6.858 Fall 2012 Lab 2: Privilege separation

Handed out: Wednesday, September 19, 2012
Part 1 due: Friday, September 28, 2012 (5:00pm)
All parts due: Friday, October 5, 2012 (5:00pm)

Introduction

This lab will introduce you to privilege separation, in the context of a simple python web application called `zoobar`, where users transfer "zoobars" (credits) between each other. To help you privilege-separate this application, the `zookws` web server used in the previous lab is a clone of the OKWS web server, discussed in lecture 3. In this lab, you will set up a privilege separated web server, examine possible vulnerabilities, and break up the application code into less-privileged components to minimize the effects of any single vulnerability.

To fetch the new source code, use Git to commit your Lab 1 solutions, and merge them into our lab2 branch:

```
httpd@vm-6858:~$ cd lab
httpd@vm-6858:~/lab$ git add answers.txt exploit-*.py [and any other new files...]
httpd@vm-6858:~/lab$ git commit -am 'my solution to lab1'
[lab1 c54dd4d] my solution to lab1
 1 files changed, 1 insertions(+), 0 deletions(-)
httpd@vm-6858:~/lab$ git pull
...
httpd@vm-6858:~/lab$ git checkout -b lab2 origin/lab2
Branch lab2 set up to track remote branch lab2 from origin.
Switched to a new branch 'lab2'
httpd@vm-6858:~/lab$ git merge lab1
Merge made by recursive.
...
httpd@vm-6858:~/lab$
```

In some cases, Git may not be able to figure out how to merge your changes with the new lab assignment (e.g. if you modified some of the code that is changed in the second lab assignment). In that case, the `git merge` command will tell you which files are *conflicted*, and you should first resolve the conflict (by editing the relevant files) and then commit the resulting files with `git commit -a`.

Once your source code is in place, make sure that you can compile and install the web server and the `zoobar` application:

```
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
...
+ python /jail/zoobar/zoodb.py init-person
+ python /jail/zoobar/zoodb.py init-transfer
httpd@vm-6858:~/lab$
```

The web server for this lab uses the `/jail` directory to setup `chroot` jails for different parts of the web server, much as in the OKWS paper. The `make` command compiles the web server, and `make setup` installs it with all the necessary permissions in the `/jail` directory.

As part of this lab, you will need to change how the files and directories are installed, such as changing their owner or permissions. To do this, you should *not* change the permissions directly. Instead, you should edit the `chroot-setup.sh` and `chroot-copy.sh` scripts in the `lab` directory, and re-run `sudo make setup`.

So they can grade

Now, make sure you can run the web server, and access the web site from your browser, as follows:

```
httpd@@vm-6858:~/lab$ /sbin/ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:0c:29:57:90:a1
          inet addr:172.16.91.143  Bcast:172.16.91.255  Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:fe57:90a1/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:149 errors:0 dropped:0 overruns:0 frame:0
          TX packets:94 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:15235 (15.2 KB)  TX bytes:12801 (12.8 KB)
          Interrupt:19 Base address:0x2000
httpd@vm-6858:~/lab$ sudo ./zookld
```

In this particular example, you would want to open your browser and go to `http://172.16.91.143:8080/zoobar/index.cgi`, or, if you are using KVM, to `http://localhost:8080/zoobar/index.cgi`. You should see the `zoobar` web site. Play around with this web application to get a feel for what it allows users to do. In short, a registered user can update his/her profile, transfer "zoobars" (credits) to another user, and look up the zoobar balance, profile, and transactions of other users in the system.

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: Web Server Setup

In the first part of this lab assignment, you will read the provided source code of the `zookws` web server and further secure it using privilege separation.

As introduced in Lab 1, the `zookws` web server is modeled after OKWS from lecture 4. Similar to OKWS, `zookws` consists of a launcher daemon `zookld` that launches services configured in the file `zook.conf`, a dispatcher `zookd` that routes requests to corresponding services, as well as several services. For simplicity `zookws` does not implement helper or logger daemon as OKWS does.

