as well as the incumbent that is de facto accepted today, despite the fact that it too isn't perfect. An alternative to this text password baseline could be to base the metric on the ability to serve a defined percentage of the population of potential users.

fewer than 1% of accounts. Lack of this benefit is meant to penalize schemes where the space of credentials is not large enough to withstand brute force search (including dictionary attacks, rainbow tables and related brute force methods smarter than raw exhaustive search, if credentials are user-chosen secrets).

S5 *Resilient-to-Internal-Observation*: An attacker cannot impersonate a user by intercepting the user's input from inside the user's device (e.g., by key-logging malware) or eavesdropping on the cleartext communication between prover and verifier (we assume that the attacker can also defeat TLS if it is used, perhaps through the CA). As with *Resilient-to-Physical-Observation* above, we grant *Quasi-Resilient-to-Internal-Observation* if the scheme could be broken only by intercepting input or eavesdropping cleartext more than, say, 10–20 times. This penalizes schemes that are not replay-resistant, whether because they send a static response or because their dynamic response countermeasure can be cracked with a few observations. This benefit assumes that general-purpose devices like software-updatable personal computers and mobile phones may contain malware, but that hardware devices dedicated exclusively to the scheme can be made malware-free. We grant *Quasi-Resilient-to-Internal-Observation* to two-factor schemes where both factors must be malware-infected for the attack to work. If infecting only one factor breaks the scheme, we don't grant the benefit.

S6 *Resilient-to-Leaks-from-Other-Verifiers*: Nothing that a verifier could possibly leak can help an attacker impersonate the user to another verifier. This penalizes schemes where insider fraud at one provider, or a successful attack on one back-end, endangers the user's accounts at other sites.

S7 *Resilient-to-Phishing*: An attacker who simulates a valid verifier (including by DNS manipulation) cannot collect credentials that can later be used to impersonate the user to the actual verifier. This penalizes schemes allowing phishers to get victims to authenticate to lookalike sites and later use the harvested credentials against the genuine sites. It is not meant to penalize schemes vulnerable to more sophisticated real-time man-in-the-middle or relay attacks, in which the attackers have one connection to the victim prover (pretending to be the verifier) and simultaneously another connection to the victim verifier (pretending to be the prover).

S8 *Resilient-to-Theft*: If the scheme uses a physical object for authentication, the object cannot be used for authentication by another person who gains possession of it. We still grant *Quasi-Resilient-to-Theft* if the protection is achieved with the modest strength of a PIN, even if attempts are not rate-controlled, because the attack doesn't easily scale to many victims.

S9 *No-Trusted-Third-Party*: The scheme does not rely on a trusted third party (other than the prover and the verifier) who could, upon being attacked or otherwise becoming untrustworthy, compromise the prover's security or privacy.

S10 *Requiring-Explicit-Consent*: The authentication process cannot be started without the explicit consent of the user. This is both a security and a privacy feature (a rogue wireless RFID-based credit card reader embedded in a sofa might charge a card without user knowledge or consent).

S11 *Unlinkable*: Colluding verifiers cannot determine, from the authenticator alone, whether the same user is authenticating to both. This is a privacy feature. To rate this benefit we disregard linkability introduced by other mechanisms (same user ID, same IP address, etc).

We emphasize that it would be simple-minded to rank competing schemes simply by counting how many benefits each offers. Clearly some benefits deserve more weight than others—but which ones? *Scalable-for-Users*, for example, is a heavy-weight benefit if the goal is to adopt a single scheme as a universal replacement; it is less important if one is seeking a password alternative for only a single account. Providing appropriate weights thus depends strongly on the specific goal for which the schemes are being compared, which is one of the reasons we don't offer any.

Having said that, readers wanting to use weights might use our framework as follows. First, examine and score each individual scheme on each benefit; next, compare (groups of) competing schemes to identify precisely which benefits each offers over the other; finally, with weights that take into account the relative importance of the benefits, determine an overall ranking by rating scheme i as $S_i = \sum_j W_j \cdot b_{i,j}$. Weights W_j are constants across all schemes in a particular comparison exercise, and $b_{i,j} \in [0,1]$ is the real-valued benefit rating for scheme i on benefit j. For different solution environments (scenarios k), the relative importance of benefits will differ, with ~~weights W_j~~ replaced by $W_j^{(k)}$.

In this paper we choose a more qualitative approach: we do not suggest any weights $W_j^{(k)}$ and the $b_{i,j}$ ratings we assign are not continuous but coarsely quantized. In Section V-D we discuss why. In our experience, *"the journey* (the rating exercise) *is the reward"*: the important technical insights we gained about schemes by discussing whether our ratings were fair and consistent were worth much more to us than the actual scores produced. As a take-home message for the value of this exercise, bringing a team of experts to

a shared understanding of the relevant technical issues is much more valuable than ranking the schemes linearly or reaching unanimous agreement over scoring.

III. Evaluating Legacy Passwords

We expect that the reader is familiar with text passwords and their shortcomings, so evaluating them is good exercise for our framework. It's also useful to have a baseline standard to refer to. While we consider "legacy passwords" as a single scheme, surveys of password deployment on the web have found substantial variation in implemention. A study of 150 sites in 2010 [13], for example, found a unique set of design choices at nearly every site. Other studies have focused on implementations of cookie semantics [15], password composition policies [16], or use of TLS to protect passwords [17]. Every study has found both considerable inconsistency and frequent serious implementation errors in practical deployments on the web.

We remind readers of our Section II assumption of best practice by implementers—thus in our ratings we do not hold against passwords the many weak implementations that their widespread deployment includes, unless due to inherent weaknesses; while on the other hand, our ratings of passwords and other schemes do assume that poor user behavior is an inherent aspect of fielded systems.

The difficulty of guessing passwords was studied over three decades ago [2] with researchers able to guess over 75% of users' passwords; follow-up studies over the years have consistently compromised a substantial fraction of accounts with dictionary attacks. A survey [3] of corporate password users found them flustered by password requirements and coping by writing passwords down on post-it notes. On the web, users are typically overwhelmed by the number of passwords they have registered. One study [18] found most users have many accounts for which they've forgotten their passwords and even accounts they can't remember registering. Another [19] used a browser extension to observe thousands of users' password habits, finding on average 25 accounts and 6 unique passwords per user.

Thus, passwords, as a purely memory-based scheme, clearly aren't *Memorywise-Effortless* or *Scalable-for-Users* as they must be remembered and chosen for each site. While they are *Nothing-to-Carry*, they aren't *Physically-Effortless* as they must be typed. Usability is otherwise good, as passwords are de facto *Easy-to-Learn* due to years of user experience and *Efficient-to-Use* as most users type only a few characters, though typos downgrade passwords to *Quasi-Infrequent-Errors*. Passwords can be easily reset, giving them *Easy-Recovery-from-Loss*.

Their highest scores are in deployability, where they receive full credit for every benefit—in part because many of our criteria are defined based on passwords. For example, passwords are *Accessible* because we defined the benefit with respect to them and accommodations already exist for most groups due to the importance of passwords. Passwords are *Negligible-Cost-per-User* due to their simplicity, and are *Server-Compatible* and *Browser-Compatible* due to their incumbent status. Passwords are *Mature* and *Non-Proprietary*, with turnkey packages implementing password authentication for many popular web development platforms, albeit not well-standardized despite their ubiquity.

Passwords score relatively poorly on security. They aren't *Resilient-to-Physical-Observation* because even if typed quickly they can be automatically recovered from high-quality video of the keyboard [20]. Perhaps generously, we rate passwords as *Quasi-Resilient-to-Targeted-Impersonation* in the absence of user studies establishing acquaintances' ability to guess passwords, though many users undermine this by keeping passwords written down in plain sight [3]. Similarly, users' well-established poor track record in selection means passwords are neither *Resilient-to-Throttled-Guessing* nor *Resilient-to-Unthrottled-Guessing*.

As static tokens, passwords aren't *Resilient-to-Internal-Observation*. The fact that users reuse them across sites means they also aren't *Resilient-to-Leaks-from-Other-Verifiers*, as even a properly salted and strengthened hash function [21] can't protect many passwords from dedicated cracking software. (Up to 50% of websites don't appear to hash passwords at all [13].) Passwords aren't *Resilient-to-Phishing* as phishing remains an open problem in practice.

Finally, their simplicity facilitates several security benefits. They are *Resilient-to-Theft* as they require no hardware. There is *No-Trusted-Third-Party*; having to type makes them *Requiring-Explicit-Consent*; and, assuming that sites add salt independently, even weak passwords are *Unlinkable*.

IV. Sample Evaluation of Replacement Schemes

We now use our criteria to evaluate a representative sample of proposed password replacement schemes. Table I visually summarizes these and others we explored. Due to space constraints, we only explain in detail our ratings for at most one representative scheme per category (e.g. federated login schemes, graphical passwords, hardware tokens, etc.). Evaluation details for all other schemes in the table are provided in a companion technical report [1].

We introduce categories to highlight general trends, but stress that any scheme must be rated individually. Contrary to what the table layout suggests, schemes are not uniquely partitioned by the categories; several schemes belong to multiple categories, and different groupings of the schemes are possible with these same categories. For example, GrIDsure is both cognitive and graphical; and, though several of the schemes we examine use some form of underlying "one-time-passwords", we did not group them into a common category and indeed have no formal category of that name.

We emphasize that, in selecting a particular scheme for inclusion in the table or for discussion as a category representative, we do not necessarily endorse it as better than

alternatives—merely that it is reasonably representative, or illuminates in some way what the category can achieve.

A. Encrypted password managers: Mozilla Firefox

The Firefox web browser [22] automatically offers to remember passwords entered into web pages, optionally encrypting them with a master password. (Our rating assumes that this option is used; use without the password has different properties.) It then pre-fills the username and password fields when the user revisits the same site. With its Sync facility the passwords can be stored, encrypted, in the cloud. After a once-per-machine authentication ritual, they are updated automatically on all designated machines.

This scheme is *Quasi-Memorywise-Effortless* (because of the master password) and *Scalable-for-Users*: it can remember arbitrarily many passwords. Without Sync, the solution would have required carrying a specific computer; with Sync, the passwords can be accessed from any of the user's computers. However it's not more than *Quasi-Nothing-to-Carry* because a travelling user will have to carry at least a smartphone: it would be quite insecure to sync one's passwords with a browser found in a cybercafé. It is *Quasi-Physically-Effortless*, as no typing is required during authentication except for the master password once per session, and *Easy-to-Learn*. It is *Efficient-to-Use* (much more so than what it replaces) and has *Infrequent-Errors* (hardly any, except when entering the master password). It does not have *Easy-Recovery-from-Loss*: losing the master password is catastrophic.

The scheme is backwards-compatible by design and thus scores quite highly on deployability: it fully provides all the deployability benefits except for *Browser-Compatible*, unavoidably because it requires a specific browser.

It is *Quasi-Resilient-to-Physical-Observation* and *Quasi-Resilient-to-Targeted-Impersonation* because an attacker could still target the infrequently-typed master password (but would also need access to the browser). It is not *Resilient-to-Throttled-Guessing* nor *Resilient-to-Unthrottled-Guessing*: even if the master password is safe from such attacks, the original web passwords remain as vulnerable as before.[4] It is not *Resilient-to-Internal-Observation* because, even if TLS is used, it's replayable static passwords that flow in the tunnel and malware could also capture the master password. It's not *Resilient-to-Leaks-from-Other-Verifiers*, because what happens at the back-end is the same as with passwords. It's *Resilient-to-Phishing* because we assume that sites follow best practice, which includes using TLS for the login page. It is *Resilient-to-Theft*, at least under our assumption that a master password is being used. It offers *No-Trusted-Third-Party* because the Sync data is pre-encrypted locally before being stored on Mozilla's servers. It offers *Requiring-Explicit-Consent* because it pre-fills the username and password fields but the user still has to press enter to submit. Finally, it is as *Unlinkable* as passwords.

[4]Security-conscious users might adopt truly random unguessable passwords, as they need no longer remember them, but most users won't. If the scheme pre-generated random passwords it would score more highly here, disregarding pre-existing passwords. Similarly, for *Resilient-to-Leaks-from-Other-Verifiers* below, this scheme makes it easier for careful users to use a different password for every site; if it forced this behaviour (vs. just allowing it), it would get a higher score on this particular benefit.

B. Proxy-based: URRSA

Proxy-based schemes place a man-in-the-middle between the user's machine and the server. One reason for doing so, employed by Impostor [23] and URRSA [5] is to enable secure logins despite malware-infected clients.

URRSA has users authenticate to the end server using one-time codes carried on a sheet of paper. At registration the user enters the password, P_j, for each account, j, to be visited; this is encrypted at the proxy with thirty different keys, K_i, giving $C_i = E_{K_i}(P_j)$. The C_i act as one-time codes which the user prints and carries. The codes are generally 8-10 characters long; thirty codes for each of six accounts fit on a two-sided sheet. The keys, but not the passwords, are stored at the proxy. At login the user visits the proxy, indicates which site is desired, and is asked for the next unused code. When he enters the code it is decrypted and passed to the end login server: $E_{K_i}^{-1}(C_i) = P_j$. The proxy never authenticates the user, it merely decrypts with an agreed-upon key, the code delivered by the user.

Since it requires carrying one-time codes URRSA is *Memorywise-Effortless*, but not *Scalable-for-Users* or *Nothing-to-Carry*. It is not *Physically-Effortless* but is *Easy-to-Learn*. In common with all of the schemes that involve transcribing codes from a device or sheet it is not *Efficient-to-Use*. However, we do consider it to have *Quasi-Infrequent-Errors*, since the codes are generally 8-10 characters. It does not have *Easy-Recovery-from-Loss*: a revocation procedure is required if the code sheet is lost or stolen. Since no passwords are stored at the proxy the entire registration must be repeated if this happens.

In common with other paper token schemes it is not *Accessible*. URRSA has *Negligible-Cost-per-User*. Rather than have a user change browser settings, URRSA relies on a link-translating proxy that intermediates traffic between the user and the server; this translation is not flawless and some functionality may fail on complex sites, thus we consider it only *Quasi-Server-Compatible*. It is, however, *Browser-Compatible*. It is neither *Mature* nor *Non-Proprietary*.

In common with other one-time code schemes it is not *Resilient-to-Physical-Observation*, since a camera might capture all of the codes on the sheet. Since it merely inserts a proxy it inherits many security weaknesses from the legacy password system it serves: it is *Quasi-Resilient-to-Targeted-Impersonation* and is not *Resilient-to-Throttled-Guessing* or *Resilient-to-Unthrottled-Guessing*. It is *Quasi-Resilient-to-Internal-Observation* as observing the client during authentication does not allow passwords to be captured, but breaking

the proxy-to-server TLS connection does. It inherits from passwords the fact that it is not *Resilient-to-Leaks-from-Other-Verifiers*, but the fact that it is *Resilient-to-Phishing* from other one-time schemes. It is not *Resilient-to-Theft* nor *No-Trusted-Third-Party*: the proxy must be trusted. It offers *Requiring-Explicit-Consent* and is *Unlinkable*.

C. Federated Single Sign-On: OpenID

Federated single sign-on enables web sites to authenticate a user by redirecting them to a trusted identity server which attests the users' identity. This has been considered a "holy grail" as it could eliminate the problem of remembering different passwords for different sites. The concept of federated authentication dates at least to the 1978 Needham-Schroeder key agreement protocol [24] which formed the basis for Kerberos [25]. Kerberos has inspired dozens of proposals for federated authentication on the Internet; Pashalidis and Mitchell provided a complete survey [26]. A well-known representative is OpenID,[5] a protocol which allows any web server to act as an "identity provider" [27] to any server desiring authentication (a "relying party"). OpenID has an enthusiastic group of followers both in and out of academia, but it has seen only patchy adoption with many sites willing to act as identity providers but few willing to accept it as relying parties [28].

In evaluating OpenID, we note that in practice identity providers will continue to use text passwords to authenticate users in the forseeable future, although the protocol itself allows passwords to be replaced by a stronger mechanism. Thus, we rate the scheme *Quasi-Memorywise-Effortless* in that most users will still have to remember one master password, but *Scalable-for-Users* as this password can work for multiple sites. OpenID is *Nothing-to-Carry* like passwords and *Quasi-Physically-Effortless* because passwords only need to be typed at the identity provider. Similarly, we rate it *Efficient-to-Use* and *Infrequent-Errors* in that it is either a password authentication or can occur automatically in a browser with cached login cookies for the identity provider. However, OpenID has found that selecting an opaque "identity URL" can be a significant usability challenge without a good interface at the relying party, making the scheme only *Quasi-Easy-to-Learn*. OpenID is *Easy-Recovery-from-Loss*, equivalent to a password reset.

OpenID is favorable from a deployment standpoint, providing all benefits except for *Server-Compatible*, including *Mature* as it has detailed standards and many open-source implementations. We do note however that it requires identity providers yield some control over trust decisions and possibly weaken their own brand [28], a deployment drawback not currently captured in our criteria.

[5]OpenID is often confused with OAuth, a technically unrelated protocol for delegating access to one's accounts to third parties. The recent OpenID Connect proposal merges the two. We consider the OpenID 2.0 standard here, though all current versions score identically in our framework.

Security-wise, OpenID reduces most attacks to only the password authentication between a user and his or her identity provider. This makes it somewhat difficult to rate; we consider it *Quasi-Resilient-to-Throttled-Guessing*, *Quasi-Resilient-to-Unthrottled-Guessing*, *Quasi-Resilient-to-Targeted-Impersonation*, *Quasi-Resilient-to-Physical-Observation* as these attacks are possible but only against the single identity provider (typically cached in a cookie) and not for each login to all verifiers. However, it is not *Resilient-to-Internal-Observation* as malware can either steal persistent login cookies or record the master password. OpenID is also believed to be badly non-*Resilient-to-Phishing* since it involves re-direction to an identity provider from a relying party [29]. OpenID is *Resilient-to-Leaks-from-Other-Verifiers*, as relying parties don't store users passwords. Federated schemes have been criticized on privacy grounds and, while OpenID does enable technically savvy users to operate their own identity provider, we rate OpenID as non-*Unlinkable* and non-*No-Trusted-Third-Party* as the vast majority of users aren't capable of doing so.

D. Graphical passwords: Persuasive Cued Clickpoints (PCCP)

Graphical passwords schemes attempt to leverage natural human ability to remember images, which is believed to exceed memory for text. We consider as a representative PCCP [7] (Persuasive Cued Click-Points), a cued-recall scheme. Users are sequentially presented with five images on each of which they select one point, determining the next image displayed. To log in, all selected points must be correctly re-entered within a defined tolerance. To flatten the password distribution, during password creation a randomly-positioned portal covers a portion of each image; users must select their point from therein (the rest of each image is shaded slightly). Users may hit a "shuffle" button to randomly reposition the portal to a different region—but doing so consumes time, thus persuading otherwise. The portal is absent on regular login. Published security analysis and testing report reasonable usability and improved security over earlier schemes, specifically in terms of resistance to both *hotspots* and *pattern-based* attacks [11].

While not *Memorywise-Effortless*, nor *Scalable-for-Users* due to extra cognitive load for each account password, PCCP offers advantages over text passwords (and other uncued schemes) due to per-account image cues reducing password interference. It is *Easy-to-Learn* (usage and mental models match web passwords, but interface details differ), but only *Quasi-Efficient-to-Use* (login times on the order of 5s to 20s exceed text passwords) and at best *Quasi-Infrequent-Errors*.

PCCP is not *Accessible* (consider blind users) and has *Negligible-Cost-per-User*. It is not *Server-Compatible*; though it might be made so by having a proxy act as intermediary (much as URRSA does). It is *Browser-Compatible*. It is not *Mature*, but apparently *Non-Proprietary*.

PCCP is not *Resilient-to-Physical-Observation* (due to video-camera shoulder surfing), but is *Resilient-to-Targeted-Impersonation* (personal knowledge of a target user does not help attacks). We rate it *Quasi-Resilient-to-Throttled-Guessing* due to portal persuasion increasing password randomness, but note individual users may repeatedly bypass portal recommendations. Although the persuasion is also intended to mitigate offline attacks, we rate it not *Resilient-to-Unthrottled-Guessing* as studies to date have been limited to full password spaces of 2^{43} (which are within reach of offline dictionary attack, especially for users choosing more predictable passwords, assuming verifier-stored hashes are available). It is not *Resilient-to-Internal-Observation* (static passwords are replayable). It is *Resilient-to-Leaks-from-Other-Verifiers* (distinct sites can insist on distinct image sets). PCCP is *Resilient-to-Phishing* per our strict definition of that benefit; to obtain the proper per-user images, a phishing site must interact (e.g., by MITM) with a legitimate server. PCCP matches text passwords on being *Unlinkable*.

E. Cognitive authentication: GrIDsure

Challenge-Response schemes attempt to address the replay attack on passwords by having the user deliver proof that he knows the secret without divulging the secret itself. If memorization and computation were no barrier then the server might challenge the user to return a cryptographic hash of the user's secret combined with a server-selected nonce. However, it is unclear if a scheme within the means of human memory and calculating ability is achievable. We examine the commercial offering GrIDsure (a variant of which is described in a paper [30] by other authors) as representative of the class.

At registration the user is presented with a grid (e.g., 5×5) and selects a pattern, or sequence of cells. There are 25^4 possible length-4 patterns, for example. At login the user is again presented with the grid, but now populated with digits. To authenticate he transcribes the digits in the cells corresponding to his pattern. Since the association of digits to cells is randomized the string typed by the user is different from login to login. Thus he reveals knowledge of his secret without typing the secret itself.

This scheme is similar to passwords in terms of usability and we (perhaps generously) rate it identically in terms of many usability benefits. An exception is that it's only *Quasi-Efficient-to-Use*: unlike passwords, which can often be typed from muscle memory, transcribing digits from the grid cells requires effort and attention and is likely to be slower.

We consider the scheme as not *Accessible* as the two-dimensional layout seems unusable for blind users. The scheme has *Negligible-Cost-per-User*, in terms of technology. It is not *Server-Compatible* but is *Browser-Compatible*. It is not *Mature*. We rate it not *Non-Proprietary*, as the intellectual property status is unknown.

The security properties are, again, similar to passwords in many respects. It is not *Resilient-to-Physical-Observation*, as a camera that captures both the grid and user input quickly learns the secret. It is an improvement on passwords in that it is *Resilient-to-Targeted-Impersonation*: we assume that an attacker is more likely to guess secret strings than secret patterns based on knowledge of the user. However, its small space of choices prevents it from being *Resilient-to-Throttled-Guessing* or *Resilient-to-Unthrottled-Guessing*. In spite of the one-time nature of what the user types the scheme is not *Resilient-to-Internal-Observation*: too many possible patterns are eliminated at each login for the secret to withstand more than three or four observations. It shares the remaining security benefits with passwords.

F. Paper tokens: OTPW

Using paper to store long secrets is the cheapest form of a physical login token. The concept is related to military codebooks used throughout history, but interest in using possession of paper tokens to authenticate humans was spurred in the early 1980's by Lamport's hash-chaining scheme [31], later developed into S/KEY [32]. OTPW is a later refinement, developed by Kuhn in 1998 [33], in which the server stores a larger set of independent hash values, consisting of about 4 kB per user. The user carries the hash pre-images, printed as 8-character values like `IZdB bqvH`. Logging in requires typing a "prefix password" as well as one randomly-queried hash-preimage.

OTPW rates poorly for usability: the prefix password means the scheme isn't *Memorywise-Effortless* or *Scalable-for-Users*; it also isn't *Nothing-to-Carry* because of the paper token. The typing of random passwords means the scheme also isn't *Physically-Effortless*, *Efficient-to-Use* or *Infrequent-Errors*. We do expect that the scheme is *Easy-to-Learn*, as typing in a numbered password upon request is only marginally more difficult than using text passwords. It is also *Easy-Recovery-from-Loss* as we expect most users can easily print a new sheet if needed.

Paper-based tokens are cheap and easy to deploy. We rate OTPW as non-*Accessible* because plain printing may be insufficient for visually-impaired users, though alternatives (e.g. braille) may be available. We consider the price of printing to be *Negligible-Cost-per-User*. While not *Server-Compatible*, the scheme is *Browser-Compatible*. Finally, OTPW has a mature open-source implementation, making it *Mature* and *Non-Proprietary*.

Though OTPW is designed to resist human observation compared to S/KEY, it isn't *Resilient-to-Physical-Observation* because the printed sheet of one-time codes can be completely captured by a camera. Otherwise, OTPW achieves all other security benefits. Because login codes are used only once and randomly generated, the scheme is *Resilient-to-Throttled-Guessing*, *Resilient-to-Unthrottled-Guessing* and *Resilient-to-Internal-Observation*.

scheme. It's *Resilient-to-Theft* because possession of the phone is insufficient: the user still needs to type user ID and password in the browser (for additional protection against theft, the authors envisage an additional PIN or biometric to authenticate the user to the device; we are not rating this). The scheme is *No-Trusted-Third-Party* if we disregard the CA that certifies the TLS certificate of the bank. It's *Requiring-Explicit-Consent* because the user must type user ID and password. Finally it's *Unlinkable* because the phone has a different key pair for each verifier.

I. Biometrics: Fingerprint recognition

Biometrics [37] are the "what you are" means of authentication, leveraging the uniqueness of physical or behavioral characteristics across individuals. We discuss in detail *fingerprint* biometrics [38]; our summary table also rates *iris recognition* [39] and voiceprint biometrics [40]. In rating for our remote authentication application, and *biometric verification* ("Is this individual asserted to be Jane Doe really Jane Doe?"), we assume unsupervised biometric hardware as might be built into client devices, vs. verifier-provided hardware, e.g., at an airport supervised by officials.

Fingerprint biometrics offer usability advantages *Memorywise-Effortless*, *Scalable-for-Users*, *Easy-to-Learn*, and *Nothing-to-Carry* (no secrets need be carried; we charge elsewhere for client-side fingerprint readers not being currently universal). Current products are at best *Quasi-Physically-Effortless* and *Quasi-Efficient-to-Use* due to user experience of not *Infrequent-Errors* (the latter two worse than web passwords) and fail to offer *Easy-Recovery-from-Loss* (here equated with requiring an alternate scheme in case of compromise, or users becoming unable to provide the biometric for physical reasons).

Deployability is poor—we rate it at best *Quasi-Accessible* due to common failure-to-register biometric issues; not *Negligible-Cost-per-User* (fingerprint reader has a cost); neither *Server-Compatible* nor *Browser-Compatible*, needing both client and server changes; at best *Quasi-Mature* for unsupervised remote authentication; and not *Non-Proprietary*, typically involving proprietary hardware and/or software.

We rate the fingerprint biometric *Resilient-to-Physical-Observation* but serious concerns include easily fooling COTS devices, e.g., by lifting fingerprints from glass surfaces with gelatin-like substances [41], which we charge by rating not *Resilient-to-Targeted-Impersonation*. It is *Resilient-to-Throttled-Guessing*, but not *Resilient-to-Unthrottled-Guessing* for typical precisions used; estimated "effective equivalent key spaces" [9, page 2032] for fingerprint, iris and voice are 13.3 bits, 19.9 bits and 11.7 bits respectively. It is not *Resilient-to-Internal-Observation* (captured samples of static physical biometrics are subject to replay in unsupervised environments), not *Resilient-to-Leaks-from-Other-Verifiers*, not *Resilient-to-Phishing* (a serious concern as biometrics are by design supposed to be hard to change), and not *Resilient-to-Theft* (see above re: targeted impersonation). As a plus, it needs *No-Trusted-Third-Party* and is *Requiring-Explicit-Consent*. Physical biometrics are also a canonical example of schemes that are not *Unlinkable*.

V. Discussion

A clear result of our exercise is that no scheme we examined is perfect—or even comes close to perfect scores. The incumbent (traditional passwords) achieves all benefits on deployability, and one scheme (the CAP reader, discussed in the tech report [1]) achieves all in security, but no scheme achieves all usability benefits. Not a single scheme is dominant over passwords, i.e., does better on one or more benefits and does at least as well on all others. Almost all schemes do better than passwords in some criteria, but all are worse in others: as Table I shows, no row is free of red (horizontal) stripes.

Thus, the current state of the world is a Pareto equilibrium. Replacing passwords with any of the schemes examined is not a question of giving up an inferior technology for something unarguably better, but of giving up one set of compromises and trade-offs in exchange for another. For example, arguing that a hardware token like RSA SecurID is better than passwords implicitly assumes that the security criteria where it does better outweigh the usability and deployability criteria where it does worse. For accounts that require high assurance, security benefits may indeed outweigh the fact that the scheme doesn't offer *Nothing-to-Carry* nor *Negligible-Cost-per-User*, but this argument is less compelling for lower value accounts.

The usability benefits where passwords excel—namely, *Nothing-to-Carry*, *Efficient-to-Use*, *Easy-Recovery-from-Loss*—are where essentially all of the stronger security schemes need improvement. None of the paper token or hardware token schemes achieves even two of these three. In expressing frustration with the continuing dominance of passwords, many security experts presumably view these two classes of schemes to be sufficiently usable to justify a switch from passwords. The web sites that crave user traffic apparently disagree.

Some sets of benefits appear almost incompatible, e.g., the pair (*Memorywise-Effortless*, *Nothing-to-Carry*) is achieved only by biometric schemes. No schemes studied achieve (*Memorywise-Effortless*, *Resilient-to-Theft*) fully, nor (*Server-Compatible*, *Resilient-to-Internal-Observation*) or (*Server-Compatible*, *Resilient-to-Leaks-from-Other-Verifiers*), though several almost do. Note that since compatibility with existing servers almost assures a static replayable secret, to avoid its security implications, many proposals abandon being *Server-Compatible*.

A. Rating categories of schemes

Password managers offer advantages over legacy passwords in selected usability and security aspects without

It is *Resilient-to-Phishing* as it is impractical for a user to enter all of their secrets into a phishing website even if asked, and *Resilient-to-Theft* thanks to the prefix password. As a one-to-one scheme with different secrets for each server, it is *Resilient-to-Leaks-from-Other-Verifiers*, *No-Trusted-Third-Party* and *Unlinkable*. Finally, the typing required makes it *Requiring-Explicit-Consent*.

G. Hardware tokens: RSA SecurID

Hardware tokens store secrets in a dedicated tamper-resistant module carried by the user; the RSA SecurID [34] family of tokens is the long-established market leader. Here we refer to the simplest dedicated-hardware version, which has only a display and no buttons or I/O ports. Each instance of the device holds a secret "seed" known to the back-end. A cryptographically strong transform generates a new 6-digit code from this secret every 60 seconds. The current code is shown on the device's display. On enrollment, the user connects to the administrative back-end through a web interface, where he selects a PIN and where the pairing between username and token is confirmed. From then on, for authenticating, instead of username and password the user shall type username and "passcode" (concatenation of a static 4-digit PIN and the dynamic 6-digit code). RSA offers an SSO facility to grant access to several corporate resources with the same token; but we rate this scheme assuming there won't be a single SSO spanning all verifiers.

In March 2011 attackers compromised RSA's back-end database of seeds [35], which allowed them to predict the codes issued by any token. This reduced the security of each account to that of its PIN until the corresponding token was recalled and reissued.

The scheme is not *Memorywise-Effortless* nor *Scalable-for-Users* (it needs a new token and PIN per verifier). It's not *Physically-Effortless*, because the user must transcribe the passcode. It's simple enough to be *Easy-to-Learn*, but *Quasi-Efficient-to-Use* because of the transcription. We rate it as having *Quasi-Infrequent-Errors*, like passwords, though it might be slightly worse. It is not *Easy-Recovery-from-Loss*: the token must be revoked and a new one reissued.

The scheme is not *Accessible*: blind users cannot read the code off the token. No token-based scheme can offer *Negligible-Cost-per-User*. The scheme is not *Server-Compatible* (a new back-end is required) but it is *Browser-Compatible*. It is definitely *Mature*, but not *Non-Proprietary*.

As for security, because the code changes every minute, SecurID is *Resilient-to-Physical-Observation*, *Resilient-to-Targeted-Impersonation*, *Resilient-to-Throttled-Guessing* and *Resilient-to-Unthrottled-Guessing* (unless we also assume that the attacker broke into the server and stole the seeds). It is *Resilient-to-Internal-Observation*: we assume that dedicated devices can resist malware infiltration. It's *Resilient-to-Leaks-from-Other-Verifiers*, as different verifiers would have their own seeds; *Resilient-to-Phishing*, because captured passcodes expire after one minute; and *Resilient-to-Theft*, because the PIN is checked at the verifier, so guesses could be rate-limited. It's not *No-Trusted-Third-Party*, as demonstrated by the March 2011 attack, since RSA keeps the seed of each token. It's *Requiring-Explicit-Consent*, as the user must transcribe the passcode, and *Unlinkable* if each verifier requires its own token.

H. Mobile-Phone-based: Phoolproof

Phoolproof Phishing Prevention [36] is another token-based design, but one in which the token is a mobile phone with special code and crypto keys. It uses public key cryptography and an SSL-like authentication protocol and was designed to be as compatible as possible with existing systems.

Phoolproof was conceived as a system to secure banking transactions against phishing, not as a password replacement. The user selects a desired site from the whitelist on the phone; the phone talks wirelessly to the browser, causing the site to be visited; an end-to-end TLS-based mutual authentication ensues between the phone and the bank's site; the user must still type the banking website password into the browser. Thus the scheme is not *Memorywise-Effortless*, nor *Scalable-for-Users*. It has *Quasi-Nothing-to-Carry* (the mobile phone). It's not *Physically-Effortless* as one must type a password. We rate it *Easy-to-Learn*, perhaps generously, and *Quasi-Efficient-to-Use* as it requires both typing a password and fiddling with a phone. It's no better than passwords on *Quasi-Infrequent-Errors*, since it still uses one. The only recovery mechanism is revocation and reissue, so it doesn't have *Easy-Recovery-from-Loss*.

On deployability: it's *Quasi-Accessible* insofar as most disabled users, including blind people, can use a mobile phone too (note the user doesn't need to transcribe codes from the phone). We assume most users will already have a phone, though perhaps not one of the right type (with Java, Bluetooth etc), hence it has *Quasi-Negligible-Cost-per-User*. The scheme requires changes, albeit minor, to both ends, so it's *Quasi-Server-Compatible* but, by our definitions, not *Browser-Compatible* because it uses a browser plugin. It's not really *Mature* (only a research prototype), but it is *Non-Proprietary*.

On security: it's *Resilient-to-Physical-Observation*, *Resilient-to-Targeted-Impersonation*, *Resilient-to-Throttled-Guessing*, *Resilient-to-Unthrottled-Guessing* because, even after observing or guessing the correct password, the attacker can't authenticate unless he also steals the user's phone, which holds the cryptographic keys. It's *Quasi-Resilient-to-Internal-Observation* because malware must compromise both the phone (to capture the private keys) and the computer (to keylog the password). It's *Resilient-to-Leaks-from-Other-Verifiers* because the phone has a key pair per verifier, so credentials are not recycled. It's definitely *Resilient-to-Phishing*, the main design requirement of the

Category	Scheme	Described in section	Reference	Usability								Deployability						Security											
				Memorywise-Effortless	*Scalable-for-Users*	*Nothing-to-Carry*	*Physically-Effortless*	*Easy-to-Learn*	*Efficient-to-Use*	*Infrequent-Errors*	*Easy-Recovery-from-Loss*	*Accessible*	*Negligible-Cost-per-User*	*Server-Compatible*	*Browser-Compatible*	*Mature*	*Non-Proprietary*	*Resilient-to-Physical-Observation*	*Resilient-to-Targeted-Impersonation*	*Resilient-to-Throttled-Guessing*	*Resilient-to-Unthrottled-Guessing*	*Resilient-to-Internal-Observation*	*Resilient-to-Leaks-from-Other-Verifiers*	*Resilient-to-Phishing*	*Resilient-to-Theft*	*No-Trusted-Third-Party*	*Requiring-Explicit-Consent*	*Unlinkable*
(Incumbent)	Web passwords	III	[13]			●		●	●	○	●	●	●	●	●	●	●		○						●	●	●	●	●
Password managers	Firefox	IV-A	[22]	○	●	○	○	●	●	●	▒	●	●	●	▒	●	●	○	○					●	●	●	▒	●	●
	LastPass		[42]	○	●	○	○	●	●	●	○	●	○	●	▒	●	▒	○	○	○	○		○	●	●	●	▒	●	●
Proxy	URRSA	IV-B	[5]	●		▒		●	▒	○	▒	▒	●	○	●	▒	▒		○			○		●	▒	▒	▒	●	●
	Impostor		[23]	○	●	●		●	▒	▒	●	●	●	●	○	▒	▒	●	○			○		●	▒	▒	▒	●	●
Federated	OpenID	IV-C	[27]	○	●	●	○	○	●	●	●	●	●	▒	▒	●	●	○	○	○	○		●		●	▒	▒	●	●
	Microsoft Passport		[43]	○	●	●	○	●	●	●	●	●	▒	▒	▒	●	▒	○	○	○	○		●		●	▒	▒	●	●
	Facebook Connect		[44]	○	●	●	○	●	●	●	●	●	▒	▒	▒	●	▒	○	○	○	○		●		●	▒	▒	●	●
	BrowserID		[45]	○	●	●	○	●	●	●	●	●	▒	▒	○	○	●	○	○	○	○		●		●	▒	▒	●	●
	OTP over email		[46]	○	●	●		●	▒	●	●	●	▒	▒	▒	▒	▒	○	○	○	○		●	●	●	▒	▒	▒	▒
Graphical	PCCP	IV-D	[7]			●		●	○	○	●	▒	●	▒	●	▒	●		●	○			●	●	●	●	●	●	●
	PassGo		[47]			●		●	○	○	●	▒	●	▒	●	○	●		●						●	●	●	●	●
Cognitive	GrIDsure (original)	IV-E	[30]			●		●	○	○	●	▒	●	▒	●	▒	▒		●				●	●	●	●	●	●	●
	Weinshall		[48]			●		▒	▒	▒	▒	▒	●	▒	●	▒	●	○	●				●	●	●	●	●	●	●
	Hopper Blum		[49]			●		▒	▒	▒	▒	●	●	▒	●	▒	▒	○	●				●	●	●	●	●	●	●
	Word Association		[50]			●		●	●	○	○	●	●	▒	●	▒	●		▒						●	●	●	●	●
Paper tokens	OTPW	IV-F	[33]			▒		●	▒	▒	●	▒	●	▒	●	●	●		●	●	●	●	●	●	●	●	●	●	●
	S/KEY		[32]	●		▒		●	▒	○	●	▒	●	▒	●	●	●		●	●	●	●	●	○	▒	●	●	●	●
	PIN+TAN		[51]			▒		●	▒	○	○	▒	●	○	●	●	●		●	●	●	●	●	●	○	●	●	●	●
Visual crypto	PassWindow		[52]	●		▒		▒	▒	▒	▒	▒	●	○	●	●	▒	○	●	●	●	○	●	●	▒	●	●	●	●
Hardware tokens	RSA SecurID	IV-G	[34]			▒		●	○	○	▒	▒	▒	▒	●	●	▒	●	●	●	●	●	●	●	●	▒	●	●	●
	YubiKey		[53]			▒		●	○	○	▒	▒	●	▒	●	●	▒	●	●	●	●	●	●	●	●	▒	●	●	●
	IronKey		[54]	○	●	▒	○	○	○	○	▒	▒	●	▒	●	●	▒	●	○			○		●	●	▒	●	●	●
	CAP reader		[55]			▒		●	○	○	▒	▒	▒	▒	●	●	▒	●	●	●	●	●	●	●	●	▒	●	●	●
	Pico		[8]	●	●	▒	●	▒	○	○	▒	▒	▒	▒	▒	▒	▒	●	●	●	●	●	●	●	○	▒	●	●	●
Phone-based	Phoolproof	IV-H	[36]			○		●	○	○	▒	○	○	○	▒	▒	●	●	●	●	●	○	●	●	●	●	●	●	●
	Cronto		[56]			○		●	○	○	▒	▒	○	▒	●	●	▒	●	●	●	●	○	●	●	●	▒	●	●	●
	MP-Auth		[6]			○		●	○	▒	○	○	○	▒	▒	▒	●		○					●	●	●	●	●	●
	OTP over SMS			●	●	○		●	▒	○	○	○	▒	▒	●	●	●	●	●	●	●	○	●	●	○	▒	●	●	●
	Google 2-Step		[57]			○		●	○	○	○	○	▒	▒	●	●	▒	○	○	●	●		●	●	●	●	●	●	●
Biometric	Fingerprint	IV-I	[38]	●	●	●	○	●	○	▒	▒	○	▒	▒	▒	○	▒	●	▒	●					▒	●	●	▒	▒
	Iris		[39]	●	●	●	○	●	○	▒	▒	○	▒	▒	▒	○	▒	●	▒	●					▒	●	○	▒	▒
	Voice		[40]	●	●	●	○	●	○	▒	▒	○	○	▒	○	○	▒	●	▒	○					▒	●	●	▒	▒
Recovery	Personal knowledge		[58]	○		●		●	●	○	●	●	●	▒	●	●	●		▒						●	●	●	●	●
	Preference-based		[59]	○		●		●	○	●	○	●	●	▒	●	▒	▒		○					●		●	●	●	●
	Social re-auth.		[60]			●		●	▒	●	▒	●	●	▒	●	○	▒	○	▒	●	●	○	○	●	●	▒	●	○	▒

●= offers the benefit; ○= almost offers the benefit. *no circle* = does not offer the benefit.
▒= better than passwords; ▒= worse than passwords; *no background pattern* = no change.
We group related schemes into categories. For space reasons, in the present paper we describe at most one representative scheme per category; the companion technical report [1] discusses all schemes listed.

Table I
COMPARATIVE EVALUATION OF THE VARIOUS SCHEMES WE EXAMINED

losing much. They could become a staple of users' coping strategies if passwords remain widespread, enabling as a major advantage the management of an ever-increasing number of accounts (*Scalable-for-Users*). However, the underlying technology remains replayable, static (mainly user-chosen) passwords.

Federated schemes are particularly hard to grade. Proponents note that security is good if authentication to the identity provider (IP) is done with a strong scheme (e.g., one-time passwords or tokens). However in this case usability is inherited from that scheme and is generally poor, per Table I. This also reduces federated schemes to be a placeholder for a solution rather than a solution itself. If authentication to the IP relies on passwords, then the resulting security is only a little better than that of passwords themselves (with fewer password entry instances exposed to attack).

Graphical passwords can approach text passwords on usability criteria, offering some security gain, but static secrets are replayable and not *Resilient-to-Internal-Observation*. Despite adoption for device access-control on some touch-screen mobile devices, for remote web authentication the advantages appear insufficient to generally displace a firmly-entrenched incumbent.

Cognitive schemes show slender improvement on the security of passwords, in return for worse usability. While several schemes attempt to achieve *Resilient-to-Internal-Observation*, to date none succeed: the secret may withstand one observation or two [61], but seldom more than a handful [62]. The apparently inherent limitations [63], [64] of cognitive schemes to date lead one to question if the category can rise above one of purely academic interest.

The hardware token, paper token and phone-based categories of schemes fare very well in security, e.g., most in Table I are *Resilient-to-Internal-Observation*, easily beating other classes. However, that S/KEY and SecurID have been around for decades and have failed to slow down the inexorable rise of passwords suggests that their drawbacks in usability (e.g., not *Scalable-for-Users*, nor *Nothing-to-Carry*, nor *Efficient-to-Use*) and deployability (e.g., hardware tokens are not *Negligible-Cost-per-User*) should not be over-looked. Less usable schemes can always be mandated, but this is more common in situations where a site has a de facto monopoly (e.g., employee accounts or government sites) than where user acceptance matters. Experience shows that the large web-sites that compete for both traffic and users are reluctant to risk bad usability [16]. Schemes that are less usable than passwords face an uphill battle in such environments.

Biometric schemes have mixed scores on our usability metrics, and do poorly in deployability and security. As a major issue, physical biometrics being inherently non-*Resilient-to-Internal-Observation* is seriously compounded by biometrics missing *Easy-Recovery-from-Loss* as well, with re-issuance impossible [9]. Thus, e.g., if malware captures the digital representation of a user's iris, possible replay makes the biometric no longer suitable in unsupervised environments. Hence despite security features appropriate to control access to physical locations under the supervision of suitable personnel, biometrics aren't well suited for unsupervised web authentication where client devices lack a trusted input path and means to verify that samples are live.

B. Extending the benefits list

Our list of benefits is not complete, and indeed, any such list could always be expanded. We did not include resistance to active-man-in-the-middle, which a few examined schemes may provide, or to relay attacks, which probably none of them do. However, tracking all security goals, whether met or not, is important and considering benefits that indicate resistance to these (and additional) attacks is worthwhile.

Continuous authentication (with ongoing assurances rather than just at session start, thereby addressing session hijacking) is a benefit worth considering, although a goal of few current schemes. Positive user affectation (how pleasant users perceive use of a scheme to be) is a standard usability metric we omitted; unfortunately, the literature currently lacks this information for most schemes. The burden on the end-user in migrating from passwords (distinct from the deployability costs of modifying browser and server infrastructure) is another important cost—both the one-time initial setup and per-account transition costs. While ease of resetting and revoking credentials falls within *Easy-Recovery-from-Loss*, the benefit does not include user and system aspects related to ease of renewing credentials that expire within normal operations (excluding loss). Other missing cost-related benefits are low cost for initial setup (including infrastructure changes by all stakeholders); low cost for ongoing administration, support and maintenance; and low overall complexity (how many inter-related "moving parts" a system has). We don't capture continued availability under denial-of-service attack, ease of use on mobile devices, nor the broad category of economic and business effects—e.g., the lack of incentive to be a relying party is cited as a main reason for OpenID's lack of adoption [28].

We have not attempted to capture these and other benefits in the present paper, though all fit into the framework and could be chosen by others using this methodology. Alas, many of these raise a difficulty: assigning ratings might be even more subjective than for existing benefits.

C. Additional nuanced ratings

We considered, but did not use, a "fatal" rating to indicate that a scheme's performance on a benefit is so poor that the scheme should be eliminated from serious consideration. For example, the 2–3 minutes required for authentication using the Weinshall or Hopper-Blum schemes may make them "fatally-non-*Efficient-to-Use*", likely preventing widespread adoption even if virtually all other benefits were provided.

We decided against this because for many properties, it isn't clear what level of failure to declare as fatal.

We also considered a "power" rating to indicate that a scheme optionally enables a benefit for power users—e.g., OpenID could be rated "amenable-to-*No-Trusted-Third-Party*" as users can run their own identity servers, in contrast to Facebook Connect or Microsoft Passport. The popularity of webmail-based password reset indicates most users accede to a heavily-trusted third party for their online identities already, so "amenable-to" may suffice for adoption. OpenID is arguably amenable to every security benefit for power users, but doesn't provide them for common users who use text passwords to authenticate to their identity provider. However, as one could argue for an amenable-to rating for many properties of many schemes, we maintained focus on properties provided by default to all users.

D. Weights and finer-grained scoring

We reiterate a caution sounded at the end of Section II: the benefits chosen as metrics are not all of equal weight. The importance of any particular benefit depends on target use and threat environment. While one could assign weights to each column to compute numerical scores for each scheme, providing exact weights is problematic and no fixed values would suit all scenarios; nonetheless, our framework allows such an endeavour. For finer-grained evaluation, table cell scores like *partially* could also be allowed beyond our very coarse {*no*, *almost*, *yes*} quantization, to further delineate similar schemes. This has merit but brings the danger of being "precisely wrong", and too fine a granularity adds to the difficulty of scoring schemes consistently. There will be the temptation to be unrealistically precise ("If scheme X gets 0.9 for this benefit, then scheme Y should get at most 0.6"), but this demands the ability to maintain a constant level of precision *repeatably* across all cells.

We have resisted the temptation to produce an aggregate score for each scheme (e.g., by counting the number of benefits achieved), or to rank the schemes. As discussed above, fatal failure of a single benefit or combined failure of a pair of benefits (e.g., not being *Resilient-to-Internal-Observation* and fatally failing *Easy-Recovery-from-Loss* for biometrics) may eliminate a scheme from consideration. Thus, seeking schemes purely based on high numbers of benefits could well prove but a distraction.

Beyond divergences of judgement, there will no doubt be errors in judgement in scoring. The table scoring methodology must include redundancy and cross-checks sufficient to catch most such errors. (Our exercise involved one author initially scoring a scheme row, co-authors verifying the scores, and independently, cross-checks within columns to calibrate individual benefit ratings across schemes; useful clarifications of benefit definitions often resulted.) Another danger in being "too precise" arises from scoring on second-hand data inferred from papers. Coarsely-quantized but self-consistent scores are likely better than inconsistent ones.

On one hand, it could be argued that different application domains (e.g., banking vs. gaming) have different requirements and that therefore they ought to assign different weights to the benefits, resulting in a different choice of optimal scheme for each domain. However on the other hand, to users, a proliferation of schemes is in itself a failure: the meta-scheme of "use the best scheme for each application" will score rather poorly on *Scalable-for-Users*, *Easy-to-Learn* and perhaps a few other usability benefits.

E. Combining schemes

Pairs of schemes that complement each other well in a two-factor arrangement might be those where *both* achieve good scores in usability and deployability and *at least one* does so in security—so a combined scheme might be viewed as having the AND of the usability-deployability scores (i.e., the combination does not have a particular usability or deployability benefit unless both of the schemes do) and the OR of the security scores (i.e., the combination has the security benefit if either of the schemes do). An exception would appear to be the usability benefit *Scalable-for-Users* which a combination might inherit from either component.

However, this is necessarily just a starting point for the analysis: it is optimistic to assume that two-component schemes always inherit benefits in this way. Wimberly and Liebrock [65] observed that the presence of a second factor caused users to pick much weaker passwords than if passwords alone were used to protect an account—as predicted by Adams's "risk thermostat" model [66]. Thus, especially where user choice is involved, there can be an erosion of the efficacy of one protection when a second factor is known to be in place. Equally, defeating one security mechanism may also make it materially easier to defeat another. We rated, e.g., Phoolproof *Quasi-Resilient-to-Internal-Observation* because it requires an attacker to compromise both a PC and a mobile device. However, malware has already been observed in the wild which leverages a compromised PC to download further malware onto mobile devices plugged into the PC for a software update [67].

See O'Gorman [9] for suggested two-factor combinations of biometrics, passwords, and tokens, for various applications (e.g., combining a hardware token with a biometric). Another common suggestion is pairing a federated scheme with a higher-security scheme, e.g., a hardware token.

VI. Concluding remarks

The concise overview offered by Table I allows us to see high level patterns that might otherwise be missed. We could at this stage draw a variety of conclusions and note, for example, that graphical and cognitive schemes offer only minor improvements over passwords and thus have little hope of displacing them. Or we could note that most of

the schemes with substantial improvements in both usability and security can be seen as incarnations of Single-Sign-On (including in this broad definition not only federated schemes but also "local SSO" systems [26] such as password managers or Pico). Having said that, we expect the long-term scientific value of our contribution will lie not as much in the raw data distilled herein, as in the methodology by which it was assembled. A carefully crafted benefits list and coherent methodology for scoring table entries, despite inevitable (albeit instructive) disagreements over fine points of specific scores, allows principled discussions about high level conclusions.

That a Table I scheme (the CAP reader) scored full marks in security does not at all suggest that its real-world security is perfect—indeed, major issues have been found [55]. This is a loud warning that it would be unwise to read absolute verdicts into these scores. Our ratings are useful and we stand by them, but they are not a substitute for independent critical analysis or for considering aspects we didn't rate, such as vulnerability to active man-in-the-middle attacks.

We note that the ratings implied by scheme authors in original publications are often not only optimistic, but also incomplete. Proponents, perhaps subconsciously, often have a biased and narrow view of what benefits are relevant. Our framework allows a more objective assessment.

In closing we observe that, looking at the green (vertical) and red (horizontal) patterns in Table I, most schemes do better than passwords on security—as expected, given that inventors of alternatives to passwords tend to come from the security community. Some schemes do better and some worse on usability—suggesting that the community needs to work harder there. But *every* scheme does worse than passwords on deployability. This was to be expected given that the first four deployability benefits are defined with explicit reference to what passwords achieve and the remaining two are natural benefits of a long-term incumbent, but this uneven playing field reflects the reality of a decentralized system like the Internet. Marginal gains are often not sufficient to reach the activation energy necessary to overcome significant transition costs, which may provide the best explanation of why we are likely to live considerably longer before seeing the funeral procession for passwords arrive at the cemetery.

Acknowledgments

The authors thank the anonymous reviewers whose comments helped improve the paper greatly. Joseph Bonneau is supported by the Gates Cambridge Trust. Paul C. van Oorschot is Canada Research Chair in Authentication and Computer Security, and acknowledges NSERC for funding the chair and a Discovery Grant; partial funding from NSERC ISSNet is also acknowledged. This work grew out of the Related Work section of Pico [8].

References

[1] J. Bonneau, C. Herley, P. C. van Oorschot, and F. Stajano, "The quest to replace passwords: A framework for comparative evaluation of web authentication schemes," University of Cambridge Computer Laboratory, Tech Report 817, 2012, www.cl.cam.ac.uk/techreports/UCAM-CL-TR-817.html.

[2] R. Morris and K. Thompson, "Password security: a case history," *Commun. ACM*, vol. 22, no. 11, pp. 594–597, 1979.

[3] A. Adams and M. Sasse, "Users Are Not The Enemy," *Commun. ACM*, vol. 42, no. 12, pp. 41–46, 1999.

[4] C. Herley and P. C. van Oorschot, "A research agenda acknowledging the persistence of passwords," *IEEE Security & Privacy*, vol. 10, no. 1, pp. 28–36, 2012.

[5] D. Florêncio and C. Herley, "One-Time Password Access to Any Server Without Changing the Server," *ISC 2008, Taipei.*

[6] M. Mannan and P. C. van Oorschot, "Leveraging personal devices for stronger password authentication from untrusted computers," *Journal of Computer Security*, vol. 19, no. 4, pp. 703–750, 2011.

[7] S. Chiasson, E. Stobert, A. Forget, R. Biddle, and P. C. van Oorschot, "Persuasive cued click-points: Design, implementation, and evaluation of a knowledge-based authentication mechanism," *IEEE Trans. on Dependable and Secure Computing*, vol. 9, no. 2, pp. 222–235, 2012.