By default `zookws` configures only one HTTP service, `simple_svc`, that serves static files and executes dynamic scripts. The `simple_svc` does so by invoking the executable `zookfs`, which is jailed in the directory `/jail` by `chroot`. You can look into `/jail`; it contains executables (except for `zookld`), supporting libraries, and the `zoobar` web site. See `zook.conf` and `zookfs.c` for details.

So thats why 2 programs?

The launcher daemon `zookld` is running under root and can bind to a privileged port like 80. Note that in the default configuration, `zookd` and vulnerable services are *inappropriately* running under root; an attacker can exploit buffer overflows and cause damages to the server, e.g., unlink a specific file as you have done in Lab 1.

To fix the problem, you should run these services under *unprivileged users* rather than root.

Exercise 1. Modify the configuration file `zook.conf` to run `zookd` and other services under unprivileged user IDs. Make sure that the attacker cannot read or write to sensitive files in the server, even if `zookd` and these services contain buffer overflow vulnerabilities. At the same time, ensure that the zoobar application continues to work: that new users can register, transfer credits, and so on.

You will need to pick additional UIDs and GIDs, set them for each service in `zook.conf`, and edit the `chroot-setup.sh` and `chroot-copy.sh` scripts as necessary. Remember to re-run `sudo make setup` after changing scripts that set up `/jail` if you want your changes to take effect. Keep in mind that the `chroot-setup.sh` and `chroot-copy.sh` scripts will be re-run each time by `make setup` and `make check`, so any commands that you add to these scripts should be safe to run multiple times.

Run `sudo make check` to verify that your modified configuration passes our basic tests (although keep in mind that our tests are not exhaustive).

Now that none of the services are running as root, we will try to further privilege-separate the `simple_svc` service that handles both static files and dynamic scripts. Although it runs under an unprivileged user, some python scripts could easily have security holes; a vulnerable python script could be tricked into deleting important static files that the server is serving. A better organization is to split `simple_svc` into two services, one for static files and the other for python scripts, running under different users.

Additionally, a client should only be able to run the intended python scripts in the `/zoobar` directory from a browser, namely the `/zoobar/index.cgi` script. For example, a client should not be able to run `/password.cgi`, which reveals the root password of the server. Similarly, a client should not be able to directly fetch the database files `person.db` and `transfer.db` via HTTP.

block those

Exercise 2. Create two new HTTP services, along the lines of the existing `simple_svc` service, such that one will only execute dynamic content, and one which will only serve static files. Modify the configuration file `zook.conf` to split the `simple_svc` service into two services running under different users: the `static` service that only serves static files, and the `dynamic` service that only executes intended python scripts in the `zoobar` directory.

.HTaccess -> not apache

You may use url filtering provided in `zook.conf`, which supports regular expressions. For example, `url = .*` matches all requests, while `url = /zoobar/(abc|def)\.html` only allows requests to `/zoobar/abc.html` and `/zoobar/def.html`. You may also find it helpful to modify the server to not serve paths containing `..`, although it is not the only solution.

Run `sudo make check` to verify that your modified configuration passes our tests.

Now that we have privilege-separated the handling of static and dynamic content in the web server, we will look at fixing some bugs in the `zoobar` web application code. One of the key features of the `zoobar` application is the ability to transfer credits between users. This feature is implemented by the script `transfer.py`. Unfortunately, `transfer.py` has some logical bugs that may result in wrong transfers.

Exercise 3. Fix as many logical bugs as you can find in `transfer.py` (don't worry about browser-side attacks such as XSS for now) and note them in `answers.txt`. Think carefully about what kinds of inputs an attacker might provide. In our solution, there are three vulnerabilities.

Deliverables

Explain in `answers.txt` your changes to the zookws source code, configuration, and setup scripts for each exercise. Feel free to include any comments about your solutions in the `answers.txt` file (we would appreciate any feedback you may have on this assignment).

Submit your answers to the first part of the lab assignment by running `make submit` to upload `lab2-handin.tar.gz` to the submission web site.

due this week

- does not seem that bad...
- famous last words
- and no ** debugging

Part 2: The `zoobar` Web Site

In the rest of this lab assignment, you will further secure the `zoobar` web site using privilege separation.