[8] F. Stajano, "Pico: No more passwords!" in *Proc. Sec. Protocols Workshop 2011*, ser. LNCS, vol. 7114. Springer.

[9] L. O'Gorman, "Comparing passwords, tokens, and biometrics for user authentication," *Proceedings of the IEEE*, vol. 91, no. 12, pp. 2019–2040, December 2003.

[10] K. Renaud, "Quantification of authentication mechanisms: a usability perspective," *J. Web Eng.*, vol. 3, no. 2, pp. 95–123, 2004.

[11] R. Biddle, S. Chiasson, and P. C. van Oorschot, "Graphical Passwords: Learning from the First Twelve Years," *ACM Computing Surveys*, vol. 44, no. 4, 2012.

[12] J. Nielsen and R. Mack, *Usability Inspection Methods*. John Wiley & Sons, Inc, 1994.

[13] J. Bonneau and S. Preibusch, "The password thicket: technical and market failures in human authentication on the web," in *Proc. WEIS 2010*, 2010.

[14] J. Bonneau, "The science of guessing: analyzing an anonymized corpus of 70 million passwords," *IEEE Symp. Security and Privacy*, May 2012.

[15] K. Fu, E. Sit, K. Smith, and N. Feamster, "Dos and don'ts of client authentication on the web," in *Proc. USENIX Security Symposium*, 2001.

[16] D. Florêncio and C. Herley, "Where Do Security Policies Come From?" in *ACM SOUPS 2010: Proc. 6th Symp. on Usable Privacy and Security*.

[17] L. Falk, A. Prakash, and K. Borders, "Analyzing websites for user-visible security design flaws," in *ACM SOUPS 2008*, pp. 117–126.

[18] S. Gaw and E. W. Felten, "Password Management Strategies for Online Accounts," in *ACM SOUPS 2006: Proc. 2nd Symp. on Usable Privacy and Security*, pp. 44–55.

[19] D. Florêncio and C. Herley, "A large-scale study of web password habits," in *WWW '07: Proc. 16th International Conf. on the World Wide Web*. ACM, 2007, pp. 657–666.

[20] D. Balzarotti, M. Cova, and G. Vigna, "ClearShot: Eavesdropping on Keyboard Input from Video," in *IEEE Symp. Security and Privacy*, 2008, pp. 170–183.

[21] B. Kaliski, *RFC 2898: PKCS #5: Password-Based Cryptography Specification Version 2.0*, IETF, September 2000.

[22] Mozilla Firefox, ver. 10.0.2, www.mozilla.org/.

[23] A. Pashalidis and C. J. Mitchell, "Impostor: A single sign-on system for use from untrusted devices," *Proc. IEEE Globecom*, 2004.

[24] R. M. Needham and M. D. Schroeder, "Using encryption for authentication in large networks of computers," *Commun. ACM*, vol. 21, pp. 993–999, December 1978.

[25] J. Kohl and C. Neuman, "The Kerberos Network Authentication Service (V5)," United States, 1993.

[26] A. Pashalidis and C. J. Mitchell, "A Taxonomy of Single Sign-On Systems," in *Proc. ACISP 2003, Information Security and Privacy, 8th Australasian Conference*. Springer LNCS 2727, 2003, pp. 249–264.

[27] D. Recordon and D. Reed, "OpenID 2.0: a platform for user-centric identity management," in *DIM '06: Proc. 2nd ACM Workshop on Digital Identity Management*, 2006, pp. 11–16.

[28] S.-T. Sun, Y. Boshmaf, K. Hawkey, and K. Beznosov, "A billion keys, but few locks: the crisis of web single sign-on," *Proc. NSPW 2010*, pp. 61–72.

[29] B. Laurie, "OpenID: Phishing Heaven," January 2007, www.links.org/?p=187.

[30] R. Jhawar, P. Inglesant, N. Courtois, and M. A. Sasse, "Make mine a quadruple: Strengthening the security of graphical one-time pin authentication," in *Proc. NSS 2011*, pp. 81–88.

[31] L. Lamport, "Password authentication with insecure communication," *Commun. ACM*, vol. 24, no. 11, pp. 770–772, 1981.

[32] N. Haller and C. Metz, "RFC 1938: A One-Time Password System," 1998.

[33] M. Kuhn, "OTPW — a one-time password login package," 1998, www.cl.cam.ac.uk/~mgk25/otpw.html.

[34] RSA, "RSA SecurID Two-factor Authentication," 2011, www.rsa.com/products/securid/sb/10695_SIDTFA_SB_0210.pdf.

[35] P. Bright, "RSA finally comes clean: SecurID is compromised," Jun. 2011, arstechnica.com/security/news/2011/06/rsa-finally-comes-clean-securid-is-compromised.ars.

[36] B. Parno, C. Kuo, and A. Perrig, "Phoolproof Phishing Prevention," in *Proc. Fin. Crypt. 2006*, pp. 1–19.

[37] A. K. Jain, A. Ross, and S. Pankanti, "Biometrics: a tool for information security," *IEEE Transactions on Information Forensics and Security*, vol. 1, no. 2, pp. 125–143, 2006.

[38] A. Ross, J. Shah, and A. K. Jain, "From Template to Image: Reconstructing Fingerprints from Minutiae Points," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 29, no. 4, pp. 544–560, 2007.

[39] J. Daugman, "How iris recognition works," *IEEE Trans. Circuits Syst. Video Techn.*, vol. 14, no. 1, pp. 21–30, 2004.

[40] P. S. Aleksic and A. K. Katsaggelos, "Audio-Visual Biometrics," *Proc. of the IEEE*, vol. 94, no. 11, pp. 2025–2044, 2006.

[41] T. Matsumoto, H. Matsumoto, K. Yamada, and S. Hoshino, "Impact of artificial "gummy" fingers on fingerprint systems," in *SPIE Conf. Series*, vol. 4677, Apr. 2002, pp. 275–289.

[42] LastPass, www.lastpass.com/.

[43] D. P. Kormann and A. D. Rubin, "Risks of the Passport single signon protocol," *Computer Networks*, vol. 33, no. 1–6, 2000.

[44] "Facebook Connect," 2011, www.facebook.com/advertising/?connect.

[45] M. Hanson, D. Mills, and B. Adida, "Federated Browser-Based Identity using Email Addresses," *W3C Workshop on Identity in the Browser*, May 2011.

[46] T. W. van der Horst and K. E. Seamons, "Simple Authentication for the Web," in *Intl. Conf. on Security and Privacy in Communications Networks*, 2007, pp. 473–482.

[47] H. Tao, "Pass-Go, a New Graphical Password Scheme," Master's thesis, School of Information Technology and Engineering, University of Ottawa, June 2006.

[48] D. Weinshall, "Cognitive Authentication Schemes Safe Against Spyware (Short Paper)," in *IEEE Symposium on Security and Privacy*, May 2006.

[49] N. Hopper and M. Blum, "Secure human identification protocols," *ASIACRYPT 2001*, pp. 52–66, 2001.

[50] S. Smith, "Authenticating users by word association," *Computers & Security*, vol. 6, no. 6, pp. 464–470, 1987.

[51] A. Wiesmaier, M. Fischer, E. G. Karatsiolis, and M. Lippert, "Outflanking and securely using the PIN/TAN-System," *CoRR*, vol. cs.CR/0410025, 2004.

[52] "PassWindow," 2011, www.passwindow.com.

[53] Yubico, "The YubiKey Manual, v. 2.0," 2009, static.yubico.com/var/uploads/YubiKey_manual-2.0.pdf.

[54] Ironkey, www.ironkey.com/internet-authentication.

[55] S. Drimer, S. J. Murdoch, and R. Anderson, "Optimised to Fail: Card Readers for Online Banking," in *Financial Cryptography and Data Security*, 2009, pp. 184–200.

[56] Cronto, www.cronto.com/.

[57] Google Inc., "2-step verification: how it works," 2012, www.google.com/accounts.

[58] S. Schechter, A. J. B. Brush, and S. Egelman, "It's no secret: Measuring the security and reliability of authentication via 'secret' questions," in *IEEE Symp. Security and Privacy*, 2009, pp. 375–390.

[59] M. Jakobsson, L. Yang, and S. Wetzel, "Quantifying the Security of Preference-based Authentication," in *ACM DIM 2008: 4th Workshop on Digital Identity Management.*

[60] J. Brainard, A. Juels, R. L. Rivest, M. Szydlo, and M. Yung, "Fourth-factor authentication: somebody you know," in *ACM CCS 2006*, pp. 168–178.

[61] D. Weinshall, "Cognitive Authentication Schemes Safe Against Spyware," *IEEE Symp. Security and Privacy*, 2006.

[62] P. Golle and D. Wagner, "Cryptanalysis of a Cognitive Authentication Scheme," *IEEE Symp. Security and Privacy*, 2007.

[63] B. Coskun and C. Herley, "Can "Something You Know" be Saved?" *ISC 2008, Taipei.*

[64] Q. Yan, J. Han, Y. Li, and H. Deng, "On limitations of designing usable leakage-resilient password systems: Attacks, principles and usability." Proc. NDSS, 2012.

[65] H. Wimberly and L. M. Liebrock, "Using Fingerprint Authentication to Reduce System Security: An Empirical Study," in *IEEE Symp. Security and Privacy*, 2011, pp. 32–46.

[66] J. Adams, "Risk and morality: three framing devices," in *Risk and Morality*, R. Ericson and A. Doyle, Eds. University of Toronto Press, 2003.

[67] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner, "A survey of mobile malware in the wild," in *ACM SPSM 2011: 1st Workshop on Security and Privacy in Smartphones and Mobile Devices*, pp. 3–14.

Q: What authentication schemes do ya use?

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.

12

Lecture 12

lengthy q

Based on the different schemes described in the paper, what do you think would be a reasonable choice for authenticating users in the following scenarios, and what trade-offs would you have to make:

1. Logging in to a public Athena machine in a cluster.
2. Checking your balance on a bank's web site via HTTPS from a private laptop.
3. Accessing Facebook from an Internet cafe.
4. Withdrawing cash from an ATM.

1. Password - also something ya have → # from ID
2. Less concerned w/ ease of use

(I didn't answer very on topic)

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 Friday, 12-Oct-2012 23:31:47 EDT*

10/17

Paper Question 12

Michael Plasmeier

1. At a public Athena terminal we want to avoid *Internal Observations*. At first glance, a biometric system may prevent this, but software can be written to record the biometric over the network, in addition to the user's password.
   The paper does not consider authentication with a trusted machine – but perhaps one is appropriate here. One could imagine a separate authentication box – 1-2 per Athena cluster which are highly protected from their environment (their only function is authentication; housed in a tamper-proof box, etc) which users could use before walking over.
   However, a one-time password could prevent many of the later replays, limiting attackers to stealing your current session only.
2. Perhaps here we are less concerned with *efficient to use*. We want to insure that only we can get on. What this paper does not include, is geographical targeting. Many systems today require additional verification if you are logging on from a new location – either through IP location lookups or leaving cookies on machines you logged in before. I would have such a scheme, with a OTP. Perhaps even the European TAN system where you need to type in a special code only for transfers (which is what really matters) based on the account number you are transferring to and the amount of the transfer.
3. Here again we want to prevent internal-observation. A OTP would prevent attackers from key logging you password and using it later on. To prevent others from attempting to access your account from cafes across the world, FB could require additional verification when you log in from a machine it has not seen you log in from before.
4. Again we have a machine that is to some regards secure – the circuit board could be stored in the safe next to the money. Currently these systems are something you have (card) and something you know (password). A harder to copy card (EMV chip) would make these attacks much harder – though would require swapping machines and cards in the US.

6858

10/17

User Authentication

User ⟷ computer ↪ verifier
server

↑ proxy sever

↑ trusted 3rd party

↑ portable device / pupe

Propose a bunch of criteria

not complete
not orthagional
One of many answers
Why so many → what solving

Prof: easy to read + get depressed

(2)

User —(username, pw)→ Computer —(username, pw)→ Verifier

---

how to store?

User	pw
Alice	hello

Plaintext →

really sucks
password in plain text
Can just steal password

Hash

User	pwhash
Alice	H(hello)

Can't reverse hash
Some chance of hash collision
but almost 0

(3)

Bt users suck at picking passwords

b/w 1,000 → 1,000,000 typical password

ie "123456" is most common in leaked db

So easy to brute force

Make rainbow table

Dictionary lm → ht(dict)

Very fast to hash

lm pw → 15 sec

So use more expensive hash function

Bcrypt

PBkDF2

If connecting over web

Can throttle

or blacklist after x tries

(4)

Salting

User	salt	hash
	random for each user	H(pw ‖ salt) or encrypt

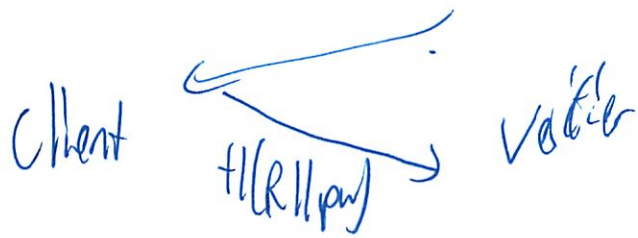
So can't build rainbow table

Large salt 2^{64}

Change salt

How to implement passwords

Challenge-response



(5)

adversary could brute force hashes

want a R_2

to prevent (missed)

but requires verifier have password!

EkE
SRP
PAKE
} schemes that use this

(missed)

(6)

Some sites require special chars
Others prohibit

So can't have a single uniform strong pw

All Schemes

Usability

Remember:
- memoryless
  - password x
  - OTP ✓
  - biometrics ✓

Scalable:
- OpenID ✓
- kerberos ✓
- password x

Carry:
- OTP ✓ Yes
- Pass - No

Effortless:
- must type in

Learn:

Deployment

Anything besides pw - hard to deploy

Accessibility

Cost per User

Server Compat

Browser Compat

Mature

Non proprietary

Security

physical observation
- Yubikey

targeted impersonation
- someone who knows you

Guessing → throttled / unthrottled

internal obs

leaks from verifier

Phishing

trusted path.

explicit consent

Unlinkable

(7)

efficient?
freq errs
recover from loss

Password Managers

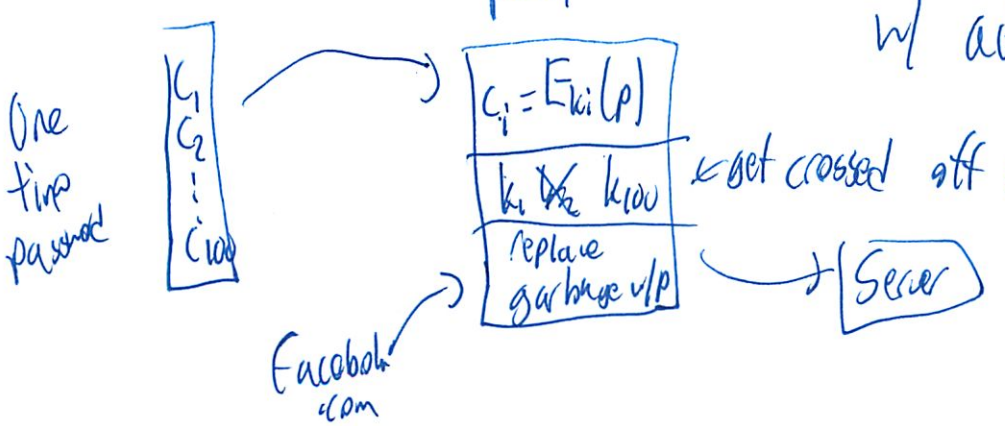
Can use long, diff passwords
auto fill
only autofills on proper site

(use mental energy to store 1 thing very well)

Proxy

UR SA

proxy
replaces magic string
w/ actually pw



trust proxy to some extent

(8)

Though neither on its own is strong enough

Designed in scenarios where don't trust machine
ie an Athena machine

but ~~for that session~~
Mal can steal your current session
but protect your long-term security
Could attacker change your password?

CapReader
Has it all at design level
Prof: but it's broken cryptographically
don't use it!

(9)

Graphical Authentication

PCCP

User gets 5 images

User clicks same place on images in same order each time

Android's draw thing

Category they missed:

usability on touch screen

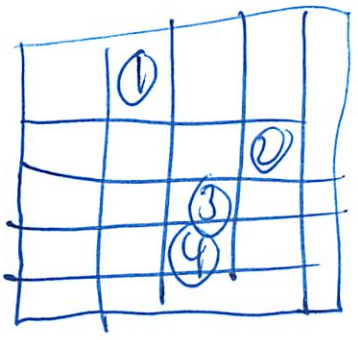
Grays at commonly used areas when setting up

Human Version of Challenge Response

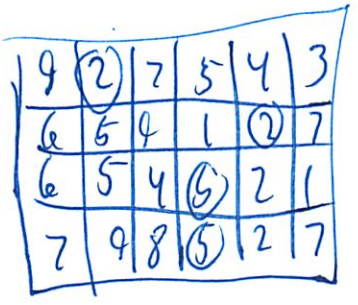
Grid Sure

(10)

Enroll



Auth



remember pattern and type it in → 2255

but worse entropy than pw

but its easy to figure at trace from multiple patterns w/ only a few examples

(12)

Single Sign On

meta-authentication scheme

(missed)

One Time Password

Mostly used

$

H_0 → Secret pw → $H_1 \rightarrow H_2 \rightarrow H_3 \rightarrow \cdots \rightarrow H_{n-1} \rightarrow H_n$

Server ↓

H_n

H_{n-1}
H_2
H_{n-3}
⋮

Cross off when list when used

Server makes it over

typing in last pw invalidates all

(12)

Cumbersome w/ list of papers to carry around

Biometrics

had to get concrete bits to compare

either very easy to trick

Or always complain

FP: 13 bits

exe: 20 bits

voice: 52 bits

Prof: not sure how we actually extract these bits

but if compromised → can't change!

easy-ish to copy

real finger vs ~~replay~~ take finger

~~fake~~

or just replay the data!

but lots of linkability! - same fingerprint

(13)

But w/ a trusted terminal
harder → need take time

Using same credential for multiple things
Good idea: split authentication

Save pw to download a free app
and wipe your Mac
So pw has to ~~know~~ be very usable

ATM
may skip easy to use
- internal observation

TAN type in act # to, from, amt
so can't change any of the 3 from underneath
a user

10/17

User authentication

Problem: how to authenticate users?

- Setting: user <-> computer <-> verifier server.
- Potential extra components might help authentication:
  - A trusted third party.
  - User's portable device (either dedicated or app in mobile phone).
  - A proxy server.
- This paper proposes a number of criteria to evaluate authentication schemes.
- Proposed criteria are reasonable; sometimes non-orthogonal, and not complete.
- Useful as a starting point to think about a new authentication scheme.

Standard authentication: passwords.

- User types in username and password.
- Server checks whether password is correct for that username.

How to store passwords?

- Server needs to be able to verify passwords.
- Naive plan: store plaintext passwords.
  - Problem: if adversary compromises server, gets full list of passwords.
- Hashing: store a table of (username, H(password)>.
  - Can still check a password: hash the supplied string, compare with table.
  - Hard for adversary to invert the hash function.
  - Problem: password space is quite small.
    - Top 5000 password values account for 20% of users.
    - Skewed distribution towards common passwords chosen by many users.
    - Yahoo password study: rule-of-thumb passwords are 10-20 bits of entropy.
  - Problem: hash functions optimized for performance -- helps adversary here.
    - E.g., my laptop can do ~2M SHA1's (of small blocks) per second.
    - Even with reasonable password (20 bits entropy), crack one account/second.
- Expensive key-derivation function (e.g., PBKDF2 or BCrypt).
  - Replace the hash with a much more expensive hash function.
  - Key-derivation functions have adjustable cost: make it arbitrarily slow.
    - E.g., can make hash cost be 1 second -- O(1M) times slower than SHA1.
  - Internally, often performs repeated hashing using a slow hash.
  - Problem: adversary can build "rainbow tables".
    - Table of password-to-hash mappings.
    - Expensive to compute, but helps efficiently invert hashes afterwards.
    - Only need to build this rainbow table for dictionary of common passwords.
  - Roughly: 1-second expensive hash -> 1M seconds = 10 days to hash common pws.
    - After that, can very quickly crack common passwords in any password db.
- Better: use salting.
  - Input some additional randomness into the password hash: H(salt, pw).
  - Where does the salt value come from? Stored on server in plaintext.
  - Why is this better if adversary compromises the salt too?
    - Cannot build rainbow tables.
  - Choose a long random salt.
  - Choose a fresh salt each time user changes password.

How to transmit passwords?

- Poor idea: sending password to the server in cleartext.
- Slightly better: send password over encrypted connection.
- Why is this bad?
  - Connection may be intercepted.
  - Shared passwords mean that one server can use password on another server.
- Strawman alternative: send hash of password, instead of the password.
  - Not so great: hash becomes a "password equivalent", can still be resent.
- Better alternative: challenge-response scheme.
  - User and server both know password.
  - Server sends challenge R.
  - User responds with H(R || password).
  - Server checks if response is H(R || password).
  - Server convinced user knows password (modulo MITM attacks), if it knew it.
  - Server does not learn password if it didn't already know it.

```
    How to prevent server from brute-force guessing password based on H() value?
      Expensive hash + salting.
      Allow client to choose some randomness too: guard against rainbow tables.
    To avoid storing the real password on the server, use protocol like SRP.
  But challenge-response requires client-side and server-side changes.

How to check passwords?
  Guessing attacks are a problem because of small key space.
  Kerberos v4, and v5 without preauth: not great -- offline guessing.
  Rate-limiting authentication attempts is important.
  What to do after many failed authentication attempts?

What matters in user's password choice?
  Many sites impose certain requirements on passwords (e.g., length, chars).
  In reality, what matters is entropy.
  Format requirements rarely translate into higher entropy.
  Defeats only the simplest dictionary attacks.
  Also has an unfortunate side-effect of complicating password generation.
    E.g., no single password-gen algorithm satisfies every possible web site.
    Conflicting length, symbol rules.
  Password distribution "key spaces" are quite small in practice [above].

Password recovery.
  Important part of the overall security story.
  Recall story with Sarah Palin's email account, etc.
  Think of this as yet another authentication mechanism.
  Composing authentication mechanisms is tricky: are both or either required?
  Recovery mechanisms are typically "either".
  Sometimes composing "both" is a good idea: token/paper + password/PIN, etc.

What factors do the authors suggest are important in replacement schemes?
  Table I.
  Why are these factors important?
  What are some schemes that fail at each of the factors?
  What are some schemes that manage to achieve each of the factors?

Password managers.
  Why resilient to phishing?
  Why poor quality passwords?

Proxy-based: URRSA.
  What problem does it solve?
  How does it work?
    User has some password P.
    Proxy stores many keys K_i, generates C_i = E_{K_i}(P).
    Proxy keeps track of whether each C_i has been used (initially unused).
    User gets a printout with all C_i values.
    To log in, user sends an unused C_i to proxy.
    Then user visits target login page via proxy server.
    User submits a fake password, proxy replaces fake password with real one.
  How does it rank on the metrics?

Graphical authentication scheme: PCCP.
  Sequence of 5 images, user remembers points on each image.
  Interesting design point: suggest random points to remember on each image.
  How does it rank on the metrics?

Cognitive authentication (human challenge-response): GrIDsure.
  5x5 grid, user chooses sequence of cells when registering.
  As challenge, server populates grid with random numbers.
  As response, user types in numbers from the chosen sequence of cells.
  How does it rank on the metrics?

Single-signon (SSO).
  Like Kerberos, will talk more about OAuth specifically next week.
```

```
  "Popular" protocols: OpenID, OAuth.  "Signin with Facebook".
  SSO can be thought of as a meta-authentication system: composes with others.
    Security/usability depends on how user authenticates with identity provider.
    Usability costs amortized over many sites that share an identity provider.
  Many sites want to have direct control over trust relationship w/ user, etc.
    SSO makes one site dependent on another trusted third party.

One-time passwords.
  Hash chaining.
  OTPW: avoid chaining because last password reveals entire chain.

Hardware tokens.
  RSA SecurID.
  Google's 2-step authentication.
    Similar to hash-based challenge-response scheme; implicit challenge = time.

Biometrics.
  How big is the keyspace?
    Fingerprints: ~13.3 bits.
    Iris scan: ~19.9 bits.
    Voice recognition: ~11.7 bits.
  Problem: hard to recover from loss of "personal" information.
  Perhaps more sensible in the presence of trusted biometric reader devices.
    But not practical for web application authentication.

Try ranking some familiar authentication schemes.
  Kerberos v4?  v5?
    Usability: renewing credentials.
  SSL client certificates?
  Challenge-response scheme?
  Bank of America "SiteKey" (recognize image, type in a word)?

How effective is scheme combining?  Which ranking criteria compose well?
  "Either" vs. "both".

What factors are difficult to achieve?  [par 4 of section V]
  Memorywise-Effortless + Nothing-to-Carry.
  Memorywise-Effortless + Resilient-to-Theft.
    Either user remembers something, or it can be stolen, except for biometric.
  Server-Compatible + Resilient-to-Internal-Observation.
  Server-Compatible + Resilient-to-Leaks-from-Other-Verifiers.
    Server compatible means sending a password.
    Passwords can be stolen on user machine, replayed by one server to another.

What are potential answers to the homework questions?  What factors matter?
  Logging into public Athena machine?
    Resilient-to-Internal-Observation: easy to install malware on machine.
    Resilient-to-Physical-Observation: though maybe less so.
    MIT IDs could be a good thing to leverage (smartcard?).
    Biometrics?  Untrusted terminals, probably not a great plan.
    Would some proxy-based schemes work (like URRSA)?
  Checking bank balance via HTTPS from private laptop?
    Less relevant: Resilient-to-Physical-Observation, Resilient-to-Theft.
    Good idea: separate transfer operations from looking at balance.
      "Progressive authentication".
      Would be a good thing for Athena machines too.
      Get access to a browser to check maps or print paper vs. websis.
    Password managers make sense for private/trusted machines.
  Accessing Facebook from Internet cafe?
    Password managers not a good idea here.
    How sensitive is the data?
    Might be leveraged to authenticate to other sites.
    (Either "Login with facebook" or by answering personal questions.)
    Maybe authentication-via-proxy (URRSA).
  Withdrawing cash from ATM?
```

```
    Security matters highly.
      Resilient-to-Physical-Observation.
      Resilient-to-Theft.
      Unlinkability less so.
    Possibly trusted terminal: biometrics might be worth considering.
    (Although in practice bank may not want to trust the terminals.)
    One thing that matters but wasn't included in list: authenticating txn.
      Adversary can re-purpose entered credentials for different operation.
      Hardware-token-based or phone-based solution could authenticate txn.
      E.g., H(challenge || pw || withdrawal-amt || atm-location).

What other factors should we worry about in user authentication?  [sec V.B]
  Continuous authentication, instead of session start.
  Migration cost from passwords / incentives for deployment (OpenID).
  Renewing credentials (Kerberos).
  Availability / DoS attacks.

Why aren't these schemes widely used?
  No single answer.
  Convenience of passwords.
  For many scenarios, security isn't important enough to justify switching cost.
    Per-user cost on the server, on the user's end, software changes, etc.
  Limited benefits of some alternative schemes.
  Often hard for an individual user to improve his/her own security.
    Perhaps partially fixed with SSO, where users can choose a better IdP.

References:
  Full tech report: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-817.pdf
  http://www.cl.cam.ac.uk/~jcb82/doc/B12-IEEESP-analyzing_70M_anonymized_passwords.pdf

[ For next year:
  Talk less about specific authentication schemes.
  Talk more about range of issues authentication schemes might need to address.
  Talk about how to compose authentication schemes: two-factor, recovery, etc. ]
```

6.858 Fall 2012 Lab 3: Server-side sandboxing

Reprint only part 2

Handed out: Wednesday, October 3, 2012
All parts due: Friday, October 19, 2012 (5:00pm)

Part 3: Extending the PyPy sandbox

In this part of the lab, we will extend the PyPy sandbox implemented in `pypysandbox.py` to support operations needed for a profile to store persistent data, and to access zoobar application state.

Exercise 3. Implement a writable and persistent `/tmp` directory in the PyPy sandbox. This is needed by the `visit-tracker.py` and `last-visits.py` profiles to store their persistent information. Make sure that the `/tmp` directory seen by each user's profile is separate from other users, so that profiles of different users cannot tamper with each other's files. As in lab 2, remember to consider the possibility of usernames with special characters.

Ensure that the `visit-tracker` and `last-visits` profiles work correctly after you implement your changes. The `/tmp` check should also pass at this time.

Exercise 4. Since the standard Python SQLite module is implemented by calling into the native SQLite C/C++ library, it is not available in the PyPy sandbox (because the native library does not know how to forward its system calls via the RPC channel). In this exercise, your job is to support the `get_xfers()` function from `proflib.py` in the sandbox. A reasonable approach to do this is to extend the RPC server (`pypysandbox.py`) to perform the `get_xfers()` functionality on behalf of the sandboxed code. You will also need to modify `proflib.py` to invoke this new interface.

Hint: to create a new interface between code in the sandbox and the RPC server outside of the sandbox, such as for performing `get_xfers()` calls, consider overloading the file namespace by defining a special file name that corresponds to `get_xfers` calls. You can take a look at `VirtualizedSocketProc` in `.../pypy-sandbox/pypy/translator/sandbox/sandlib.py` to see an example of how the PyPy sandbox exposes access to TCP sockets in this manner.

Once you are finished with this exercise, the `xfer-tracker.py` should be functioning correctly in the sandbox.

Exercise 5. Implement the last remaining parts of `proflib.py` in the sandbox: `get_user()` and `xfer()`. Once you are done, `granter.py` should work from within the sandbox.

Challenge! (optional) For extra credit, allow sandboxed code to safely manipulate sub-directories under `/tmp` using `mkdir` and `rmdir`, to open files in those sub-directories, and to be able to `unlink` and `rename` files and sub-directories. Write an example Python profile that uses sub-directories and renames files.

Challenge! (optional) Allow sandboxed code to safely create and use symlinks inside of its `/tmp` directory.

You are done! Run `make submit` to upload `lab3-handin.tar.gz` to the submission web site.

```
53          # * can access its own executable
54          # * can access the pure Python libraries
55          # * can access the temporary usession directory as /tmp
56          exclude = ['.pyc', '.pyo']
57          tmpdirnode = RealDir('/tmp/sandbox-root', exclude=exclude)
58          libroot = str(LIB_ROOT)
59
60          return Dir({
61              'bin':  Dir({'pypy-c':     RealFile(self.executable),
62                           'lib-python': RealDir(libroot + '/lib-python', exclude=
    exclude),
63                           'lib_pypy':   RealDir(libroot + '/lib_pypy',   exclude=
    exclude)
64                          }),
65              'proc': Dir({'cpuinfo': RealFile('/proc/cpuinfo'), }),
66              'tmp':  tmpdirnode,
67              'proflib': RealFile('/jail/zoobar/proflib-sb.py')
68              })
69
70      ## Implement / override system calls
71      ##
72      ## Useful reference:
73      ##    pypy-sandbox/pypy/translator/sandbox/sandlib.py
74      ##    pypy-sandbox/pypy/translator/sandbox/vfs.py
75      ##
76      def do_ll_os__ll_os_geteuid(self):
77          return 0
78
79      def do_ll_os__ll_os_getuid(self):
80          return 0
81
82      def do_ll_os__ll_os_getegid(self):
83          return 0
84
85      def do_ll_os__ll_os_getgid(self):
86          return 0
87
88      def do_ll_os__ll_os_fstat(self, fd):
89          ## Limitation: fd's 0, 1, and 2 are not in open_fds table
90          f = self.get_file(fd)
91          try:
92              return os.fstat(f.fileno())
93          except:
94              raise OSError(errno.EINVAL)
95      do_ll_os__ll_os_fstat.resulttype = s_StatResult
96
97      def do_ll_os__ll_os_open(self, vpathname, flags, mode):
98          if flags & (os.O_CREAT):
99              dirnode, name = self.translate_path(vpathname)
100             ## LAB 3: handle file creation
101
102         node = self.get_node(vpathname)
```

```
import os, sys, errno
from cStringIO import StringIO

pypy_sandbox_dir = '/zoobar/pypy-sandbox'
sys.path = [pypy_sandbox_dir] + sys.path

from pypy.translator.sandbox import pypy_interact, sandlib, vfs
from pypy.translator.sandbox.vfs import Dir, RealDir, RealFile
from pypy.rpython.module.ll_os_stat import s_StatResult
from pypy.tool.lib_pypy import LIB_ROOT

class WritableFile(vfs.RealFile):
    def __init__(self, basenode):
        self.path = basenode.path
    def open(self):
        try:
            return open(self.path, 'wb')
        except IOError, e:
            raise OSError(e.errno, 'write open failed')

class MySandboxedProc(pypy_interact.PyPySandboxedProc):
    def __init__(self, profile_owner, code, args):
        super(MySandboxedProc, self).__init__(
            pypy_sandbox_dir + '/pypy/translator/goal/pypy-c',
            ['-S', '-c', code] + args
        )
        self.debug = True
        self.virtual_cwd = '/'

    ## Replacements for superclass functions
    def get_node(self, vpath):
        dirnode, name = self.translate_path(vpath)
        if name:
            node = dirnode.join(name)
        else:
            node = dirnode
        if self.debug:
            sandlib.log.vpath('%r => %r' % (vpath, node))
        return node

    def handle_message(self, fnname, *args):
        if '__' in fnname:
            raise ValueError("unsafe fnname")
        try:
            handler = getattr(self, 'do_' + fnname.replace('.', '__'))
        except AttributeError:
            raise RuntimeError("no handler for " + fnname)
        resulttype = getattr(handler, 'resulttype', None)
        return handler(*args), resulttype

    def build_virtual_root(self):
        # build a virtual file system:
```

```
103             if flags & (os.O_RDONLY|os.O_WRONLY|os.O_RDWR) != os.O_RDONLY:
104                 ## LAB 3: handle writable files, by not raising OSError in some cases
105                 raise OSError(errno.EPERM, "write access denied")
106                 node = WritableFile(node)
107
108             f = node.open()
109             return self.allocate_fd(f)
110
111         def do_ll_os__ll_os_write(self, fd, data):
112             try:
113                 f = self.get_file(fd)
114             except:
115                 f = None
116
117             if f is not None:
118                 ## LAB 3: if this file should be writable, do the write,
119                 ##        and return the number of bytes written
120                 raise OSError(errno.EPERM, "write not implemented yet")
121
122             return super(MySandboxedProc, self).do_ll_os__ll_os_write(fd, data)
123
124     def run(profile_owner, code, args = [], timeout = None):
125         sandproc = MySandboxedProc(profile_owner, code, args)
126
127         if timeout is not None:
128             sandproc.settimeout(timeout, interrupt_main=True)
129         try:
130             code_output = StringIO()
131             sandproc.interact(stdout=code_output, stderr=code_output)
132             return code_output.getvalue()
133         finally:
134             sandproc.kill()
135
136
```

W258
Lab 3
Pt 2

W13
SP

Get a head start on pt 2 of the lab
↳ well pt 3

Pt 3

Allow sandbox to store persistant data + access Zoobar app state

Last time: just a hello world
w/ access to prof lib.py

Allow writable /tmp directory
- visit-tracker
- last-visits

- Usernames w/ special characters
↳ didn't consider earlier

Each user has their own /tmp directory

(1)

Now how will we going about doing this?
Prob best to do in OH...

What is the existing temp behavior?

There is already a tmp folder specified
Though I could not find it caller...

Currently /jail/tmp

; where could we find?

sudo find / -name "last_visit"

nothing when not running
new

even when running not likyl to change much

(3)

Need to look at how it saves files

It looks like we need to build all this write code

Prob not all that challenging

But if figuring out what to do...

Basically have some tmp somewhere

And we implement the sys calls

---

is a translate path method

Don't use Python OS temp file method

Put in a lot of debugging code

- vpath name: /tmp/last_visit-The ... .dat
- flags 577
- mode 438

(4)

Dir node

Real Dir /tmp/sandbox-root

So it noticed

name : file name

Sandbox address : run profile for Replay

↑ I think line break missing

Fixed file error

Now where is my debug code?

So its still running

Now error w/ RealFile object

I don't know how this is set up

6.858 OH

10/15
LP

So need to figure out how to build these calls

TA: Path seems to be just string

Part of class

Real Dir join op?

look for how translate node returned

dirnode.join(component)

Tstr

/tmp/sandbox-root/theplace/last_visit_...dat

Tr, or

Should be root against jail dir

No change w/ sandbox

TA: Will be created

(2)

TA: What are all those errors

Something returns no file

Write output to file

sudo ./zookld 2> output

redir. stderr at /stderr

In more

/search term

So must create the 2 dir

/sandbox-root/theplace

As the right permissions

0012 for those 2 directories

(3)

Open against s file descriptor
Lneeds fd

How to get a fd

Go os.open(file, flags, mode)
returns fd

TA: automate this stuff — your dev
testing workflow

Looking into what 577, 438 are

They provide an interface

Is open returning a dirnode

(X) Unmarshable Object
On Close

Know atleast speed

(4)

Should open file descriptor

Provide to fd open

Can hard code

as dir node output

or ~~what~~ handler writable file

Understand why flag for create

Diff for creating + writing

Special creation
Write flag
Read only flag

What if name is dot dot

5

Build search

any non alpha numeric

what ever we want to enforce

Write

file d provided

Os. Write

Why super call here?

~~Is fd~~

What is f?

~~Assume~~

look at get_File

Then do proper system call

(6)

file object is class in python w/ convience

depends on if ya want to take advantage of the library

read from start to end of file

6.858
Lab Mae

10/5
7:30p

Should try open lst
But will write not work w/o write
Taesoo look through lib

From Sand lib
ft. returns self.open_fds[fd]

Im not see how do it w/ lib

For same as sand lib
open_fds table are real file objects

(X) Global have Non Writable File not defined
spelling error

(X) No module named proflib
no file /jail/zoobar/proflib-db.py
in the sand lib

(2)

So this related to how Jhang referenced it

What what older stuff called
but this is all running as a service
set up in last lab

run in auth service

---

earlier same close fail

Adding close code

---

protlib is here 544
though

Is it since file not stating
added that code

Oh scrolling up w/ mouse
LS not valid

(3)

But stat should still work!

Or remove /jail
Since everything rel to that

Ok that is not showing up
; But same error...

Or did I remove that lib name?
no - should work
Why did it work for name?

That broke as well!
What happen?
Still converted it over

What did a break?
WTF did I change?
Back to 644?
no

(4)

I don't get it → it reads that file
or our read does not work

Yeah its using our readfile

prob is I am appending /theplat

last_freebie_name1_name2.dat
last_visit_name1_name2.dat
last_visits_name.dat

Built that

(X) Dir obj has no attribute path
but can't see for which file
this is /proc/cpuinto
which we should not consider
dir node is long thing

(5)

Must run w/ subs

But what is that point?

But I can't really see it...

How to kill bg process?

Or restart server

Posted to Piazza @163

Just die on this

Or return w/ std file open

(X) Out of memory error

└ Stuff on piazza

Git pull

So not on piazza

But they talked about this in last week OL

(6)

So tried it pre
L starting w/ that
same error

What is ~~SIG~~ SIGIOT?
= SIGABRT
tells it to terminate
process itself when it calls abort

~~DAC~~ Mode string empty
L added ~
So that was not called
(Well at least getting somewhere)
So that fixed the /proc/cpu thing
but still unmarshable

(1)

; So use old std way

But it does not like the real file thing

build our own

(X) No module pro lib

└ the read open failed w/ that r option

(X) do "rb" to "b"

(X) Proc error

Just "r" ;

(X) No module proflib

; why did read open fail ;

; file permission 777

No

8

Try super

✓ Woot!!!

It works!

└well we are back to hello-user.py which we were before today

Now try next thing

visit-tracker.py

✗ Writable File not defined

└spelling error

Username finding also broken

✓ File created

✗ File has no attribute send

which is what I found from that file

how send lib sends

could try regular writing

(9)

(✓) Could that be working!?

No - nothing actually written....
always reports last write

Try doing super

(✗) op not permitted

Posted to piazza (✓) 165

"or we don't have write permissions"
Lno mode is wb

What is this f, open file object?

Student: f.xxx write()

I was thinking that

(✗) Now all I this value error

(✓) BA file written when I fail it

(10)

Though it does not tell me last visit and file contents +

|

135`...

What should it be?

5
135_ _

So 2 lines...
Why the extra line?

Student i make sure return # of bytes written
Print data

Student i returning # of bytes written must be manually
; so return len(F.read())
⊗ file not open for reading

(11)

So "wb" → "rwb" ?

Oh the + was file name wrapping

(X) still IO err

Online i r+ for reading + writing

the Py instructions don't seem to agree

(X) Now stuck in an ∞ loop for some reason

But result is 0

Why?

is my f.read not working?

Why was it ∞ looping

"didn't think it had written"

One more try on f.read

Lcan always return input

len is 0!

(12)

how it appends
back to w

~~&~~ ✓ No more ∞ looping

✓ File written → No! its not!
old time

last 10:18 P
now 10:22 P

but why did it not read that?

✓ File updated

Still not read!

Why error when opening → no such file
Permissions!

Oh no /theplaz!
it should deal

(13)

Calling super breaks that

So rewrite ↓ pathname

✓ Woot it works

visit count increments!

So are we done?

---

well some cleanup code

- make dir ~~perms~~
- dir file permissions
- last visits ...

So try some file directory code

? Why is my make dir not running?

So needs to be after join

(4)

And it says it makes the dir
file is made

but when error creating it ?
it does not create when I run the real file
join
looks for it there...

So file never created...
I just create it

So I'm trying open() close() just to "touch" it
So file name was wrong

✓ No errors
But does not read...
its always writing the dir

(15)

Check if file exists

(✓) Works again

This is the ribitest POS code ever

So try to clean up a bit

(✓) Still works visit-tracker

(✓) last visits on same profile

(✓) last-visits on other profile

So almost done

Must strip bad chars

(✗) fail

Ex 1 app fun passes

hello user passes

sandbox check passes

but not the other 2

Weird - track down later

16

Or I said alpha only

Need alpha numeric

Prints output

but extra trash

Some tives

✓ seems to be working

✓ Passing much more

✓ visit_trace

✓ test visits

So I am going to call ex 3 done

There is still a /tmp check that fails

OH-ish

10/16

Exercise 4

Support get_xfers() fn from profile in sandbox

Have py.py sandbox.py do it

Modify profile to use new interface

Consider overloading file name space w/ special file

- good plan

Can also look at Virtualized Socket Proc

- wait are we doing that or TCP sockets

Proc = process

~~Where are TCP sockets~~

So it does an open w/ tcp

And socket.connect

(2)

'So do that instead of a file'

TA: notice they trick out file thing
don't try to do any socket.socket stuff

So I'll have it open transfer_IO_replay
and it will do it

'Must it be secure w/ token'

'So instead of calling with sock
Or is auth sock now calling that
It seem modify proflib
So it seems forgot the sock

Anything w/ zodb must be transitoed

(3)

¡And put most logic on other end...

transfer_theplace_iwang_7.dat
q

Pofile wants to see a normal /tmp
But put it in a diff place in each
¡Fake one looks like
Will fail other tests - do other files
So just change tmp dir node
for each user
need to add line to create it
dirnode.join req files exists

9

Make that be /tmp

Then move up to top to make directory

dirnode = obj in virtual file system

open = open real file

~~FILE *~~

open returns file descriptor #

dirobj → rename → file object

join walks down one step

Virtual file root

— a tree of object

Lab is set up to do much of this

(5)

Open w/ O create
↳ flag
man open

python – higher level flags
r, w, r+, b

O create – make it exist

translate path → does lst part for you
stops one before

O create stop at one before
creating file depends on your file system

real dir
or Dir obj

(6)

Can't write in Dir objects
- read only

So /tmp is there

---

1. translate path
   - containing dir object
     - fake file system
       - could be a RealDir object
       - could not
     - (Only real path has path info)

get_node() is a provided fn
- calls translate path + join

(7)

O create

Can pass it on file exists
↳ just ignores that flg

Now decide what it means in real world

Check if Real Dir - then go create
or not

No need to do a permissions check?

Want to check real disk

Have not checked if can opened for writing

tabs = 8 spaces

(8)

either it exists or not
(now we know)
Now can call dirnode.join

Writable File is wrapper for Real File

File 8 is Fake → gives ya data
Relying on OS to do permission
(file)
If fancy → do check

len(data) works
Just to let you know

1. ✓ Hello works
2. ✓ No error
3. ✓ Visit cant works

9

- ✓ App functionality
- ✓ hello-user
- ✓ visit tracker
- ✗ xfr tracker
- ✗ grants
- ✓ ex2 sandbox check
- ✓ ex3 tmp check

get xfrs is read
So is xfer
get user

Need to return in nice format

Make a new proflib box file
Call from that

TA: Python can't import stuff w/ - in name

(10) Do we have to serialize our response

TA: Yes

Ok now implement a bunch of things

(X) string error

oops

(X) file not found

Oh the debug code is causing!

So move the translate-path further down

(X) JSON error

import json

fix ~~file~~ username scrapp

(X) Username still blank

(X) Now function still returns blank even though have username

(11)

Profilebox code never seems to print

Oh grr-wrong fn!

Oh I don't think these actually are any!

Need to special case no result

(✓) So was returning proper json

just need to return correctly

Seriously Confused Py interpreter

Screwed up a syscode

Write returned wrong thing! (the value not fd)

debug-think critically!

~~X~~ Open should returns self.allocate_fd

look at how File looks

make node a Fake File

File obj which gives ya a string

cStringIO in Py

(12)

vfs.py

File

Make a string IO

Allocate File Descriptor

Want Read to return data

Could make a temp file os.tempfile

Or StringIO.

Open → allocate fd [fd] ← returns #s

~~Read → if this is one of our #s~~

~~return our special stuff~~

(13)

Put in a magical file

↳ StringIO

(or a os.tempfile

↑
in open

open returns that

StringIO Library

file-like class

reads + writes a string buffer

initialize w/ value

can .getvalue()

.close()

Ah so can implement directly

or via File()

(14)

So how get fd to interact w/ StringIO

```
s = cStringIO.StringIO(data)
return self.allocate_fd(s)
```

(x) Module obj is not callable

This is like before any code runs...

Python importating

import socket

socket

↳ module

socket.socket

↳ class

from socket import socket

socket

↳ class

(18)

Ahh

that would explain a lot

I should learn this basic Py stuff earlier

✓ Sweet returned []

but ~~didn't~~ printed at extra

So can't print to std error

in prof lib - sb

(started to test by editing jail)

✓ seems to work! - ex 4 done?

✓ ex 1

✓ hello user

✓ visit tracker

✓ xfr tracker

✓ sandbox check

✓ /tmp check

✓ ~~✓~~ ✗ grantr

Next ex

(16)

Ex 5

implement get_user()

xfer()

this may already be done
just need to check / debug

- (X) file error in proflib-sb
  fixed above!
- (X) str + int error
  prof lib line 56
  oh err py
- (/) I gave yar 1 zapbar
  but a lot of link
  also printed

17

✓ It looks to work!

But it did not subtract zoobars

Need to convert to #

(So close!)

Subtraction still does not

Now it creates a zoobar

Oh now it ↓

But it takes time

✓ Think done!

Ogranter.py

```
Username: theplaz

PASS Exercise 1: App functionality
PASS Profile hello-user.py
PASS Profile visit-tracker.py
PASS Profile last-visits.py
PASS Profile xfer-tracker.py
PASS Profile granter.py
PASS Exercise 2: Sandbox check
PASS Exercise 3: /tmp check

Exercise 1: 20/20
Exercise 2: 20/20
Exercise 3: 20/20
Exercise 4: 20/20
Exercise 5: 20/20

Total: 100/100
```

10/30 Rec

Exercise 3 Grade

6.858 Exam 1 Review

10/24

What is the format of the exam?

Open book

Open notes

Open laptop

No internet though

So prep readings ✓ Downloaded

This will likely be like 6.033 - same prof

Exam 1 2011

When did we do XFI?

Not a paper we read this year

Force HTTPS

(7)

So it's the exact same

know the details of the systems in depth

So can answer the questions

Zoobar security

From lab code

Buggy Bards

Browser Same Origin

So some lecture material as well

Static Analysis of JS

Review Notes

Don't write down the little stuff I already know - instead focus on exact details!

Goals - confidentiality

- integrity

- availability

(3)

adding the superintendent to class
now can view everything!

bad assumptions

"Orange book" - security design

buffer overflow
copy string in w/o checking lenght
overwrite return address

multiply can wrap around

strings terminated differently
len
or null

acl checks not on indv ajax calls

economy of mechanism - reuse existing

trust = bad

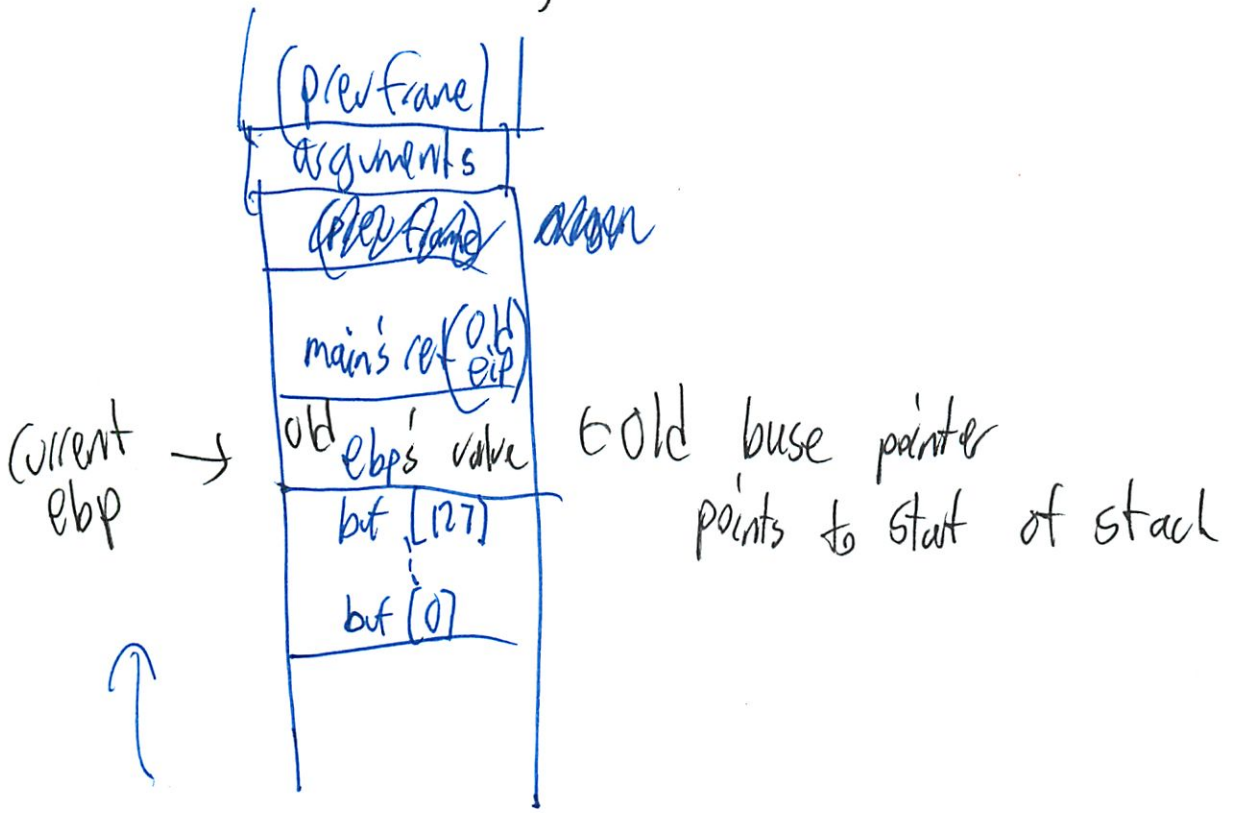
trustworthy = good

(4)

Baggy Bounds Checking

(need to really understand how it works)

(Lecture covers)



Stack canaries

gcc in gcc

(5)

Can be terminator 0, CR, LF, (-1)

can be random

but must store somewhere

doesn't protect function pointer

Bounds Checking

When read pointer have pointer escaped

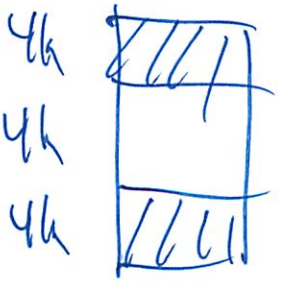
Start + size should = lenght

baggy Checks entire struct

Electric fense

Surrand w/ sep pgs

3 pgs of memory (4k each)



(6)

Fat pointer

every pointer is

```
struct {
    void *ptr
    void *base
    void *limit
```

Check pointer arithmitic & and deferencing

Side data structure

baggy is

must know where pointer came from

Can do Stanford splay tree

set so illegal → seg fault → we catch + try

1)

Nieve → each byte of mem a bit in table

buggy → store the log of
so value must be power of 2
or pad it

Char *q = p + 0x3c
↑ will look up p
↑ compute q
Check if still same allocation

Something about a table:
XOR p, q gives you bits they differ by
This must be < size of the object
Set high order bits to 8 → so program crashes
lots of false alarms
Since some legit art of bounds uses

, going out of then back in bands is legal
depending on how you add

Buggy can only go $\frac{\text{slotsize}}{2}$ out of band
in order to be able to fix it

non executable memory
randomized memory address for system()

When looking up go to next lowest or = multiple

OkWS / Web server Security

Unix protection Mechanisms

User id
grap id
processes have 1 uid, 0 or more Gids
0 = root

(9)

r	w	x
4	2	1

to change permission → must be owner
to create hardlink → to write to it
) no good reason why diff

for intersection of 2 groups

create /a/b/c.txt

group 1 only readable

group 2 only readable

Once have file descriptor → do anything

Memory not shared
except p-trace for same uid

(10)

be root for new user / set up low ports

many things need root access

↳ if 1 compromised ⇒ full access

So setuid binaries

Special flag in inode bit

* run as permission of the owner

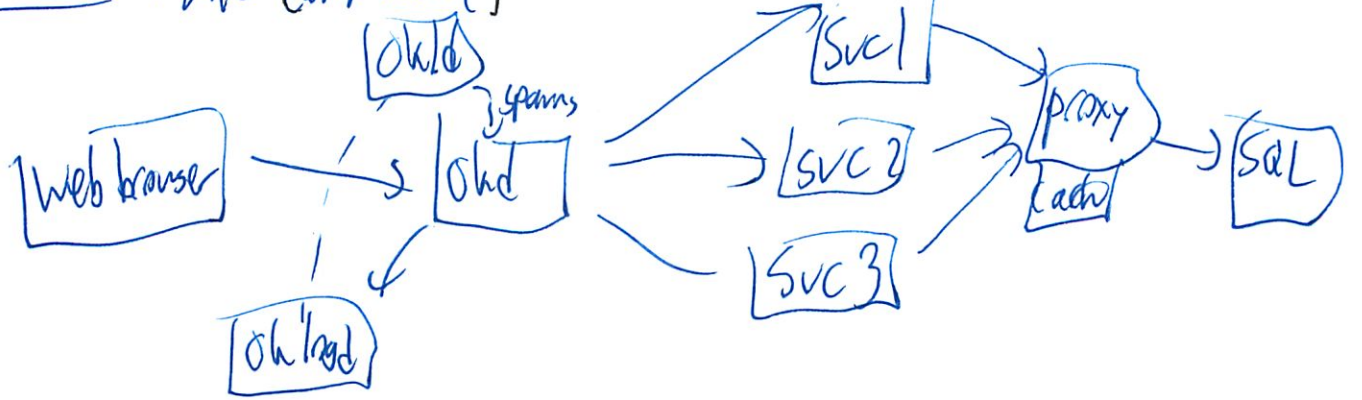
So must trust no buffer overflows

Chroot changes file system namespace

So that there is like root

apache ⇒ all runs as 1 uid

Okws Regex [complicated]



(11)

Proxies limit access to the db
like templated queries

I now have a good sense of how it works from lab 5

Okld stays as root

/var/okws/run/svc

owner = 0 ← read only
gid = 5000 ← execute only

So permissions can't modify
(remember that for lab 4

attack surface → would could touch
svcs have a big one since execute
hardest part: SQL proxies

(12)

Capabilities / Protection Mechanisms / Capsicum

Sandboxes

Only expose certain APIs

filters calls to make sure legit

Which calls does kernel allow?