In the previous exercise, we fixed the bugs in the transfer code; now we would like to make sure we can deal with any future such bugs that come up. To do so, we want to make sure that we have a reliable log of all zoobar transfers that happened in the system. The current design stores the transfer log in the `transfer` SQL table, stored in `zoobar/db/transfer/transfer.db`. This table is accessible to all python code in the `zoobar` site, which means that an attacker might be able to change the history so that we will never find out about his or her attack.

Oh gr - hmm

We will try to make the transfer log more reliable by performing the logging operations in a separate process, running as a different user from the rest of the `zoobar` code. This user ID will only run logging code, which will insert new log entries into the `transfer` table. By setting permissions on the `zoobar/db/transfer` directory accordingly, we will ensure that only the logging code (which will hopefully be trustworthy) will be able to modify log entries, but any other python code will be able to read the log.

To break off some python code into a separate process, running as a separate user ID, we have provided you with some helper tools. The `zooksvc` service creates a Unix domain socket, and when someone connects to this socket, it will launch an arbitrary program. We have created a simple echo service using this tool. Look at how `zook.conf` spawns this echo service, the source code for the echo service tool in `zoobar/svc-echo.py`, and the sample client of this service in `zoobar/demo-client.py`. The client uses a simple library for connecting to Unix domain socket services, in `zoobar/unixclient.py`. Note that the client is meant to be invoked from the command line, rather than being executed as a CGI script via HTTP.

To debug the low-level protocol between the client and the server, you can use the netcat tool. For example, once `zookld` is running and has started the echo service, you should be able to connect to and interact with the echo service as follows:

neat

```
httpd@vm-6858:~/lab$ sudo nc -U /jail/echosvc/sock
foo
You said: foo
httpd@vm-6858:~/lab$
```

You may find this tool helpful in debugging any new Unix domain socket services you create.

Exercise 4. Create a new service to perform transfer logging as a separate user ID. You will need to create a new service along the lines of `svc-echo.py`; modify `zook.conf` to start it appropriately (under a different UID); modify the permissions on the `transfer` database directory such that only this new service can modify it; and modify the `transfer.py` code to invoke this service to log transactions, instead of logging transactions directly.

Make all of your changes in the `lab` directory rather than in `/jail`. In particular, if you need to set certain permissions on files or directories, or install additional files or directories in `/jail`, do so in the `chroot-setup.sh` script.

Note: be careful when picking a format for messages in your service. What if someone tries to passes spaces or a newline as an argument? (Hint: use some existing encoding like JSON. But don't use Python's pickle module.)

Run `sudo make check` to verify that your privilege-separated transfer service passes our tests.

Now, you will break up the `zoobar` code into two additional protection domains. First, we want to make sure that only the transfer code is actually able to modify the zoobar balances of ~~different~~ users, so that a vulnerability in the rest of the python code will not be able to directly modify the number of zoobars that some user has.

One complication in doing this rests in the fact that the zoobar balance information is stored in the same database table, `person`, that stores profile and login information that the rest of the code must be able to modify. To protect zoobar balances from being corrupted by the rest of the python code, you will need to create a new database table holding just the zoobar balances for each user, and remove the balance information from the `person` table.

Exercise 5. In preparation for privilege-separating the transfer code, split the `zoobars` field from the `person` table into a new `zoobars` table stored in the database file `zoobar/db/zoobars/zoobars.db`, and remove the `zoobars` column from the `person` table. Change the rest of the python code to access the correct table when fetching zoobar balances. Don't forget to handle the case of account creation, when the new user needs to get an initial 10 zoobars.

Exercise 6. Create a new service to transfer zoobars from one user to another. Change the `transfer.py` code to invoke this service instead of modifying the zoobar balances directly. Set the permissions on the new balances table such that only the transfer code can modify it, and the rest of the python code can only read it. Don't forget to handle account creation, which needs to involve your new transfer service.