Plan 0: VM

Very high overhead

Course-grained

but good control

DAC

Discretion Access Control

✓ Ware ✓ App & privileges

Permission → [Object]

at discretion of object owner

(14)

Capabilities unforgetable handles
instead of name
like Unix file descriptors

but w/ coarse or more course grained options

(I did not get this lecture at all)

Notes | break up app into smaller pieces
give sandboxed code some limited access

discretionary → apps set permission on things
check programs' permission to decide

give new Uid
Chroot

(had since half real - half theoritical)

mandatory → users/admins set polices
apps can't change

15

Capicum → if have handle (capability) can access

needs kernal changes
for lots of reasons

libcapsicum → starts new process in Sandbox

cap_enter()
for else leftover data in memory

any process can create a sandbox
but requires code changes
some OSes pre capabilities based

or trying to partition roots' privileges up into
Sep items

for every file → open fd ahead of time
Some lines of code changes
so can't open other fds

This lecture seems to have been cut super short
I did not realize all of the fd stuff

16

Recap Paper) new kernal primatives

Compartmentalizes UNIX apps into logical apps

least-privilege operation

Extension of file descriptor objects
- files
- sockets

inheritence or message passing

decompose old school monolythic apps

cap_enter()
can't be cleared

cap_new(fd, mask of rights)

fget() check extra rights

Change kernal calls

libcapsicum forks new processes

(is this just Fd protection - or more?)

(17)

- ;can only open files under it
  So open those directories under them
  think I kinda get now

ISEC Guest Lecture

(these guys were not that good
-fell asleep too)

Session cookies

all the dif people

Pen testers - find vunerability

fuzzing - giving weird inputs

Confused deputy - send wrong message

Use after free errors in JS

Operation Aurora

Stuxnet

CRC32 → not good for protecting data

(18)

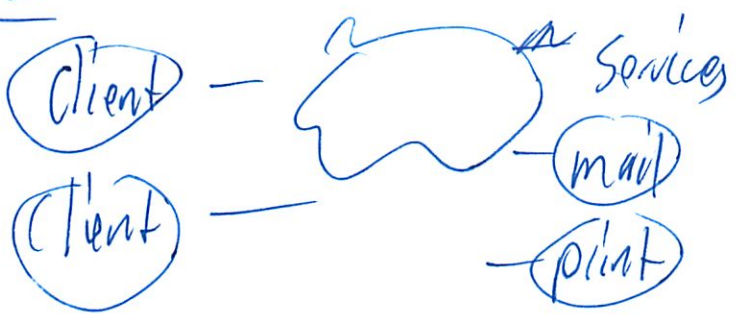
"Our job is to read 1 more sentence in man page"

Think about interactions

Print Spooler

Flame
gets lots of info from phone
Signed by Windows Update

Kerberos

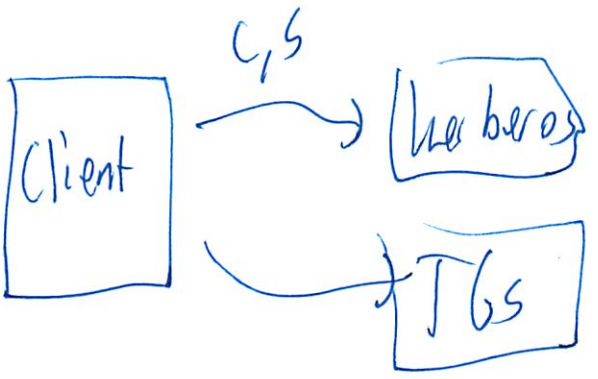


Authenticate client to service

Add central authority - every org runs their own
Must also trust client machine

2 services Kerberos
Ticket Generators

19



to get more tickets for more services

TGs don't require password
So don't have to keep pw in memory

Naming: <name, instance, realm>
Extra

Ac = authenticator
Lservice should know client talking to
(Speeding up studying) (should do more each day)

table of used authentators
nothing binds authentr to request
extra field T to change password
Can't ever reveal your old passwords

29

Diffie Hellman key Ex

This is the exponent $(g^x)^y = g^{xy} = (g^y)^x$ thing

Web App Security

JS code isolated

need to get to visit the page

but want cross site → like FB [Like] button

Window, document global objects

Same origin policy

Frames/windows

Cookies - a slightly similar matching

ie no protocol in JS's access to cookies

Secure flag

(1)

load images
- can't see pixels
- but can see dimension

POST

CSRF Just issue ajax request
often add token
I don't get this - actually I do

XSS have site display script

Should be consistent

Static Analysis

Gives report to developers
hard when dynamic typed
this tool focuses on SQL injection in PHP
lots of example code here

(22)

3 types of bugs

- SQL injection
- directory traversal
- eval

3 levels of analysis

- basic blck
- function
- ?

So for each block

Γ = Gamma = memory map locations like variable names

$\bot$ = unknown

(23)

Store summary

E = error set
variables that might flow into SQL query

R = return set
variables that might end up here
only ~~tainted~~ strings here

S = which values sanitized if result Boolean

X = does this function always exit?

The summary in notes are better

E = memory locations that might flow in
but aren't sanitized

R = parms or global variables in ret val

S = params or global variables sanitized
Can be conditional
T → ~~the~~ arg 1
F → {3

X = does fn always exit?

not going to look at too closly

E should be empty when run SQL

Built in fns have summaries pre-computed

Prompts user on Reg Ex
it does not try to deal w/

Sandboxing JS

How do we isolate JS?

Ad banner on NYT.com

1: Run an interpeter

narcissus → JS interpeter written in JS

2: Lang Level Isolation

this paper

(25)

a) prohibit sensative names
   - document
   - window
   - eval()

Oh this was the FBJS one

b) rename variables

   a_12345_variable

c) prohibit custom @ values

d) for "this"

   instead $FBJS.ref(this)

   which checks if a sensative name

   otherwise returns

e) Objects

   use as a stack frame

f) prohibit with()

   So only scope objects rewritten

(26)

What are JS scope objects?

Scope is env that fn exists in

Looked up with()
pretty stupid
don't know if global objects getting hammered
is unnessary

Non scope objects accessed w/ .[ ]

if don't find an object in its instance,
follow the prototype link

obj from which other objects can inherit properties
any obj can be a prototype

So g) block . prototype
__proto__

h) trust some support routines
.toString()

i) exceptions for
e. parent
e. innerHTML
e. addElement()

but hard to get it to work cross browser

SSL

Network not safe

Symetric
Asymetric

Public key crypto
not really focusing on

Cookies don't really follow rules

28

Same origin issues

- Download not secure JS
  Runs in same context
  can call back w/ data

HSTS tell it to only connect over SSL
in browse header

EV certs extra verification

key Pining in Chrome say which CA will use
but bootstrapping issue

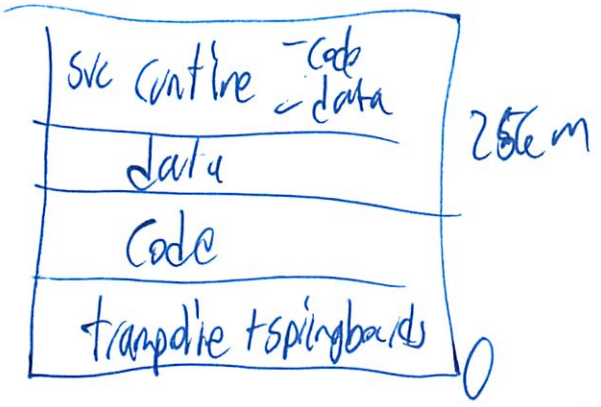
Native Client (NaCl)

by Google
to run native code directly in browser
Cross platform
IPC channel to trusted object

(29)

SFI rewrite code - recompile ← server
verify it ← client/browse

1. no disallowed instructions
   - sys calls
2. Restrict memory addresses



So can only access within 0 → 256m

x86 allows variable lenght instr

So can only jmp to start/end at 32 bit gaps

Some runtime instrumentation to tell

0 out the low order bits

Using AND $0xfffffff0, %eax

before JMP * %eax

(30)

Let and/or clears lower 5 bits
So must be verified instruction location

HALT data at end of block to prevent it from running over

0 out higher bits to prevent from jumping too high

Call lib w/ segmentation
Segment selector register
% ds
I don't get this

flags that only executable
not writable

actually faster

trampoline Sandbox → service runtime
Undo sandbox

Springboard svc runtime → Sandbox
reenables sandbox

User authentication

(last lecture!)

This was the one computing all the passwords

User ↔ computer ↔ verifier server

proposed a bunch of criteria

hash + salt
- to prevent rainbow tables

password managers

graphical authentication
- grid

one time password

single sign on

biometrics

tan

10/22 practice

Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.858 Fall 2011

Quiz I

You have 80 minutes to answer the questions in this quiz. In order to receive credit you must answer the question as precisely as possible.

Some questions are harder than others, and some questions earn more points than others. You may want to skim them all through first, and attack them in the order that allows you to make the most progress.

If you find a question ambiguous, be sure to write down any assumptions you make. Be neat and legible. If we can't understand your answer, we can't give you credit!

Write your name on this cover sheet.

THIS IS AN OPEN BOOK, OPEN NOTES EXAM.

Please do not write in the boxes below.

I (xx/20)	II (xx/10)	III (xx/16)	IV (xx/22)	V (xx/10)	VI (xx/16)	VII (xx/6)	Total (xx/100)

Name:

Username from handin site:

did not cover

I XFI

Consider the following assembly code, which zeroes out 256 bytes of memory pointed to by the EAX register. This code will execute under XFI. XFI's allocation stack is not used in this code.

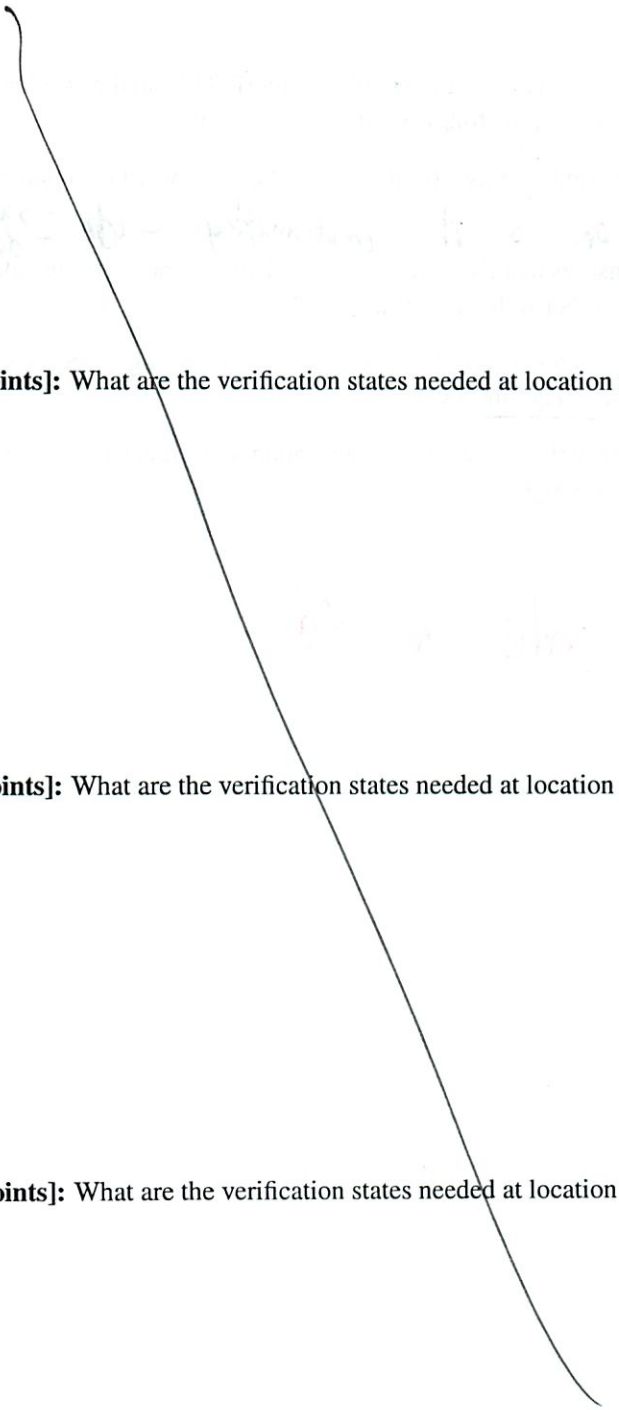
You will need fill in the verification states for this code, which would be required for the verifier to check the safety of this code, along the lines of the example shown in Figure 4 of the XFI paper. Following the example from the paper, possible verification state statements include:

```
valid[regname+const, regname+const)
origSSP = regname+const
retaddr = Mem[regname]
```

where `regname` and `const` are any register names and constant expressions, respectively. Include all verification states necessary to ensure safety of the subsequent instruction, and to ensure that the next verification state is legal.

```
1   x86 instructions          Verification state
2
3   mrguard(EAX, 0, 256)
4                             (1)
5   ECX := EAX       # current pointer
6   EDX := EAX+256   # end of 256-byte array
7
8 loop:
9                             (2)
10  Mem[ECX] := 0
11  ECX := ECX+4
12                            (3)
13  if ECX+4 > EDX, jmp out
14                            (4)
15  jmp loop
16
17 out:
18  ...
```

1. **[5 points]:** What are the verification states needed at location marked (1)?



2. **[5 points]:** What are the verification states needed at location marked (2)?

3. **[5 points]:** What are the verification states needed at location marked (3)?

4. **[5 points]:** What are the verification states needed at location marked (4)?

II ForceHTTPS

I knew depends on ans

5. [10 points]: Suppose `bank.com` uses and enables ForceHTTPS, and has a legitimate SSL certificate signed by Verisign. Which of the following statements are true?

A. True / False ForceHTTPS prevents the user from entering their password on a phishing web site impersonating `bank.com`.

how is it impersonating - URL - same URL - does it have a cert

B. True / False ForceHTTPS ensures that the developer of the `bank.com` web site cannot accidentally load Javascript code from another web server using `<SCRIPT SRC=...>`.

real →

C. True / False ForceHTTPS prevents a user from accidentally accepting an SSL certificate for `bank.com` that's not signed by any legitimate CA.

D. True / False ForceHTTPS prevents a browser from accepting an SSL certificate for `bank.com` that's signed by a CA other than Verisign.

only non SSL code blocked

III Zoobar security

Ben Bitdiddle is working on lab 2. For his privilege separation, he decided to create a separate database to store each user's zoobar balance (instead of a single database called `zoobars` that stores everyone's balance). He stores the zoobar balance for user `x` in the directory `/jail/zoobar/db/zoobars.x`, and ensures that usernames cannot contain slashes or null characters. When a user first registers, the login service must be able to create this database for the user, so Ben sets the permissions for `/jail/zoobar/db` to 0777.

6. [4 points]: Explain why this design may be a bad idea. Be specific about what an adversary would have to do to take advantage of a weakness in this design.

Other services on server could delete those (i but not modify)
Well delete + recreate

Ben Bitdiddle is now working on lab 3. He has three user IDs for running server-side code, as suggested in lab 2 (ignoring transfer logging):

- User ID 900 is used to run dynamic python code to handle HTTP requests (via zookfs). The database containing user profiles is writable only by uid 900.
- User ID 901 is used to run the authentication service, which provides an interface to obtain a token given a username and password, and to check if some token for a username is valid. The database containing user passwords and tokens is stored in a DB that is readable and writable only by uid 901.
- User ID 902 is used to run the transfer service, which provides an interface to transfer zoobar credits from user *A* to user *B*, as long as a token for user *A* is provided. The database storing zoobar balances is writable only by uid 902. The transfer service invokes the authentication service to check whether a token is valid.

Recall that to run Python profile code for user *A*, Ben must give the profile code access to *A*'s token (the profile code may want to transfer credits to visitors, and will need this token to invoke the transfer service).

To support Python profiles, Ben adds a new operation to the authentication service's interface, where the caller supplies an argument `username`, the authentication service looks up the profile for `username`, runs the profile's code with a token for `username`, and returns the output of that code.

7. [4 points]: Ben discovers that a bug in the HTTP handling code (running as uid 900) can allow an adversary to steal zoobars from any user. Explain how an adversary can do this in Ben's design.

Set their profile to transfer

have auth service call run profile

✓

8. [8 points]: Propose a design change that prevents attackers from stealing zoobars even if they compromise the HTTP handling code. Do not make any changes to the authentication or transfer services (i.e., code running as uid 901 and 902).

protect ✓ changing profiles

(write longer)

IV Baggy bounds checking

Consider a system that runs the following code under the Baggy bounds checking system, as described in the paper by Akritidis et al, with slot_size=16:

```
1  struct sa {
2    char buf[32];
3    void (*f) (void);
4  };
5
6  struct sb {
7    void (*f) (void);
8    char buf[32];
9  };
10
11 void handle(void) {
12   printf("Hello.\n");
13 }
14
15 void buggy(char *buf, void (**f) (void)) {
16   *f = handle;
17   gets(buf);
18   (*f) ();
19 }
20
21 void test1(void) {
22   struct sa x;
23   buggy(x.buf, &x.f);
24 }
25
26 void test2(void) {
27   struct sb x;
28   buggy(x.buf, &x.f);
29 }
30
31 void test3(void) {
32   struct sb y;
33   struct sa x;
34   buggy(x.buf, &y.f);
35 }
36
37 void test4(void) {
38   struct sb x[2];
39   buggy(x[0].buf, &x[1].f);
40 }
```

Assume the compiler performs no optimizations and places variables on the stack in the order declared, the stack grows down (from high address to low address), that this is a 32-bit system, and that the address of `handle` contains no zero bytes.

See I had no clue what the code did/does is running buggy

* = pointer
& = address of

9. [6 points]:

A. True / False If function `test1` is called, an adversary can construct an input that will cause the program to jump to an arbitrary address. if overflow x.buf into x.f - remain inside X

B. True / False If function `test2` is called, an adversary can construct an input that will cause the program to jump to an arbitrary address. if overflow x.buf into higher location, exceed bounds

C. True / False If function `test3` is called, an adversary can construct an input that will cause the program to jump to an arbitrary address. " " " " " " " "

D. True / False If function `test4` is called, an adversary can construct an input that will cause the program to jump to an arbitrary address. overflow x[0] into x[4] remain within X

I don't get diff

For the next four questions, determine what is the minimum number of bytes that an adversary has to provide as input to cause this program to likely crash, when running different test functions. Do not count the newline character that the adversary has to type in to signal the end of the line to `gets`. Recall that `gets` terminates its string with a zero byte.

10. [4 points]: What is the minimum number of bytes that an adversary has to provide as input to likely cause a program running `test1` to crash?

$\frac{4}{2} = 8$ 32 overwrite x.f w/ NUL byte + jump to it

11. [4 points]: What is the minimum number of bytes that an adversary has to provide as input to likely cause a program running `test2` to crash?

60 buggy exception

12. [4 points]: What is the minimum number of bytes that an adversary has to provide as input to likely cause a program running `test3` to crash?

64 " "

13. [4 points]: What is the minimum number of bytes that an adversary has to provide as input to likely cause a program running `test4` to crash?

32 overwriting x[0].f w/ NUL byte + jumping to it

V Browser security

The same origin policy generally does not apply to images or scripts. What this means is that a site may include images or scripts from any origin.

14. [3 points]: Explain why including images from other origins may be a bad idea for user privacy.

Can leak info in url
e.g. ~~get~~ username = Michael

15. [3 points]: Explain why including scripts from another origin can be a bad idea for security.

JS has a lot of power
Can suck info off pg + send to server
or set harmful cookies

16. [4 points]: In general, access to the file system by JavaScript is disallowed as part of JavaScript code sandboxing. Describe a situation where executing JavaScript code will lead to file writes.

e.g. cookie writing is saved to disk
I know FF saves open URLs to disk in case of crashes

images cached
this is a stupid qu

VI Static analysis

Consider the following snippet of JavaScript code:

```
1  var P = false;
2
3  function foo() {
4    var t1 = new Object();
5    var t2 = new Object();
6    var t = bar(t1, t2);
7    P = true;
8  }
9
10 function bar(x, y) {
11   var r = new Object();
12   if (P) {
13     r = x;
14   } else {
15     r = y;
16   }
17
18   return r;
19 }
```

A flow sensitive pointer analysis means that the analysis takes into account the order of statements in the program. A flow insensitive pointer analysis does not consider the order of statements.

17. [4 points]: Assuming no dead code elimination is done, a flow-insensitive pointer analysis (i.e., one which does not consider the control flow of a program) will conclude that variable `t` in function `foo` may point to objects allocated at the following line numbers:

A. True / False Line 1

B. True / False Line 4

C. True / False Line 5

D. True / False Line 11

18. [4 points]: Assuming no dead code elimination is done, a flow-sensitive pointer analysis (i.e., one which considers the control flow of a program) will conclude that variable `t` in function `foo` may point to objects allocated at the following line numbers:

A. True / False Line 1

B. True / False Line 4

C. True / False Line 5

D. True / False Line 11

oh calls bar()

19. [2 points]: At runtime, variable `t` in function `foo` may only be observed pointing to objects allocated at the following line numbers:

A. True / False Line 1

B. True / False Line 4

C. True / False Line 5

D. True / False Line 11

I don't get

It does not explain why

I think I see

I was ~~guessing~~ overthinking / not thinking at all

20. [2 points]: Do you think a sound analysis that supports the `eval` construct is going to be precise? Please explain.

No very difficult to restrict
Only if strictly blacklist
must restrict no arbitrary code at runtime

21. [4 points]: What is one practical advantage of the bottom-up analysis of the call graph described in the PHP paper by Xie and Aiken (discussed in class)?

Can see mistakes + fix them
if a lot of time to sort through errors

performance + scalability
Summarizing

it was more about the bottom up nature

VII 6.858

We'd like to hear your opinions about 6.858, so please answer the following questions. (Any answer, except no answer, will receive full credit.)

22. [2 points]: How could we make the ideas in the course easier to understand?

23. [2 points]: What is the best aspect of 6.858 so far?

24. [2 points]: What is the worst aspect of 6.858 so far?

End of Quiz

Department of Electrical Engineering and Computer Science

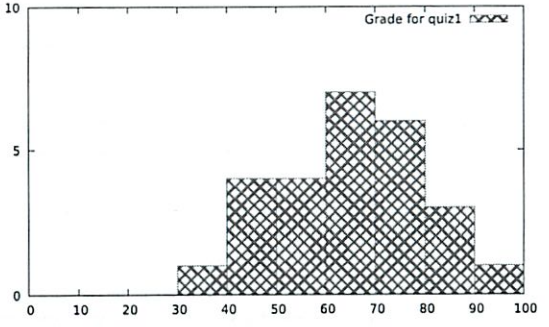
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.858 Fall 2011

Quiz I: Solutions

Please do not write in the boxes below.

I (xx/20)	II (xx/10)	III (xx/16)	IV (xx/22)	V (xx/10)	VI (xx/16)	VII (xx/6)	Total (xx/100)



I XFI

Consider the following assembly code, which zeroes out 256 bytes of memory pointed to by the EAX register. This code will execute under XFI. XFI's allocation stack is not used in this code.

You will need fill in the verification states for this code, which would be required for the verifier to check the safety of this code, along the lines of the example shown in Figure 4 of the XFI paper. Following the example from the paper, possible verification state statements include:

```
valid[regname+const, regname+const)
origSSP = regname+const
retaddr = Mem[regname]
```

where `regname` and `const` are any register names and constant expressions, respectively. Include all verification states necessary to ensure safety of the subsequent instruction, and to ensure that the next verification state is legal.

```
1   x86 instructions          Verification state
2
3   mrguard(EAX, 0, 256)
4                             (1)
5   ECX := EAX        # current pointer
6   EDX := EAX+256    # end of 256-byte array
7
8 loop:
9                             (2)
10  Mem[ECX] := 0
11  ECX := ECX+4
12                            (3)
13  if ECX+4 > EDX, jmp out
14                            (4)
15  jmp loop
16
17 out:
18  ...
```

1. [5 points]: What are the verification states needed at location marked (1)?

Answer:

- valid[EAX-0, EAX+256) is the only verification state that can be inferred at this point.

2. [5 points]: What are the verification states needed at location marked (2)?

Answer:

- valid[EAX-0, EAX+256), from above.
- valid[ECX-0, ECX+4), to satisfy the subsequent write to 4 bytes at ECX.
- valid[ECX-0, EDX+0), to represent the loop condition.

3. [5 points]: What are the verification states needed at location marked (3)?

Answer:

- valid[EAX-0, EAX+256), from above.
- valid[ECX-4, EDX+0), the loop condition updated with new value of ECX.

4. [5 points]: What are the verification states needed at location marked (4)?

Answer:

- valid[EAX-0, EAX+256), from above.
- valid[ECX-4, EDX+0), from above.
- valid[ECX-0, ECX+4), inferred from the check just before.

Note that these verification states must imply (i.e., be at least as strong) as the verification states at (2).

II ForceHTTPS

5. [10 points]: Suppose `bank.com` uses and enables ForceHTTPS, and has a legitimate SSL certificate signed by Verisign. Which of the following statements are true?

A. True / False ForceHTTPS prevents the user from entering their password on a phishing web site impersonating `bank.com`.

Answer: False.

B. True / False ForceHTTPS ensures that the developer of the `bank.com` web site cannot accidentally load Javascript code from another web server using `<SCRIPT SRC=...>`.

Answer: False.

C. True / False ForceHTTPS prevents a user from accidentally accepting an SSL certificate for `bank.com` that's not signed by any legitimate CA.

Answer: True.

D. True / False ForceHTTPS prevents a browser from accepting an SSL certificate for `bank.com` that's signed by a CA other than Verisign.

Answer: False.

III Zoobar security

Ben Bitdiddle is working on lab 2. For his privilege separation, he decided to create a separate database to store each user's zoobar balance (instead of a single database called `zoobars` that stores everyone's balance). He stores the zoobar balance for user `x` in the directory `/jail/zoobar/db/zoobars.x`, and ensures that usernames cannot contain slashes or null characters. When a user first registers, the login service must be able to create this database for the user, so Ben sets the permissions for `/jail/zoobar/db` to 0777.

6. [4 points]: Explain why this design may be a bad idea. Be specific about what an adversary would have to do to take advantage of a weakness in this design.

Answer: Since the directory is world-writable, an adversary could replace the contents of an arbitrary database, by first renaming the existing database's subdirectory to some unused name, and then creating a fresh directory (database) with the desired name of the database. For example, the adversary could replace all passwords with ones that the adversary chooses.

Answer: If an attacker can compromise any service, he can rename the `zoobars.x` file, since the directory is world-writable and not sticky, and replace it with a new one. (He can also replace the file with a symbolic link to an interesting other file that the zoobar-handling user can write to, and mount something along the lines of a confused-deputy attack.)

Full credit was also given for creating a directory before the user gets created; partial credit was given for removing a directory (since you cannot remove a non-empty directory you don't have permissions on).

Ben Bitdiddle is now working on lab 3. He has three user IDs for running server-side code, as suggested in lab 2 (ignoring transfer logging):

- User ID 900 is used to run dynamic python code to handle HTTP requests (via zookfs). The database containing user profiles is writable only by uid 900.
- User ID 901 is used to run the authentication service, which provides an interface to obtain a token given a username and password, and to check if some token for a username is valid. The database containing user passwords and tokens is stored in a DB that is readable and writable only by uid 901.
- User ID 902 is used to run the transfer service, which provides an interface to transfer zoobar credits from user A to user B, as long as a token for user A is provided. The database storing zoobar balances is writable only by uid 902. The transfer service invokes the authentication service to check whether a token is valid.

Recall that to run Python profile code for user A, Ben must give the profile code access to A's token (the profile code may want to transfer credits to visitors, and will need this token to invoke the transfer service).

To support Python profiles, Ben adds a new operation to the authentication service's interface, where the caller supplies an argument `username`, the authentication service looks up the profile for `username`, runs the profile's code with a token for `username`, and returns the output of that code.

7. [4 points]: Ben discovers that a bug in the HTTP handling code (running as uid 900) can allow an adversary to steal zoobars from any user. Explain how an adversary can do this in Ben's design.

Answer: An adversary can modify an arbitrary user's profile and inject Python code that will transfer all of the user's zoobars to the adversary's account.

8. [8 points]: Propose a design change that prevents attackers from stealing zoobars even if they compromise the HTTP handling code. Do not make any changes to the authentication or transfer services (i.e., code running as uid 901 and 902).

Answer: Use a separate service, running as a separate uid, to edit profiles. Make sure the profile database is writable only by this new service's uid. Require the user's token to be passed to this service when editing a user's profile. Have this profile-editing service check the token using the authentication service.

Note that this only prevents attacking users who never log in, as the HTTP service can get the token of any user who does log in. An argument that compromising the HTTP service gets you wide latitude in compromising any user's activity would have been accepted for full credit.

IV Baggy bounds checking

Consider a system that runs the following code under the Baggy bounds checking system, as described in the paper by Akritidis et al, with slot_size=16:

```
1  struct sa {
2    char buf[32];
3    void (*f) (void);
4  };
5
6  struct sb {
7    void (*f) (void);
8    char buf[32];
9  };
10
11 void handle(void) {
12   printf("Hello.\n");
13 }
14
15 void buggy(char *buf, void (**f) (void)) {
16   *f = handle;
17   gets(buf);
18   (*f) ();
19 }
20
21 void test1(void) {
22   struct sa x;
23   buggy(x.buf, &x.f);
24 }
25
26 void test2(void) {
27   struct sb x;
28   buggy(x.buf, &x.f);
29 }
30
31 void test3(void) {
32   struct sb y;
33   struct sa x;
34   buggy(x.buf, &y.f);
35 }
36
37 void test4(void) {
38   struct sb x[2];
39   buggy(x[0].buf, &x[1].f);
40 }
```

Assume the compiler performs no optimizations and places variables on the stack in the order declared, the stack grows down (from high address to low address), that this is a 32-bit system, and that the address of `handle` contains no zero bytes.

9. [6 points]:

A. True / False If function `test1` is called, an adversary can construct an input that will cause the program to jump to an arbitrary address.

Answer: True. (If you overflow x.buf into x.f, you remain within the allocation bounds of x.)

B. True / False If function `test2` is called, an adversary can construct an input that will cause the program to jump to an arbitrary address.

Answer: False. (If you overflow x.buf into any higher location, like the return pointer, you exceed the allocation bounds of x.)

C. True / False If function `test3` is called, an adversary can construct an input that will cause the program to jump to an arbitrary address.

Answer: False. (If you overflow x.buf into any higher location, like y, you exceed the allocation bounds of x.)

D. True / False If function `test4` is called, an adversary can construct an input that will cause the program to jump to an arbitrary address.

Answer: True. (If you overflow x[0] into x[1], you remain within the allocation bounds of the array x.)

For the next four questions, determine what is the minimum number of bytes that an adversary has to provide as input to cause this program to likely crash, when running different test functions. Do not count the newline character that the adversary has to type in to signal the end of the line to `gets`. Recall that `gets` terminates its string with a zero byte.

10. [4 points]: What is the minimum number of bytes that an adversary has to provide as input to likely cause a program running `test1` to crash?

Answer: 32 (by overwriting x.f with a NUL byte, and jumping to it). Overwriting 64 bytes would cause a baggy bounds exception, but you can crash the program earlier.

11. [4 points]: What is the minimum number of bytes that an adversary has to provide as input to likely cause a program running `test2` to crash?

Answer: 60 (via a baggy bounds exception).

12. [4 points]: What is the minimum number of bytes that an adversary has to provide as input to likely cause a program running `test3` to crash?

Answer: 64 (via a baggy bounds exception).

13. [4 points]: What is the minimum number of bytes that an adversary has to provide as input to likely cause a program running `test4` to crash?

Answer: 32 (by overwriting x[1].f with a NUL byte, and jumping to it). Overwriting 124 bytes would cause a baggy bounds exception, but you can crash the program earlier.

V Browser security

The same origin policy generally does not apply to images or scripts. What this means is that a site may include images or scripts from any origin.

14. [3 points]: Explain why including images from other origins may be a bad idea for user privacy.

Answer: The other origin's server can track visitors to the page embedding images from that server.

15. [3 points]: Explain why including scripts from another origin can be a bad idea for security.

Answer: The other origin's server must be completely trusted, since the script runs with the privileges of the embedding page. For example, the script's code can access and manipulate the DOM of the embedding page, or access and send out the cookies from the embedding page.

16. [4 points]: In general, access to the file system by JavaScript is disallowed as part of JavaScript code sandboxing. Describe a situation where executing JavaScript code will lead to file writes.

Answer: Setting a cookie in Javascript typically leads to a file write, since the browser usually stores cookies persistently. Loading images can cause the image content to be saved in the cache (in some local file).

VI Static analysis

Consider the following snippet of JavaScript code:

```
1  var P = false;
2
3  function foo() {
4    var t1 = new Object();
5    var t2 = new Object();
6    var t = bar(t1, t2);
7    P = true;
8  }
9
10 function bar(x, y) {
11   var r = new Object();
12   if (P) {
13     r = x;
14   } else {
15     r = y;
16   }
17
18   return r;
19 }
```

A flow sensitive pointer analysis means that the analysis takes into account the order of statements in the program. A flow insensitive pointer analysis does not consider the order of statements.

17. [4 points]: Assuming no dead code elimination is done, a flow-insensitive pointer analysis (i.e., one which does not consider the control flow of a program) will conclude that variable `t` in function `foo` may point to objects allocated at the following line numbers:

A. True / False Line 1

Answer: False.

B. True / False Line 4

Answer: True.

C. True / False Line 5

Answer: True.

D. True / False Line 11

Answer: True.

18. [4 points]: Assuming no dead code elimination is done, a flow-sensitive pointer analysis (i.e., one which considers the control flow of a program) will conclude that variable `t` in function `foo` may point to objects allocated at the following line numbers:

A. True / False Line 1

Answer: False.

B. True / False Line 4

Answer: True.

C. True / False Line 5

Answer: True.

D. True / False Line 11

Answer: False.

19. [2 points]: At runtime, variable `t` in function `foo` may only be observed pointing to objects allocated at the following line numbers:

A. True / False Line 1

Answer: False.

B. True / False Line 4

Answer: True.

C. True / False Line 5

Answer: True.

D. True / False Line 11

Answer: False.

20. [2 points]: Do you think a sound analysis that supports the `eval` construct is going to be precise? Please explain.

Answer: No, because it is difficult to statically reason about the code that may be executed at runtime when eval is invoked, unless the analysis can prove that arbitrary code cannot be passed to eval at runtime, and can statically analyze all possible code strings that can be passed to eval.

21. [4 points]: What is one practical advantage of the bottom-up analysis of the call graph described in the PHP paper by Xie and Aiken (discussed in class)?

Answer: Performance and scalability, by not analyzing functions that are not invoked by application code, and by summarizing the effects of the function once and reusing that information for inter-procedural analysis.

VII 6.858

We'd like to hear your opinions about 6.858, so please answer the following questions. (Any answer, except no answer, will receive full credit.)

22. [2 points]: How could we make the ideas in the course easier to understand?

Answer: Any answer received full credit.

23. [2 points]: What is the best aspect of 6.858 so far?

Answer: Any answer received full credit.

24. [2 points]: What is the worst aspect of 6.858 so far?

Answer: Any answer received full credit.

End of Quiz

Ted

function pointer

Ans w/ bounds checking on!

in bytes

void is ~~decreted~~ fn

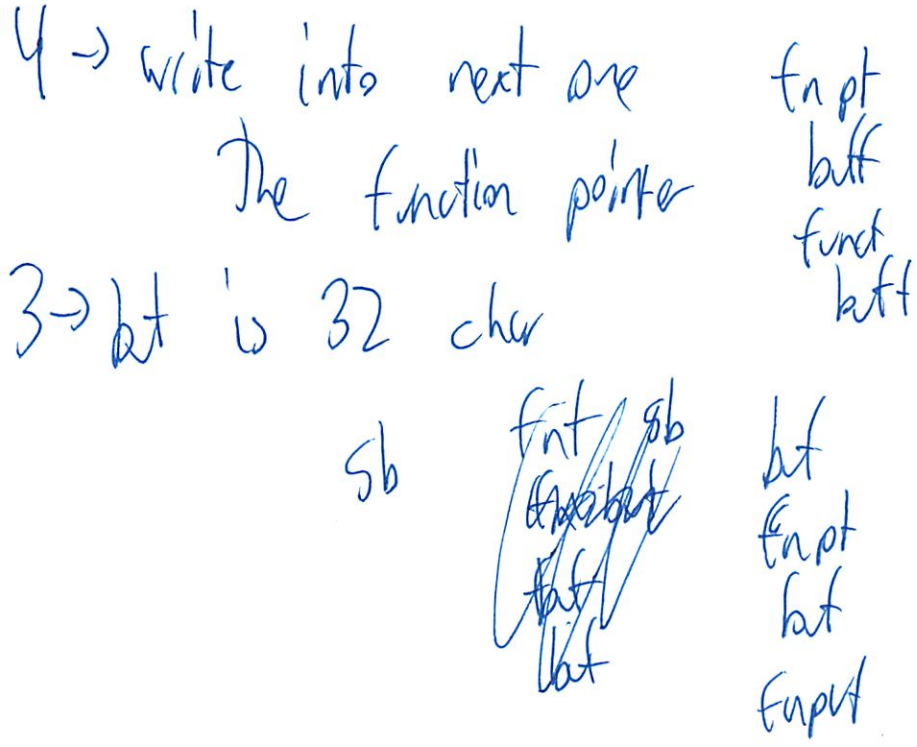
late decreted + called

That is the arbitrary execution!

4 → write into next one

The function pointer

3 → bt is 32 char



flow-insensitive → read def'n

Changes P false later

Can be false or true

So can be either x, y

Can be either x, y or R

One

overwritten no matter what

Data-flow analysis

From Wikipedia, the free encyclopedia

Data-flow analysis is a technique for gathering information about the possible set of values calculated at various points in a computer program. A program's control flow graph (CFG) is used to determine those parts of a program to which a particular value assigned to a variable might propagate. The information gathered is often used by compilers when optimizing a program. A canonical example of a data-flow analysis is reaching definitions.

A simple way to perform data-flow analysis of programs is to set up data-flow equations for each node of the control flow graph and solve them by repeatedly calculating the output from the input locally at each node until the whole system stabilizes, i.e., it reaches a fixpoint. This general approach was developed by Gary Kildall while teaching at the Naval Postgraduate School.[1]

Contents

Basic principles

It is the process of collecting information about the way the variables are used, defined in the program. Data-flow analysis attempts to obtain particular information at each point in a procedure. Usually, it is enough to obtain this information at the boundaries of basic blocks, since from that it is easy to compute the information at points in the basic block. In forward flow analysis, the exit state of a block is a function of the block's entry state. This function is the composition of the effects of the statements in the block. The entry state of a block is a function of the exit states of its predecessors. This yields a set of data-flow equations:

For each block b:

$$out_b = trans_b(in_b)$$

$$in_b = join_{p \in pred_b}(out_p)$$

In this, $trans_b$ is the **transfer function** of the block b. It works on the entry state in_b, yielding the exit state out_b. The join operation $join$ combines the exit states of the predecessors $p \in pred_b$ of b, yielding the entry state of b.

After solving this set of equations, the entry and / or exit states of the blocks can be used to derive properties of the program at the block boundaries. The transfer function of each statement separately can be applied to get information at a point inside a basic block.

Each particular type of data-flow analysis has its own specific transfer function and join operation. Some data-flow problems require backward flow analysis. This follows the same plan, except that the transfer function is applied to the exit state yielding the entry state, and the join operation works on the entry states of the successors to yield the exit state.

The entry point (in forward flow) plays an important role: Since it has no predecessors, its entry state is well defined at the start of the analysis. For instance, the set of local variables with known values is empty. If the control flow graph does not contain cycles (there were no explicit or implicit loops in the procedure) solving the equations is straightforward. The control flow graph can then be topologically sorted; running in the order of this sort, the entry states can be computed at the start of each block, since all predecessors of that block have already been processed, so their exit states are available. If the control flow graph does contain cycles, a more advanced algorithm is required.

An iterative algorithm

The most common way of solving the data-flow equations is by using an iterative algorithm. It starts with an approximation of the in-state of each block. The out-states are then computed by applying the transfer functions on the in-states. From these, the in-states are updated by applying the join operations. The latter two steps are repeated until we reach the so-called **fixpoint**: the situation in which the in-states (and the out-states in consequence) do not change.

A basic algorithm for solving data-flow equations is the **round-robin iterative algorithm**:

```
for i ← 1 to N
        initialize node i
while (sets are still changing)
        for i ← 1 to N
                recompute sets at node i
```

Convergence

To be usable, the iterative approach should actually reach a fixpoint. This can be guaranteed by imposing constraints on the combination of the value domain of the states, the transfer functions and the join operation.

The value domain should be a partial order with **finite height** (i.e., there are no infinite ascending chains $x_1 < x_2 < ...$). The combination of the transfer function and the join operation should be monotonic with respect to this partial order. Monotonicity ensures that on each iteration the value will either stay the same or will grow larger, while finite height ensures that it cannot grow indefinitely. Thus we will ultimately reach

a situation where $T(x) = x$ for all x, which is the fixpoint.

The work list approach

It is easy to improve on the algorithm above by noticing that the in-state of a block will not change if the out-states of its predecessors don't change. Therefore, we introduce a **work list**: a list of blocks that still need to be processed. Whenever the out-state of a block changes, we add its successors to the work list. In each iteration, a block is removed from the work list. Its out-state is computed. If the out-state changed, the block's successors are added to the work list. For efficiency, a block should not be in the work list more than once.

The algorithm is started by putting information generating blocks in the work list. It terminates when the work list is empty.

The order matters

The efficiency of iteratively solving data-flow equations is influenced by the order at which local nodes are visited.[2] Furthermore, it depends, whether the data-flow equations are used for forward or backward data-flow analysis over the CFG. Intuitively, in a forward flow problem, it would be fastest if all predecessors of a block have been processed before the block itself, since then the iteration will use the latest information. In the absence of loops it is possible to order the blocks in such a way that the correct out-states are computed by processing each block only once.

In the following, a few iteration orders for solving data-flow equations are discussed (a related concept to iteration order of a CFG is tree traversal of a tree).

- **Random order** - This iteration order is not aware whether the data-flow equations solve a forward or backward data-flow problem. Therefore, the performance is relatively poor compared to specialized iteration orders.
- **Postorder** - This is a typical iteration order for backward data-flow problems. In *postorder iteration*, a node is visited after all its successor nodes have been visited. Typically, the *postorder iteration* is implemented with the **depth-first** strategy.
- **Reverse postorder** - This is a typical iteration order for forward data-flow problems. In **reverse-postorder iteration**, a node is visited before all its successor nodes have been visited, except when the successor is reached by a back edge. (Note that this is not the same as preorder.)

Initialization

The initial value of the in-states is important to obtain correct and accurate results. If the results are used for compiler optimizations, they should provide **conservative** information, i.e. when applying the information, the program should not change semantics. The iteration of the fixpoint algorithm will take the values in the direction of the maximum element. Initializing all blocks with the maximum element is therefore not useful. At least one block starts in a state with a value less than the maximum. The details depend on the data-flow problem. If the minimum element represents totally conservative information, the results can be used safely even during the data-flow iteration. If it represents the most accurate information, fixpoint should be reached before the results can be applied.

Examples

The following are examples of properties of computer programs that can be calculated by data-flow analysis. Note that the properties calculated by data-flow analysis are typically only approximations of the real properties. This is because data-flow analysis operates on the syntactical structure of the CFG without simulating the exact control flow of the program. However, to be still useful in practice, a data-flow analysis algorithm is typically designed to calculate an upper respectively lower approximation of the real program properties.

Forward Analysis

The reaching definition analysis calculates for each program point the set of definitions that may potentially reach this program point.

```
1: if b==4 then
2:     a = 5;
3: else
4:     a = 3;
5: endif
6:
7: if  a < 4 then
8:     ...
```

The reaching definition of variable "a" at line 7 is the set of assignments a=5 at line 2 and a=3 at line 4.

Backward Analysis

The live variable analysis calculates for each program point the variables that may be potentially read afterwards before their next write update. The result is typically used by dead code elimination to remove statements that assign to a variable whose value is not used afterwards.

The out-state of a block is the set of variables that are live at the end of the block. Its in-state is the set of variable that is live at the start of it. The out-state is the union of the in-states of the blocks successors. The transfer function of a statement is applied by making the variables that are written dead, then making the variables that are read live.

Initial Code:

```
b1: a = 3;
    b = 5;
    d = 4;
    if a > b then
b2:     c = a + b;
        d = 2;
b3: endif
    c = 4;
    return b * d + c;
```

Backward Analysis:

```
// in: {}
b1: a = 3;
    b = 5;
    d = 4;
    x = 100; //x is never being used later thus not in the out set {a,b,d}
    if a > b then
// out: {a,b,d}    //union of all (in) successors of b1 => b2: {a,b}, and b3:{b,d}

// in: {a,b}
b2: c = a + b;
    d = 2;
// out: {b,d}

// in: {b,d}
b3: endif
    c = 4;
    return b * d + c;
// out:{}
```

The out-state of b3 only contains *b* and *d*, since *c* has been written. The in-state of b1 is the union of the out-states of b2 and b3. The definition of *c* in b2 can be removed, since *c* is not live immediately after the statement.

Solving the data-flow equations starts with initializing all in-states and out-states to the empty set. The work list is initialized by inserting the exit point (b3) in the work list (typical for backward flow). Its computed out-state differs from the previous one, so its predecessors b1 and b2 are inserted and the process continues. The progress is summarized in the table below.

processing	in-state	old out-state	new out-state	work list
b3	{}	{}	{b,d}	(b1,b2)
b1	{b,d}	{}	{}	(b2)
b2	{b,d}	{}	{a,b}	(b1)
b1	{a,b,d}	{}	{}	()

Note that b1 was entered in the list before b2, which forced processing b1 twice (b1 was re-entered as predecessor of b2). Inserting b2 before b1 would have allowed earlier completion.

Initializing with the empty set is an optimistic initialization: all variables start out as dead. Note that the out-states cannot shrink from one iteration to the next, although the out-state can be smaller that the in-state. This can be seen from the fact that after the first iteration the out-state can only change by a change of the in-state. Since the in-state starts as the empty set, it can only grow in further iterations.

Other approaches

In 2002, Markus Mohnen described a new method of data-flow analysis that does not require the explicit construction of a data-flow graph,[3] instead relying on abstract interpretation of the program and keeping a working set of program counters. At each conditional branch, both targets are added to the working set. Each path is followed for as many instructions as possible (until end of program or until it has looped with no changes), and then removed from the set and the next program counter retrieved.

Bit vector problems

The examples above are problems in which the data-flow value is a set, e.g. the set of reaching definitions (Using a bit for a definition position in the program), or the set of live variables. These sets can be represented efficiently as **bit vectors**, in which each bit represents set membership of one particular element. Using this representation, the join and transfer functions can be implemented as bitwise logical operations. The join operation is typically union or intersection, implemented by bitwise *logical or* and *logical and*. The transfer function for each block can be decomposed in so-called *gen* and *kill* sets.

As an example, in live-variable analysis, the join operation is union. The *kill* set is the set of variables that are written in a block, whereas the *gen* set is the set of variables that are read without being written first. The data-flow equations become

$$out_b = \bigcup_{s \in succ_b} in_s$$

$$in_b = (out_b - kill_b) \cup gen_b$$

In logical operations, this reads as

```
out(b) = 0
for s in succ(b)
        out(b) = out(b) or in(s)
in(b) = (out(b) and not kill(b)) or gen(b)
```

Sensitivities

Data-flow analysis is inherently flow-sensitive. Data-flow analysis is typically path-insensitive, though it is possible to define data-flow equations that yield a path-sensitive analysis.

- A **flow-sensitive** analysis takes into account the order of statements in a program. For example, a flow-insensitive pointer alias analysis may determine "variables *x* and *y* may refer to the same location", while a flow-sensitive analysis may determine "after statement 20, variables *x* and *y* may refer to the same location".
- A **path-sensitive** analysis computes different pieces of analysis information dependent on the predicates at conditional branch instructions. For instance, if a branch contains a condition *x>0*, then on the *fall-through* path, the analysis would assume that *x<=0* and on the target of the branch it would assume that indeed *x>0* holds.
- A **context-sensitive** analysis is an *interprocedural* analysis that considers the calling context when analyzing the target of a function call. In particular, using context information one can *jump back* to the original call site, whereas without that information, the analysis information has to be propagated back to all possible call sites, potentially losing precision.

List of data-flow analyses

- Reaching definitions
- Liveness analysis
- Definite assignment analysis

Do 10/22 prof'up

Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.858 Fall 2010

Quiz I

All problems are open-ended questions. In order to receive credit you must answer the question as precisely as possible. You have 80 minutes to finish this quiz.

Write your name on this cover sheet.

Some questions may be harder than others. Read them all through first and attack them in the order that allows you to make the most progress. If you find a question ambiguous, be sure to write down any assumptions you make. Be neat. If we can't understand your answer, we can't give you credit!

THIS IS AN OPEN BOOK, OPEN NOTES EXAM.

Please do not write in the boxes below.

I (xx/16)	II (xx/15)	III (xx/17)	IV (xx/10)	V (xx/16)	VI (xx/6)	Total (xx/80)

Name:

& = bit AND
&& = boolean AND

I Baggy bounds checking

Suppose that you use Baggy bounds checking to run the following C code, where X and Y are constant values. Assume that *slot_size* is 16 bytes, as in the paper.

```
1  char *p = malloc(40);
2  char *q = p + X;
3  char *r = q + Y;
4  *r = '\0';
```

Easier

For the following values of X and Y, indicate which line number will cause Baggy checking to abort, or NONE if the program will finish executing without aborting.

1. **[2 points]:** $X = 45, Y = 0$

   None - within 64 limit

2. **[2 points]:** $X = 60, Y = 0$

   None

3. **[2 points]:** $X = 50, Y = -20$

   None

4. **[2 points]:** $X = 70, Y = -20$

   None q goes out, but less than $\frac{64}{2}$

5. **[2 points]:** $X = 80, Y = -20$

   Line 4, ret is 1 byte before obj start

6. **[2 points]:** $X = -5, Y = 4$

   Goes out by more than $\frac{64}{2}$

7. **[2 points]:** $X = -5, Y = 60$

   None, within

8. **[2 points]:** $X = -10, Y = 20$

   Out more $\frac{64}{2}$ in ← dir

What are we storing in the slots?

6 ← $\log_2(64)$

$\frac{1024}{16}$ base ← look up slot

don't get is this multiple slots?

∴ So would it be 16 INT blank

1024/16 since this is 64

II Control hijacking

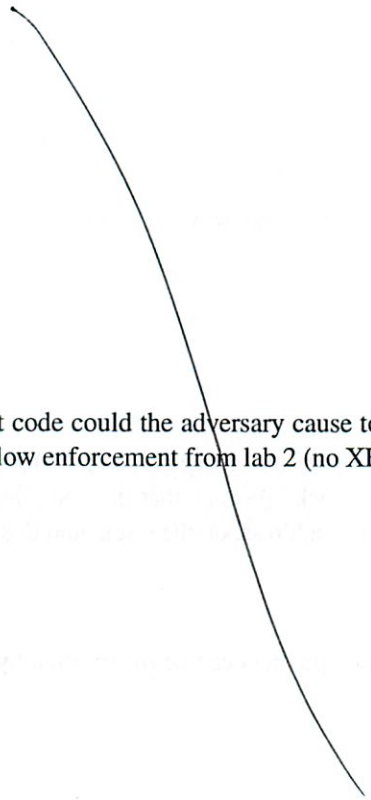
Consider the following C code:

```
struct foo {
    char buf[40];
    void (*f2) (struct foo *);
};

void
f(void)
{
    void (*f1) (struct foo *);
    struct foo x;

    /* .. initialize f1 and x.f2 in some way .. */

    gets(x.buf);
    if (f1)   f1(&x);
    if (x.f2) x.f2(&x);
}
```

There are three possible code pointers that may be overwritten by the buffer overflow vulnerability: $f1$, $x.f2$, and the function's return address on the stack. Assume that the compiler typically places the return address, $f1$, and x in that order, from high to low address, on the stack, and that the stack grows down.

9. [5 points]: Which of the three code pointers can be overwritten by an adversary if the code is executed as part of an XFI module?

10. [5 points]: What code could the adversary cause to be executed, if any, if the above code is executed as part of an XFI module?



11. [5 points]: What code could the adversary cause to be executed, if any, if the above code is executed under control-flow enforcement from lab 2 (no XFI)?

III OS protection

Ben Bitdiddle is running a web site using OKWS, with one machine running the OKWS server, and a separate machine running the database and the database proxy.

12. [12 points]: The database machine is maintained by another administrator, and Ben cannot change the 20-byte authentication tokens that are used to authenticate each service to the database proxy. This makes Ben worried that, if an adversary steals a token through a compromised or malicious service, Ben will not be able to prevent the adversary from accessing the database at a later time.

Propose a change to the OKWS design that would avoid giving tokens to each service, while providing the same guarantees in terms of what database operations each service can perform, without making any changes to the database machine.

13. [5 points]: Ben is considering running a large number of services under OKWS, and is worried he might run out of UIDs. To this end, Ben considers changing OKWS to use the same UID for several services, but to isolate them from each other by placing them in separate `chroot` directories (instead of the current OKWS design, which uses different UIDs but the same `chroot` directory). Explain, specifically, how an adversary that compromises one service can gain additional privileges under Ben's design that he or she cannot gain under the original OKWS design.

IV Capabilities and C

Ben Bitdiddle is worried that a plugin in his web browser could be compromised, and decides to apply some ideas from the "Security Architectures for Java" paper to sandboxing the plugin's C code using XFI.

Ben decides to use the capability model (§3.2 from the Java paper), and writes a function `safe_open` as follows:

```
int
safe_open(const char *pathname, int flags, mode_t mode)
{
    char buf[1024];
    snprintf(buf, sizeof(buf), "/safe-dir/%s", pathname);
    return open(buf, flags, mode);
}
```

which is intended to mirror Figure 2 from the Java paper. To allow his plugin's XFI module to access to files in `/safe-dir`, Ben allows the XFI module to call the `safe_open` function, as well as the standard `read`, `write`, and `close` functions (which directly invoke the corresponding system calls).

14. [10 points]: Can a malicious XFI module access files (i.e., read or write) outside of `/safe-dir`? As in the Java paper, let's ignore symbolic links and ".." components in the path name. Explain how or argue why not.

V Browser security

15. [6 points]: In pages of a site which has enabled ForceHTTPS, `<SCRIPT SRC=...>` tags that load code from an `http://.../` URL are redirected to an `https://.../` URL. Explain what could go wrong if this rewriting was not performed.

load unsave code

Ben Bitdiddle runs a web site that frequently adds and removes files, which leads to customers complaining that old links often return a 404 File not found error. Ben decides to fix this problem by adding a link to his site's search page, and modifies how his web server responds to requests for missing files, as follows (in Python syntax):

```
def missing_file(reqpath):
    print "HTTP/1.0 200 OK"
    print "Content-Type: text/html"
    print ""
    print "We are sorry, but the server could not locate file", reqpath
    print "Try using the <A HREF=/search>search function</A>."
```

16. [10 points]: Explain how an adversary may be able to exploit Ben's helpful error message to compromise the security of Ben's web application.

XSS!

VI 6.858

We'd like to hear your opinions about 6.858, so please answer the following questions. (Any answer, except no answer, will receive full credit.)

17. [2 points]: How could we make the ideas in the course easier to understand?

18. [2 points]: What is the best aspect of 6.858?

19. [2 points]: What is the worst aspect of 6.858?

End of Quiz

Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.858 Fall 2010

Quiz I

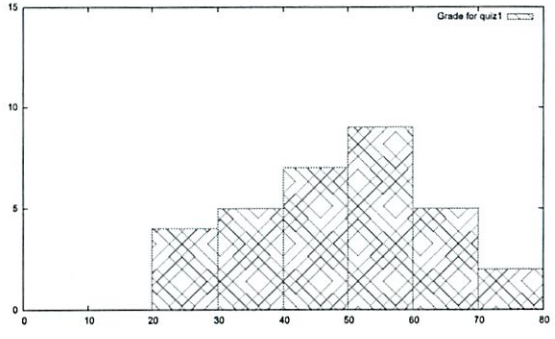
All problems are open-ended questions. In order to receive credit you must answer the question as precisely as possible. You have 80 minutes to finish this quiz.

Write your name on this cover sheet.

Some questions may be harder than others. Read them all through first and attack them in the order that allows you to make the most progress. If you find a question ambiguous, be sure to write down any assumptions you make. Be neat. If we can't understand your answer, we can't give you credit!

THIS IS AN OPEN BOOK, OPEN NOTES EXAM.

Please do not write in the boxes below.



Mean 49, Median 51, Std. dev. 14

I Baggy bounds checking

Suppose that you use Baggy bounds checking to run the following C code, where X and Y are constant values. Assume that *slot_size* is 16 bytes, as in the paper.

```
1  char *p = malloc(40);
2  char *q = p + X;
3  char *r = q + Y;
4  *r = '\0';
```

For the following values of X and Y, indicate which line number will cause Baggy checking to abort, or NONE if the program will finish executing without aborting.

Answer: Recall that Baggy bounds checking rounds up the allocation size to the nearest power of 2 (in this case, 64 for pointer p), and can track out-of-bounds pointers that are up *slot_size*/2 out of bounds.

1. [2 points]: $X = 45, Y = 0$
Answer: NONE: the access is within the 64 bytes limit.

2. [2 points]: $X = 60, Y = 0$
Answer: NONE: the access is within the 64 bytes limit.

3. [2 points]: $X = 50, Y = -20$
Answer: NONE: the access is within the 64 bytes limit.

4. [2 points]: $X = 70, Y = -20$
Answer: NONE: q goes out of bounds, but by less than *slot_size*/2, so r is in bounds again.

5. [2 points]: $X = 80, Y = -20$
Answer: Line 2: since q goes out of bounds by more than *slot_size*/2, Baggy aborts the pointer arithmetic. (We also accepted the answer of Line 4, due to some confusion.)

6. [2 points]: $X = -5, Y = 4$
Answer: Line 4: the reference is 1 byte before the start of the object, i.e. out of bounds.

7. [2 points]: $X = -5, Y = 60$
Answer: NONE: within the 64-byte object bounds.

8. [2 points]: $X = -10, Y = 20$
Answer: Line 2: since q goes out of bounds by more than *slot_size*/2, in the negative direction. (We also accepted the answer of Line 4, due to some confusion.)

II Control hijacking

Consider the following C code:

```
struct foo {
    char buf[40];
    void (*f2) (struct foo *);
};

void
f(void)
{
    void (*f1) (struct foo *);
    struct foo x;

    /* .. initialize f1 and x.f2 in some way .. */

    gets(x.buf);
    if (f1)   f1(&x);
    if (x.f2) x.f2(&x);
}
```

There are three possible code pointers that may be overwritten by the buffer overflow vulnerability: $f1$, $x.f2$, and the function's return address on the stack. Assume that the compiler typically places the return address, $f1$, and x in that order, from high to low address, on the stack, and that the stack grows down.

9. [5 points]: Which of the three code pointers can be overwritten by an adversary if the code is executed as part of an XFI module?

Answer: $x.f2$ can be overwritten, because it lives at a higher address than $x.buf$ on the allocation stack. $f1$ and the return address live on the scoped stack, which cannot be written to via pointers.

10. [5 points]: What code could the adversary cause to be executed, if any, if the above code is executed as part of an XFI module?

Answer: Any function inside the XFI module that is the target of indirect jumps (i.e., has an XFI label), and any stubs for allowed external functions (which also have XFI labels). The adversary cannot jump to arbitrary functions inside the XFI module that are not the targets of indirect jumps (and thus do not have an XFI label).

11. [5 points]: What code could the adversary cause to be executed, if any, if the above code is executed under control-flow enforcement from lab 2 (no XFI)?

Answer: Any code that was jumped to during the training run at the calls to $f1$ or $x.f2$, or any call sites of this function f during the training run.

III OS protection

Ben Bitdiddle is running a web site using OKWS, with one machine running the OKWS server, and a separate machine running the database and the database proxy.

12. [12 points]: The database machine is maintained by another administrator, and Ben cannot change the 20-byte authentication tokens that are used to authenticate each service to the database proxy. This makes Ben worried that, if an adversary steals a token through a compromised or malicious service, Ben will not be able to prevent the adversary from accessing the database at a later time.

Propose a change to the OKWS design that would avoid giving tokens to each service, while providing the same guarantees in terms of what database operations each service can perform, without making any changes to the database machine.

Answer 1: Implement a second proxy on the OKWS machine that keeps the real database tokens, accepts queries from services (authenticating the service using UIDs or another token), and forwards the queries to the real database server / proxy.

Answer 2: Establish connections to the database proxies in the launcher, send the 20-byte token from the launcher, and then pass the (now authenticated) file descriptors to the services.

13. [5 points]: Ben is considering running a large number of services under OKWS, and is worried he might run out of UIDs. To this end, Ben considers changing OKWS to use the same UID for several services, but to isolate them from each other by placing them in separate `chroot` directories (instead of the current OKWS design, which uses different UIDs but the same `chroot` directory). Explain, specifically, how an adversary that compromises one service can gain additional privileges under Ben's design that he or she cannot gain under the original OKWS design.

Answer: The compromised service could use `kill` or `ptrace` to interfere with or take over other services running under the same UID.

IV Capabilities and C

Ben Bitdiddle is worried that a plugin in his web browser could be compromised, and decides to apply some ideas from the "Security Architectures for Java" paper to sandboxing the plugin's C code using XFI.

Ben decides to use the capability model (§3.2 from the Java paper), and writes a function `safe_open` as follows:

```
int
safe_open(const char *pathname, int flags, mode_t mode)
{
    char buf[1024];
    snprintf(buf, sizeof(buf), "/safe-dir/%s", pathname);
    return open(buf, flags, mode);
}
```

which is intended to mirror Figure 2 from the Java paper. To allow his plugin's XFI module to access to files in `/safe-dir`, Ben allows the XFI module to call the `safe_open` function, as well as the standard `read`, `write`, and `close` functions (which directly invoke the corresponding system calls).

14. [10 points]: Can a malicious XFI module access files (i.e., read or write) outside of `/safe-dir`? As in the Java paper, let's ignore symbolic links and ".." components in the path name. Explain how or argue why not.

Answer: No. There are two possible attacks. First, a malicious XFI module could guess legitimate integer file descriptor numbers of other open files in the browser process (e.g., cookie files or cache files), and invoke `read` or `write` on them. Second, a malicious XFI module could write arbitrary data D to address A by first writing D to a file in `/safe-dir`, and then invoking `read` on that file, passing A as the buffer argument to `read`. This will write D to memory location A (since `read` is outside of XFI), and allow the attacker to gain control of the entire process.

V Browser security

15. [6 points]: In pages of a site which has enabled ForceHTTPS, `<SCRIPT SRC=...>` tags that load code from an `http://.../` URL are redirected to an `https://.../` URL. Explain what could go wrong if this rewriting was not performed.

Answer: An active attacker could replace the Javascript code in the HTTP response with arbitrary malicious code that could modify the containing HTTPS page or steal any of the data in that page, or the cookie for the HTTPS page's origin.

Ben Bitdiddle runs a web site that frequently adds and removes files, which leads to customers complaining that old links often return a 404 File not found error. Ben decides to fix this problem by adding a link to his site's search page, and modifies how his web server responds to requests for missing files, as follows (in Python syntax):

```
def missing_file(reqpath):
    print "HTTP/1.0 200 OK"
    print "Content-Type: text/html"
    print ""
    print "We are sorry, but the server could not locate file", reqpath
    print "Try using the <A HREF=/search>search function</A>."
```

16. [10 points]: Explain how an adversary may be able to exploit Ben's helpful error message to compromise the security of Ben's web application.

Answer: An adversary could construct a link such as:

```
http://ben.com/<SCRIPT>alert(5);</SCRIPT>,
```

containing arbitrary Javascript code, and trick legitimate users into visiting that link (e.g., by purchasing ads on some popular site). Ben's server would echo the request path back verbatim, including the Javascript code, causing the victim's browser to execute the resulting Javascript as part of Ben's page, giving the attacker's Javascript code access to the victim's cookies for Ben's site.

VI 6.858

We'd like to hear your opinions about 6.858, so please answer the following questions. (Any answer, except no answer, will receive full credit.)

17. [2 points]: How could we make the ideas in the course easier to understand?

18. [2 points]: What is the best aspect of 6.858?

19. [2 points]: What is the worst aspect of 6.858?

End of Quiz

Pratice 10/22

Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.893 Fall 2009

Quiz I

All problems are open-ended questions. In order to receive credit you must answer the question as precisely as possible. You have 80 minutes to finish this quiz.

Write your name on this cover sheet.

Some questions may be harder than others. Read them all through first and attack them in the order that allows you to make the most progress. If you find a question ambiguous, be sure to write down any assumptions you make. Be neat. If we can't understand your answer, we can't give you credit!

THIS IS AN OPEN BOOK, OPEN NOTES EXAM.

Please do not write in the boxes below.

I (xx/18)	II (xx/16)	III (xx/18)	IV (xx/8)	V (xx/18)
VI (xx/8)	**VII (xx/8)**	**VIII (xx/6)**	**Total (xx/100)**	

Name:

I Buffer Overflows

Ben Bitdiddle is building a web server that runs the following code sequence, in which `process_req()` is invoked with a user-supplied string of arbitrary length. Assume that `process_get()` is safe, and for the purposes of this question, simply returns right away.

```
void process_req(char *input) {
    char buf[256];
    strcpy(buf, input);
    if (!strncmp(buf, "GET ", 4))
        process_get(buf);
    return;
}
```

1. [6 points]: Ben Bitdiddle wants to prevent attackers from exploiting bugs in his server, so he decides to make the stack memory non-executable. Explain how an attacker can still exploit a buffer overflow in his code to delete files on the server. Draw a stack diagram to show what locations on the stack you need to control, what values you propose to write there, and where in the input string these values need to be located.

2. [6 points]: Seeing the difficulty of preventing exploits with a non-executable stack, Ben instead decides to make the stack grow up (towards larger addresses), instead of down like on the x86. Explain how you could exploit `process_req()` to execute arbitrary code. Draw a stack diagram to illustrate what locations on the stack you plan to corrupt, and where in the input string you would need to place the desired values.

3. [6 points]: Consider the StackGuard system from the "Buffer Overflows" paper in the context of Ben's new system where the stack grows *up*. Explain where on the stack the canary should be placed, at what points in the code the canary should be written, and at what points it should be checked, to prevent buffer overflow exploits that take control of the return address.

II XFI

4. [2 points]: Suppose a program has a traditional buffer overflow vulnerability where the attacker can overwrite the return address on the stack. Explain what attacks, if any, an attacker would be able to mount if the same program is run under XFI. Be specific.

5. [4 points]: Suppose a program has a buffer overflow vulnerability which allows an attacker to overwrite a function pointer on the stack (which is invoked shortly after the buffer is overflowed). Explain what attacks, if any, an attacker would be able to mount if the same program is run under XFI. Be specific.

6. [4 points]: Suppose a malicious XFI module, which is not allowed to invoke `unlink()`, attempts to remove arbitrary files by directly jumping to the `unlink()` code in `libc`. What precise instruction will fail when the attacker tries to do so, if any?

7. [6 points]: Suppose a malicious XFI module wants to circumvent XFI's inline checks in its code. To do so, the module allocates a large chunk of memory, copies its own executable code to it (assume XFI is running with only write-protection enabled, for performance reasons, so the module is allowed to read its own code), and replaces all XFI check instructions in the copied code with `NOP` instructions. The malicious module then calls a function pointer, whose value is the start of the copied version of the function that the module would ordinarily invoke. Does XFI prevent an attacker from bypassing XFI's checks in this manner, and if so, what precise instruction would fail?

III Privilege Separation

8. [4 points]: OKWS uses database proxies to control what data each service can access, but lab 2 has no database proxies. Explain what controls the data that each service can access in lab 2.

9. [8 points]: In lab 2, logging is implemented by a persistent process that runs under a separate UID and accepts log messages, so that an attacker that compromises other parts of the application would not be able to corrupt the log. Ben Bitdiddle dislikes long-running processes, but still wants to protect the log from attackers. Suggest an alternative design for Ben that makes sure past log messages cannot be tampered with by an attacker, but does not assume the existence of any long-running user process.

10. [6 points]: Ben proposes another strawman alternative to OKWS: simply use `chroot()` to run each service process in a separate directory root. Since each process will only be able to access its own files, there is no need to run each process under a separate UID. Explain why Ben's approach is faulty, and how an attacker that compromises one service will be able to compromise other services too.

IV Information Flow Control

11. [8 points]:

This problem was buggy; everyone received full credit.

V Java

12. [4 points]: When a privileged operation is requested, extended stack introspection walks up the stack looking for a stack frame which called `enablePrivilege()`, but stops at the first stack frame that is not authorized to call `enablePrivilege()`. Give an example of an attack that could occur if stack inspection did not stop at such stack frames.

13. [8 points]: Suppose you wanted to run an applet and allow it to connect over the network to web.mit.edu port 80, but nowhere else. In Java, opening a network connection is done by constructing a Socket object, passing the host and port as arguments to the constructor. Sketch out how you would implement this security policy with extended stack introspection, assuming that the system library implementing sockets calls `checkPrivilege("socket")` in the Socket constructor. Explain how the applet must change, if any.

14. [6 points]: Sketch out how you would implement the same security policy as in part (b), except by using name space management. Explain how the applet must change, if any.

VI Browser

15. [8 points]: The paper argues that the child policy (where a frame can navigate its immediate children) is unnecessarily strict, and that the descendant policy (where a frame can navigate the children of its children's frames, and so on) is just as good. Give an example of how the descendant policy can lead to security problems that the child policy avoids.

VII Resin

16. [8 points]: Sketch out the Resin filter and policy objects that would be needed to avoid cross-site scripting attacks through user profiles in `zoobar`. Assume that you have a PHP function to strip out JavaScript.

Didn't cover

VIII (6.893) Diff Class

We'd like to hear your opinions about 6.893, so please answer the following questions. (Any answer, except no answer, will receive full credit.)

17. [2 points]: How could we make the ideas in the course easier to understand?

18. [2 points]: What is the best aspect of 6.893?

19. [2 points]: What is the worst aspect of 6.893?

End of Quiz

Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.893 Fall 2009

Quiz I Solutions

All problems are open-ended questions. In order to receive credit you must answer the question as precisely as possible. You have 80 minutes to finish this quiz.

Write your name on this cover sheet.

Some questions may be harder than others. Read them all through first and attack them in the order that allows you to make the most progress. If you find a question ambiguous, be sure to write down any assumptions you make. Be neat. If we can't understand your answer, we can't give you credit!

THIS IS AN OPEN BOOK, OPEN NOTES EXAM.

Please do not write in the boxes below.

I (xx/18)	II (xx/16)	III (xx/18)	IV (xx/8)	V (xx/18)
VI (xx/8)	**VII (xx/8)**	**VIII (xx/6)**	**Total (xx/100)**	

The mean score on the quiz was 67, median was 62, and standard deviation 15.

Name:

I Buffer Overflows

Ben Bitdiddle is building a web server that runs the following code sequence, in which `process_req()` is invoked with a user-supplied string of arbitrary length. Assume that `process_get()` is safe, and for the purposes of this question, simply returns right away.

```
void process_req(char *input) {
    char buf[256];
    strcpy(buf, input);
    if (!strncmp(buf, "GET ", 4))
        process_get(buf);
    return;
}
```

1. [6 points]: Ben Bitdiddle wants to prevent attackers from exploiting bugs in his server, so he decides to make the stack memory non-executable. Explain how an attacker can still exploit a buffer overflow in his code to delete files on the server. Draw a stack diagram to show what locations on the stack you need to control, what values you propose to write there, and where in the input string these values need to be located.

Answer: An attacker can still take control of Ben's server, and in particular, remove files, by using a "return-to-libc" attack, where the return address is overflowed with the address of the `unlink` function in libc. The attacker must also arrange for the stack to contain proper arguments for `unlink`, at the right location on the stack.

2. [6 points]: Seeing the difficulty of preventing exploits with a non-executable stack, Ben instead decides to make the stack grow up (towards larger addresses), instead of down like on the x86. Explain how you could exploit `process_req()` to execute arbitrary code. Draw a stack diagram to illustrate what locations on the stack you plan to corrupt, and where in the input string you would need to place the desired values.

Answer: The return address from the `strcpy` function is on the stack following the `buf` array. If the attacker provides an input longer than 256 bytes, the subsequent bytes can overwrite the return address from `strcpy`, vectoring the execution of the program to an arbitrary address when `strcpy` returns.

3. [6 points]: Consider the StackGuard system from the "Buffer Overflows" paper in the context of Ben's new system where the stack grows *up*. Explain where on the stack the canary should be placed, at what points in the code the canary should be written, and at what points it should be checked, to prevent buffer overflow exploits that take control of the return address.

Answer: The canary must be placed at an address immediately before each function's return address on the stack. Because the stack grows up, this space must be reserved by the caller (although it's OK if the callee puts the canary value there, before executing any code that might overflow the stack and corrupt the return address). The callee must verify the canary value before returning to the caller.

II XFI

4. [2 points]: Suppose a program has a traditional buffer overflow vulnerability where the attacker can overwrite the return address on the stack. Explain what attacks, if any, an attacker would be able to mount if the same program is run under XFI. Be specific.

Answer: Because XFI has two stacks, the attacker will not be able to exploit a traditional buffer overflow (corrupting the return address). The attacker may still be able to corrupt other data or pointers on the allocation stack; see below.

5. [4 points]: Suppose a program has a buffer overflow vulnerability which allows an attacker to overwrite a function pointer on the stack (which is invoked shortly after the buffer is overflowed). Explain what attacks, if any, an attacker would be able to mount if the same program is run under XFI. Be specific.

Answer: The attacker can cause the module to start executing the start of any legal function in the XFI module, or any legal stub that will in turn execute allowed external functions.

6. [4 points]: Suppose a malicious XFI module, which is not allowed to invoke `unlink()`, attempts to remove arbitrary files by directly jumping to the `unlink()` code in `libc`. What precise instruction will fail when the attacker tries to do so, if any?

Answer: The CFI label check before the jump to `unlink` will notice that the `unlink` function does not have the appropriate CFI label, and abort execution.

7. [6 points]: Suppose a malicious XFI module wants to circumvent XFI's inline checks in its code. To do so, the module allocates a large chunk of memory, copies its own executable code to it (assume XFI is running with only write-protection enabled, for performance reasons, so the module is allowed to read its own code), and replaces all XFI check instructions in the copied code with `NOP` instructions. The malicious module then calls a function pointer, whose value is the start of the copied version of the function that the module would ordinarily invoke. Does XFI prevent an attacker from bypassing XFI's checks in this manner, and if so, what precise instruction would fail?

Answer: XFI assumes and relies on the hardware/OS to prevent execution of data memory (e.g. the NX flag on recent x86 CPUs).

III Privilege Separation

8. [4 points]: OKWS uses database proxies to control what data each service can access, but lab 2 has no database proxies. Explain what controls the data that each service can access in lab 2.

Answer: Lab 2 relies on file permissions (and data partitioning) to control what service can access what data.

9. [8 points]: In lab 2, logging is implemented by a persistent process that runs under a separate UID and accepts log messages, so that an attacker that compromises other parts of the application would not be able to corrupt the log. Ben Bitdiddle dislikes long-running processes, but still wants to protect the log from attackers. Suggest an alternative design for Ben that makes sure past log messages cannot be tampered with by an attacker, but does not assume the existence of any long-running user process.

Answer: One approach may be to use a setuid binary that will execute the logging service on-demand under the appropriate user ID.

10. [6 points]: Ben proposes another strawman alternative to OKWS: simply use `chroot()` to run each service process in a separate directory root. Since each process will only be able to access its own files, there is no need to run each process under a separate UID. Explain why Ben's approach is faulty, and how an attacker that compromises one service will be able to compromise other services too.

Answer: Processes running under the same UID can still kill or debug each other, even though they cannot interact through the file system.

IV Information Flow Control

11. [8 points]: This problem was buggy; everyone received full credit.

V Java

12. [4 points]: When a privileged operation is requested, extended stack introspection walks up the stack looking for a stack frame which called `enablePrivilege()`, but stops at the first stack frame that is not authorized to call `enablePrivilege()`. Give an example of an attack that could occur if stack inspection did not stop at such stack frames.

Answer: A luring attack, whereby trusted code that has called `enablePrivilege()` accidentally invokes untrusted code, which can then perform privileged operations.

13. [8 points]: Suppose you wanted to run an applet and allow it to connect over the network to web.mit.edu port 80, but nowhere else. In Java, opening a network connection is done by constructing a Socket object, passing the host and port as arguments to the constructor. Sketch out how you would implement this security policy with extended stack introspection, assuming that the system library implementing sockets calls `checkPrivilege("socket")` in the Socket constructor. Explain how the applet must change, if any.

Answer: Something like the following code:

```
public class MitSocketFactory {
    public static Socket getSocket() {
        enablePrivilege("socket");
        return new Socket("web.mit.edu", 80);
    }
}
```

The applet's code would need to invoke `MitSocketFactory.getSocket()` instead of using the `Socket` constructor directly.

14. [6 points]: Sketch out how you would implement the same security policy as in part (b), except by using name space management. Explain how the applet must change, if any.

Answer: Replace the Socket object with MitSocket in the applet's namespace:

```
public class MitSocket extends Socket {
    public MitSocket(String host, int port) {
        super();
        if (!host.equal("web.mit.edu") || port != 80)
            throw SecurityException("only web.mit.edu:80 allowed");
        connect(host, port);
    }

    ...
}
```

The applet would not have to change. Note that `MitSocket`'s constructor does not call `super(host, port)`. Instead, it invokes `super()` and calls `connect(host, port)` later. Invoking the `super(host, port)` constructor would have allowed the applet to open connections to arbitrary hosts (which would then be immediately closed), by constructing an `MitSocket` object with the right `host` and `port` arguments, since the security check comes after the superclass constructor.

VI Browser

15. [8 points]: The paper argues that the child policy (where a frame can navigate its immediate children) is unnecessarily strict, and that the descendant policy (where a frame can navigate the children of its children's frames, and so on) is just as good. Give an example of how the descendant policy can lead to security problems that the child policy avoids.

Answer: Consider a web site that contains a login frame, where users are expected to input passwords. Under the descedant policy, an attacker can put the web site in a frame, and navigate the login frame (a grandchild) to http://attacker.com, which looks similar to the original one, to steal passwords.

VII Resin

16. [8 points]: Sketch out the Resin filter and policy objects that would be needed to avoid cross-site scripting attacks through user profiles in `zoobar`. Assume that you have a PHP function to strip out JavaScript.

Answer: There are several possible solutions. One approach is to define two "empty" policies, *UnsafePolicy* and *JSSanitizedPolicy*, the *export_check* functions of which do nothing. Input strings are tagged *UnsafePolicy*, and the PHP function to strip out JavaScript attaches *JSSanitizedPolicy* to resulting strings. The standard output filter checks that strings must contain neither or both policies.

VIII 6.893

We'd like to hear your opinions about 6.893, so please answer the following questions. (Any answer, except no answer, will receive full credit.)

17. [2 points]: How could we make the ideas in the course easier to understand?

More and simpler examples to illustrate the problem;
More labs.

18. [2 points]: What is the best aspect of 6.893?

Labs;
Recent papers.

19. [2 points]: What is the worst aspect of 6.893?

Long, conceptual papers;
Repetitive, time-consuming labs.

End of Quiz

10/22

Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.858 Fall 2012

Quiz I

You have 80 minutes to answer the questions in this quiz. In order to receive credit you must answer the question as precisely as possible.

Some questions are harder than others, and some questions earn more points than others. You may want to skim them all through first, and attack them in the order that allows you to make the most progress.

If you find a question ambiguous, be sure to write down any assumptions you make. Be neat and legible. If we can't understand your answer, we can't give you credit!

Write your name and submission website username (typically your Athena username) on this cover sheet.

This is an open book, open notes, open laptop exam.
NO INTERNET ACCESS OR OTHER COMMUNICATION.

Please do not write in the boxes below.

I (xx/20)	II (xx/16)	III (xx/12)	IV (xx/20)	V (xx/10)	VI (xx/16)	VII (xx/6)	Total (xx/100)
12	4 GB	7 NZ	20	3	4	6	56
T			FW	T	FW		

Name: Michael Plasmeier

Submission website username:

theplaz

(12(-8)

I Buffer overflows

Consider the following C program, where an adversary can supply arbitrary input on `stdin`. Assume no compiler optimizations, and assume a 32-bit system. In this example, `fgets()` never writes past the end of the 256-byte `buf`, and always makes it NULL-terminated.

fgets() - stops when n-1 read new line / EOF

```
int main() {
  char buf[256];
  fgets(buf, 256, stdin);
  foo(buf);
  printf("Hello world.\n");
}
```

runs buf w/ foo

null always at end
ret pointer to string

1. [12 points]: Suppose the `foo` function is as follows:

```
void foo(char *buf) {
  char tmp[200];  // assume compiler places "tmp" on the stack

  // copy from buf to tmp
  int i = 0;      // assume compiler places "i" in a register
  while (buf[i] != 0) {
    tmp[i] = buf[i];
    i++;
  }
}
```

no limit - until nul - which is always 256 but tmp 200

buf
arg
ret
ebp
tmp

Which of the following are true? Assume there is an unmapped page both above and below the stack in the process's virtual memory.

(Circle True or False for each choice.)

Stack usually grows down

A. True / False An adversary can trick the program to delete files on a system where the stack grows down.

B. True / False An adversary can trick the program to delete files on a system where the stack grows up.

Can't overflow that far

C. True / False An adversary can trick the program to delete files on a system using Baggy bounds checking with `slot_size=16`. (Stack grows down.)

grows to 256

D. True / False An adversary can prevent the program from printing "Hello world" on a system using Baggy bounds checking with `slot_size=16`. (Stack grows down.)

opposite

E. True / False An adversary can trick the program to delete files on a system using terminator stack canaries for return addresses. (Stack grows down.)

Can't called

F. True / False An adversary can prevent the program from printing "Hello world" on a system using terminator stack canaries for return addresses. (Stack grows down.)

opposite prevented

2. [8 points]: Suppose the `foo` function is as follows:

```
struct request {
  void (*f)(void);  // function pointer
  char path[240];
};

void foo(char *buf) {
  struct request r;
  r.f = /* some legitimate function */;
  strcpy(r.path, buf);
  r.f();
}
```

Which of the following are true?

(Circle True or False for each choice.)

A. True / False An adversary can trick the program to delete files on a system where the stack grows down.

B. True / False An adversary can trick the program to delete files on a system where the stack grows up.

C. True / False An adversary can trick the program to delete files on a system using Baggy bounds checking with `slot_size=16`. Assume `strcpy` is compiled with Baggy. (Stack grows down.)

D. True / False An adversary can prevent the program from printing "Hello world" on a system using Baggy bounds checking with `slot_size=16`. Assume `strcpy` is compiled with Baggy. (Stack grows down.)

II OS sandboxing

Ben Bitdiddle is modifying OKWS to use Capsicum. To start each service, Ben's `okld` forks, opens the service executable binary, then calls `cap_enter()` to enter capability mode in that process, and finally executes the service binary. Each service gets file descriptors only for sockets connected to okd, and for TCP connections to the relevant database proxies.

4

3. [6 points]: Which of the following changes are safe now that the services are running under Capsicum, assuming the kernel implements Capsicum perfectly and has no other bugs?
(Circle True or False for each choice.)

A. True / False It is safe to run all services with the same UID/GID.

B. True / False It is safe to run services without `chroot`. Capsicum mirrors

C. True / False It is safe to also give each service an open file descriptor for a per-service directory `/cores/servicename`.

UID/GID used to provide protection from other accessing
- other files
- but assuming fd set correctly

Also need to give db proxy access

We can run multiple copies of same service
don't want them to interact!

Ben also considers replacing the `oklogd` component with a single log file, and giving each service a file descriptor to write to the log file.

4. [5 points]: What should `okld` do to ensure one service cannot read or overwrite log entries from another service? Be as specific as possible.

okld should have a API that only allows services to suggest log entries to add
okld should then do the actual writing - being the only process that can write these files
(e.g. int fd[...], CAP_WRITE)

5. [5 points]: What advantages could an `oklogd`-based design have over giving each service a file descriptor to the log file?

As mentioned above → prevents modification/tampering/deletion of log entries

Also function abstraction - so free to change underlying storage mechanism later on

- Prevents writing half a log entry
- enforce structure

III Network protocols

Ben Bitdiddle is designing a file server that clients connect to over the network, and is considering using either Kerberos (as described in the paper) or SSL/TLS (without client certificates, where users authenticate using passwords) for protecting a client's network connection to the file server. For this question, assume users choose hard-to-guess passwords.

6

6. [6 points]: Would Ben's system remain secure if an adversary learns the server's private key, but that adversary controls only a single machine (on the adversary's own home network), and does not collude with anyone else? Discuss both for Kerberos and for SSL/TLS.

does server store session key - no includes in ticket

Kerberos With the server's private key to make their own initial tickets which they can send to the TGS along w/ any service id to get tickets for it. So attacker can get access to any service for any user.

TLS With the server's private key, the attacker could view any traffic sent to/from the server (ARP poisoning) but this is below - can't do anything yet besides fake server traffic for its local app - ultimately useless

7. [6 points]: Suppose an adversary learns the server's private key as above, and the adversary also controls some network routers. Ben learns of this before the adversary has a chance to take any action. How can Ben prevent the adversary from mounting attacks that take advantage of the server's private key (e.g., not a denial-of-service attack), and when will the system be secure? Discuss both for Kerberos and for SSL/TLS.

1

0

Kerberos The attacker could not function as a rogue authenticator giving any body access to any service because it does not have the password db to be able to make messages

1

TLS ~~Ben~~ could change the servers public/private key. Then system secure when all clients have new public key

nice synopsis!

IV Static analysis

Would Yichen Xie's PHP static analysis tool for SQL injection bugs, as described in the paper, flag a potential error/warning in the following short but complete PHP applications?

8. [10 points]:

A. True / False The tool would report a potential error/warning in the following code:

```
function q($s) {
  return mysql_query($s);
}

$x = $_GET['id'];
q("SELECT .. $x");
```

Argument not sanitized before running

What does this do?

B. True / False The tool would report a potential error/warning in the following code:

```
function my_validate() {
  return isnumeric($_GET['id']);
}

$x = $_GET['id'];
if (my_validate()) {
  mysql_query("SELECT .. $x");
}
```

is_numeric does count as sanitize

← if true is sanitized

Only run if true, so good

9. [10 points]:

A. True / False The tool would report a potential error/warning in the following code:

```
mysql_query("SELECT .. $n");
```

(completly unsanitized input

$n would be in E

(which is by default w/ stored code

B. True / False The tool would report a potential error/warning in the following code:

```
function check_arg($n) {
  $v = $_GET[$n];
  return isnumeric($v);
}

$x = $_GET['id'];
if (check_arg('id')) {
  mysql_query("SELECT .. $x");
}
```

checks

← but not here

↙ just uses X

it would not be smart enough to flag this silly code

V Runtime instrumentation

10. [10 points]:

Consider the following Javascript code:

```
function foo(x, y) {
  return x + y;
}

var a = 2;
eval("foo(a, a)");

var p_c = {
  k: 5,
  f: function() { return a + this.k; }
};
var kk = 'k';
p_c[kk] = 6;
p_c.f();
```

Based on the description in the paper by Sergio Maffeis et al, and based on lecture 9, what will be the FBJS rewritten version of this code, assuming the application-specific prefix is p_?

```
function foo(x, y) {
  return x + y;
}
var p_a = 2;
eval("foo(a, a)");   no eval!, error message/parse error
var p_p_c = {
  k: 5
  f: function(){ return p_a + FBJS.ref(this).k; }
};
var p_kk = 'k';
p_p_c[p_kk] = 6;
p_p_c.f();
```

VI Browser security

old submission

Ben Bitdiddle is taking 6.858. Once he's done with his lab at 4:55pm, he rushes to submit it by going to `https://taesoo.scripts.mit.edu/submit/handin.py/student`, selecting his `labN-handin.tar.gz` file, and clicking "Submit". The 6.858 submission web site also allows a student to download a copy of their past submission.

For your reference, when the user logs in, the submission web site stores a cookie in the user's browser to keep track of their user name. To prevent a user from constructing their own cookie, or arbitrarily changing the value of an existing cookie, the server includes a signature/MAC of the cookie's value in the cookie, and checks the signature when a new request comes in. Finally, users can log out by clicking on the "Logout" link, `https://taesoo.scripts.mit.edu/submit/handin.py/logout`, which clears the cookie.

Alyssa P. Hacker, an enterprising 6.858 student, doesn't want to do lab 5, and wants to get a copy of Ben's upcoming lab 5 submission instead. Alyssa has her own web site at `https://alyssa.scripts.mit.edu/`, and can convince Ben to visit that site at any point.

11. [16 points]: How can Alyssa get a copy of Ben's lab 5 submission?

Alyssa's attack should rely only on the Same-Origin Policy. Assume there are no bugs in any software, Ben's (and Taesoo's) password is unguessable, the cookie signature scheme is secure, etc.

Alyssa has Ben visit her page.
On the page Alyssa has an iframe load the submission page. Since the iframe is running on Alyssa's page, she can have access to the cookie We can read its cookies w/ document.cookie.
Alyssa then sets her browser to that cookie and visits the submission site
Or if Taesoo's site has a XSS vunerability in the iframe URL add get param to include script to load img at attacker.com ; get = document.cookie

4

close! not about stealing but should set because of SOP.

VII 6.858

We'd like to hear your opinions about 6.858. Any answer, except no answer, will receive full credit.

12. [2 points]: This year we started using Piazza for questions and feedback. Did you find it useful, and how could it be improved?

Yes Piazza is helpful, esp for conceptual qu
It's not a substute for in-person lab help
Fast response times always appreciated :)

13. [2 points]: What aspects of the labs were most time-consuming? How can we make them less tedious?

Struggling w/o help - more OH - this has been getting better
Needing to know a lot about framework / give code
- provide more hints / documentation for those

14. [2 points]: Are there other things you'd like to see improved in the second half of the semester?

The labs in this class are relatively new (2-3 years)
It takes some time for them to get up to high quality
More hints on the tricky parts of the lab
esp when they are not focusing on the concepts from class.

End of Quiz

1st → 30 min

9/10/22

Post

Didn't know it can reach into iframe
Still unclear on the buffer overflow questions

PHP ~~This~~ static don't know details
Same w/ FBJS
) but good enough

Rest was good I think
Kerberos I ~~might~~ have gotten

Same domain iframe does work
Same 3rd level domains can com w/ 2nd level domain
(but that still won't get it to work)
Ah must change to common second level domain w/ document.domain

Exam Debief

10/22

Looks like iframe default must be exact same domain

Unless calls document.domain

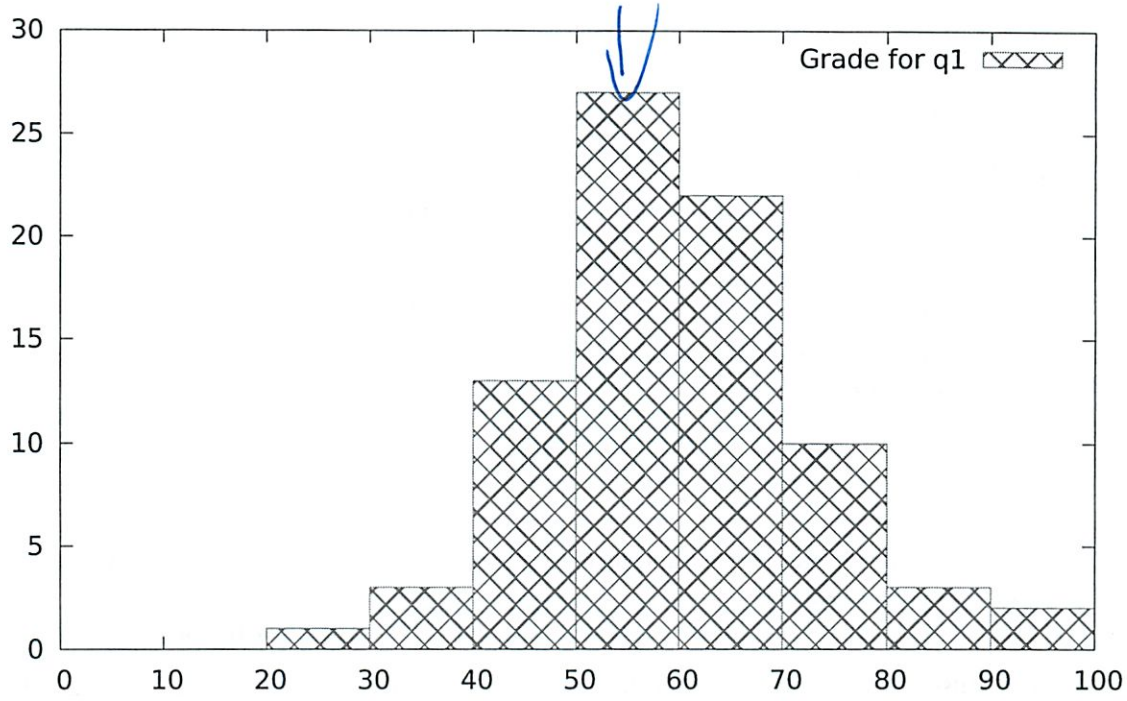
Then how do you do it?

Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.858 Fall 2012

Quiz I Solutions



Histogram of grade distribution

I Buffer overflows

Consider the following C program, where an adversary can supply arbitrary input on `stdin`. Assume no compiler optimizations, and assume a 32-bit system. In this example, `fgets()` never writes past the end of the 256-byte `buf`, and always makes it NULL-terminated.

```
int main() {
  char buf[256];
  fgets(buf, 256, stdin);
  foo(buf);
  printf("Hello world.\n");
}
```

1. [12 points]: Suppose the `foo` function is as follows:

```
void foo(char *buf) {
  char tmp[200];  // assume compiler places "tmp" on the stack

  // copy from buf to tmp
  int i = 0;      // assume compiler places "i" in a register
  while (buf[i] != 0) {
    tmp[i] = buf[i];
    i++;
  }
}
```

Which of the following are true? Assume there is an unmapped page both above and below the stack in the process's virtual memory.

(Circle True or False for each choice.)

A. True / False An adversary can trick the program to delete files on a system where the stack grows down.

Answer: True. The adversary can overwrite `foo`'s return address.

B. True / False An adversary can trick the program to delete files on a system where the stack grows up.

Answer: False. If the stack grows up, then no other state is placed above `tmp` on the stack, so even if the adversary overflows `tmp`, it will not affect the program's execution.

C. True / False An adversary can trick the program to delete files on a system using Baggy bounds checking with `slot_size=16`. (Stack grows down.)

Answer: False. Baggy will prevent memory writes to `tmp` from overflowing to the return address or any other stack variable.

D. True / False An adversary can prevent the program from printing "Hello world" on a system using Baggy bounds checking with `slot_size=16`. (Stack grows down.)

Answer: False. The argument `buf` points to a string that is at most 255 bytes long, since `fgets` NULL-terminates the buffer. Baggy enforces a power-of-2 allocation bound for `tmp`, which ends up being 256 bytes.

E. True / False An adversary can trick the program to delete files on a system using terminator stack canaries for return addresses. (Stack grows down.)

Answer: False. A terminator stack canary includes a NULL byte, and if the adversary overwrites the return address on the stack, the canary value will necessarily not contain any NULL bytes (since otherwise the `while` loop would have exited).

F. True / False An adversary can prevent the program from printing "Hello world" on a system using terminator stack canaries for return addresses. (Stack grows down.)

Answer: True. The adversary could simply overwrite the canary value, which will terminate the program as `foo` returns.

2. [8 points]: Suppose the `foo` function is as follows:

```
struct request {
  void (*f)(void);  // function pointer
  char path[240];
};

void foo(char *buf) {
  struct request r;
  r.f = /* some legitimate function */;
  strcpy(r.path, buf);
  r.f();
}
```

Which of the following are true?

(Circle True or False for each choice.)

A. True / False An adversary can trick the program to delete files on a system where the stack grows down.

Answer: True. The adversary can overwrite `foo`'s return address on the stack.

B. True / False An adversary can trick the program to delete files on a system where the stack grows up.

Answer: True. The adversary can overwrite `strcpy`'s return address on the stack.

C. True / False An adversary can trick the program to delete files on a system using Baggy bounds checking with `slot_size=16`. Assume `strcpy` is compiled with Baggy. (Stack grows down.)

Answer: False. Baggy bounds checking will prevent `strcpy` from going past `r`'s allocation bounds, and `r.f` is before `r.path` in `r`'s memory layout.

D. True / False An adversary can prevent the program from printing "Hello world" on a system using Baggy bounds checking with `slot_size=16`. Assume `strcpy` is compiled with Baggy. (Stack grows down.)

Answer: True. If the adversary supplies 255 bytes of input, then `strcpy` will write past `r`'s allocation bounds of 256 bytes, and Baggy will terminate the program.

II OS sandboxing

Ben Bitdiddle is modifying OKWS to use Capsicum. To start each service, Ben's `okld` forks, opens the service executable binary, then calls `cap_enter()` to enter capability mode in that process, and finally executes the service binary. Each service gets file descriptors only for sockets connected to `okd`, and for TCP connections to the relevant database proxies.

3. [6 points]: Which of the following changes are safe now that the services are running under Capsicum, assuming the kernel implements Capsicum perfectly and has no other bugs?
(Circle True or False for each choice.)

A. True / False It is safe to run all services with the same UID/GID.

Answer: True.

B. True / False It is safe to run services without `chroot`.

Answer: True.

C. True / False It is safe to also give each service an open file descriptor for a per-service directory `/cores/servicename`.

Answer: True.

Ben also considers replacing the `oklogd` component with a single log file, and giving each service a file descriptor to write to the log file.

4. [5 points]: What should `okld` do to ensure one service cannot read or overwrite log entries from another service? Be as specific as possible.

Answer: `okld` should call:

```
lc_limitfd(logfd, CAP_WRITE);
```

To ensure that the service cannot seek, truncate, or read the log file.

5. [5 points]: What advantages could an `oklogd`-based design have over giving each service a file descriptor to the log file?

Answer: `oklogd` can enforce structure on the log file, such as adding a timestamp to each record, ensuring each record is separated from other records by a newline, ensuring multiple records are not interleaved, etc.

III Network protocols

Ben Bitdiddle is designing a file server that clients connect to over the network, and is considering using either Kerberos (as described in the paper) or SSL/TLS (without client certificates, where users authenticate using passwords) for protecting a client's network connection to the file server. For this question, assume users choose hard-to-guess passwords.

6. [6 points]: Would Ben's system remain secure if an adversary learns the server's private key, but that adversary controls only a single machine (on the adversary's own home network), and does not collude with anyone else? Discuss both for Kerberos and for SSL/TLS.

Answer: With Kerberos, no: the adversary can impersonate any user to this server, by constructing any ticket using the server's private key.

With SSL, yes: the server can impersonate the server to another client, but no clients will connect to the adversary's fake server.

7. [6 points]: Suppose an adversary learns the server's private key as above, and the adversary also controls some network routers. Ben learns of this before the adversary has a chance to take any action. How can Ben prevent the adversary from mounting attacks that take advantage of the server's private key (e.g., not a denial-of-service attack), and when will the system be secure? Discuss both for Kerberos and for SSL/TLS.

Answer: With Kerberos, Ben should change the server's private key. The system will be secure from that point forward. If the adversary was recording network traffic from before the attack, Ben should also make sure he changes the server's private key over a secure network link, because `kpasswd` does not provide forward secrecy. The adversary may be able to decrypt network traffic to the file server before the key is changed.

With SSL, Ben should obtain a new SSL certificate for the server, with a new secret key, but the adversary can continue to impersonate Ben's file server until the compromised certificate expires. The system will only be secure once the certificate expires, or once all clients learn of the certificate being revoked.

IV Static analysis

Would Yichen Xie's PHP static analysis tool for SQL injection bugs, as described in the paper, flag a potential error/warning in the following short but complete PHP applications?

8. [10 points]:

A. True / False The tool would report a potential error/warning in the following code:

```
function q($s) {
  return mysql_query($s);
}

$x = $_GET['id'];
q("SELECT .. $x");
```

Answer: True. The summary for `q()` indicates that the argument must be sanitized on entry, and the main function does not sanitize the argument.

B. True / False The tool would report a potential error/warning in the following code:

```
function my_validate() {
  return isnumeric($_GET['id']);
}

$x = $_GET['id'];
if (my_validate()) {
  mysql_query("SELECT .. $x");
}
```

Answer: False. The summary for `my_validate()` indicates that `$_GET[id]` is sanitized if the return value is true.

9. [10 points]:

A. True / False The tool would report a potential error/warning in the following code:

```
mysql_query("SELECT .. $n");
```

Answer: True. The tool reports warnings when any variable must be sanitized on entry into the main function, and the variable is not known to be easily controlled by the user, such as `$_GET` and `$_POST`.

B. True / False The tool would report a potential error/warning in the following code:

```
function check_arg($n) {
  $v = $_GET[$n];
  return isnumeric($v);
}

$x = $_GET['id'];
if (check_arg('id')) {
  mysql_query("SELECT .. $x");
}
```

Answer: True. The summary for `check_arg()` indicates that `$_GET[⊥]` is sanitized if the return value is true, but the call to `mysql_query()` requires `$_GET[id]` to be sanitized.

V Runtime instrumentation

10. [10 points]:

Consider the following Javascript code:

```
function foo(x, y) {
  return x + y;
}

var a = 2;
eval("foo(a, a)");

var p_c = {
  k: 5,
  f: function() { return a + this.k; }
};
var kk = 'k';
p_c[kk] = 6;
p_c.f();
```

Based on the description in the paper by Sergio Maffeis et al, and based on lecture 9, what will be the FBJS rewritten version of this code, assuming the application-specific prefix is p_?

Answer: FBJS adds a p_ prefix to every variable name, and wraps `this` and variable array indexes in `$FBJS.ref()` and `$FBJS.idx()` respectively.

```
function p_foo(p_x, p_y) {
  return p_x + p_y;
}

var p_a = 2;
p_eval("foo(a, a)");

var p_p_c = {
  k: 5,
  f: function() { return p_a + $FBJS.ref(this).k; }
};
var p_kk = 'k';
p_p_c[$FBJS.idx(p_kk)] = 6;
p_p_c.f();
```

VI Browser security

Ben Bitdiddle is taking 6.858. Once he's done with his lab at 4:55pm, he rushes to submit it by going to `https://taesoo.scripts.mit.edu/submit/handin.py/student`, selecting his `lab`*N*`-handin.tar.gz` file, and clicking "Submit". The 6.858 submission web site also allows a student to download a copy of their past submission.

For your reference, when the user logs in, the submission web site stores a cookie in the user's browser to keep track of their user name. To prevent a user from constructing their own cookie, or arbitrarily changing the value of an existing cookie, the server includes a signature/MAC of the cookie's value in the cookie, and checks the signature when a new request comes in. Finally, users can log out by clicking on the "Logout" link, `https://taesoo.scripts.mit.edu/submit/handin.py/logout`, which clears the cookie.

Alyssa P. Hacker, an enterprising 6.858 student, doesn't want to do lab 5, and wants to get a copy of Ben's upcoming lab 5 submission instead. Alyssa has her own web site at `https://alyssa.scripts.mit.edu/`, and can convince Ben to visit that site at any point.

11. [16 points]: How can Alyssa get a copy of Ben's lab 5 submission?

Alyssa's attack should rely only on the Same-Origin Policy. Assume there are no bugs in any software, Ben's (and Taesoo's) password is unguessable, the cookie signature scheme is secure, etc.

Answer: Alyssa's web site should force Ben's browser to log out from the 6.858 submission web site, by inserting the following tag:

```
<IMG SRC="https://taesoo.scripts.mit.edu/submit/handin.py/logout">
```

and then set a cookie for `domain=scripts.mit.edu` containing Alyssa's own cookie. When Ben visits the submission web site to upload his lab 5, he will actually end up uploading it under Alyssa's username, allowing Alyssa to then download it at 4:56pm.

VII 6.858

We'd like to hear your opinions about 6.858. Any answer, except no answer, will receive full credit.

12. [2 points]: This year we started using Piazza for questions and feedback. Did you find it useful, and how could it be improved?

Answer: Generally good; UI not so great; appreciate anonymity. Would be good if answers show up quicker. Bypass email preferences for important announcements. In-person office hours are also important. Submit paper questions via Piazza. Signal-to-noise ratio too low. More TA/professor participation other than David. Ask people not to be anonymous. Separate login is annoying. RSS feed.

13. [2 points]: What aspects of the labs were most time-consuming? How can we make them less tedious?

Answer: Include more debugging tools, especially for the PyPy sandbox. Explain where errors / debug output goes. Explanation of provided lab code; explain what parts to focus on. Start discussions of papers. Avoid asking for the same thing multiple times in the lab; 2nd part of lab 2 was repetitive. Speed up the VM / run Python fewer times. More office hours. Better / more fine-grained / faster make check. Review/recitation session to provide background knowledge for a lab.

14. [2 points]: Are there other things you'd like to see improved in the second half of the semester?

Answer: Past exploits. More time on labs. More late days. More summaries of papers / what to focus on / background info. More attacks. More recent papers. Explicit lectures on lab mechanics. More design freedom in labs / more challenging design choices; less fill-in-the-blank style. Some papers are too technical. Fewer ways to turn things in (submit via make; submit via text file; email question; Piazza). Balance first and second parts of labs. Novelty lecture on lockpicking. Shorter quiz. Weekly review of lecture (not labs) material – recitation? Relate labs to lectures, teach more hands-on stuff. Labs where you solve some problem rather than get the details right? Explain what corner cases matter for labs. Lecture should focus on application of paper's ideas and short review of paper content. More info on final projects. Scrap the answers.txt stuff, just code.

End of Quiz