Finally, make sure that only the transfer code is able to invoke the logging service -- after all, no other python code should be able to generate log entries! You should be able to do this by using groups and group permissions on the directory containing the logging service socket. As before, make sure all of your changes are reflected in the `chroot-setup.sh` script, and not only in the `/jail` directory in your VM.

the linux file permissions in class

Now our web application should be more secure, because compromises of most of the python code will not allow the attacker to modify zoobar balances. Unfortunately, the attacker can still subvert the web site by modifying user passwords or HTTP cookies in the `person` database table. For the final part of this assignment, you will move the authentication and cookie-verification code into a separate service that runs under a distinct user ID, to prevent such attacks.

Exercise 7. Split the authentication information (`password`, `salt`, and `token` fields) into an `auth` table that is separate from the original `person` table. Store this table in the database file `zoobar/db/auth/auth.db`. After you do this, the only remaining fields in the `person` table should be the `username` and the `profile`.

Create a new service that implements user login and cookie verification using this table. This service should implement three functions, which correspond to existing functions implemented in `auth.py` that you will need to replace. First, check the username and password for login, returning an HTTP cookie token if the password is correct. Second, verify whether a token is correct, returning true or false. Third, register a new user, again returning true or false depending on success.

Make sure that the `auth` table storing passwords and tokens is only readable by your new authentication service.

Now that the attacker cannot obtain anyone's passwords or HTTP cookies from the database, there is one last problem to fix.

Exercise 8. In the current design, the attacker can still invoke the transfer service and ask for credits to be transferred between an arbitrary pair of users. Modify the transfer service protocol to require a valid token from the sender of credits, and modify the transfer service implementation to validate this token with the authentication service before approving the transfer.

Although `make check` does not include an explicit test for this exercise, you should be able to check whether this feature is working or not by manually connecting to your transfer service using `sudo nc -U /jail/.../sock`, and verifying that it is not possible to perform a transfer without supplying a valid token.

Deliverables

Again, explain in `answers.txt` any non-obvious changes you made to `zookws` and `zoobar` for each exercise. Feel free to include any comments about your solutions in the `answers.txt` file.

You are done! Run `make submit` to submit your answers to the the submission web site.

Acknowledgments

Thanks to Stanford's CS155 course staff for the initial zoobar web application code, which we extended in this lab assignment.

Collaboration

```
#!/bin/sh -x
if id | grep -qv uid=0; then
    echo "Must run setup as root"
    exit 1
fi

create_socket_dir() {
    local dirname="$1"
    local ownergroup="$2"
    local perms="$3"

    mkdir -p $dirname
    chown $ownergroup $dirname
    chmod $perms $dirname
}

set_perms() {
    local ownergroup="$1"
    local perms="$2"
    local pn="$3"

    chown $ownergroup $pn
    chmod $perms $pn
}

mkdir -p /jail
cp -p index.html /jail
cp -p password.cgi /jail

./chroot-copy.sh zookd /jail
./chroot-copy.sh zookfs /jail
./chroot-copy.sh zooksvc /jail

#./chroot-copy.sh /bin/bash /jail

./chroot-copy.sh /usr/bin/env /jail
./chroot-copy.sh /usr/bin/python /jail

mkdir -p /jail/usr/lib/
cp -r /usr/lib/python2.6 /jail/usr/lib
cp -r /usr/lib/pymodules /jail/usr/lib
cp /usr/lib/libsqlite3.so.0 /jail/usr/lib

mkdir -p /jail/usr/local/lib/
cp -r /usr/local/lib/python2.6 /jail/usr/local/lib

mkdir -p /jail/etc
cp /etc/localtime /jail/etc/
cp /etc/timezone /jail/etc/

mkdir -p /jail/usr/share/zoneinfo
cp -r /usr/share/zoneinfo/America /jail/usr/share/zoneinfo/
```

```

create_socket_dir /jail/echosvc 61010:61010 755

mkdir -p /jail/tmp
chmod a+rwxt /jail/tmp

cp -r zoobar /jail/

python /jail/zoobar/zoodb.py init-person
python /jail/zoobar/zoodb.py init-transfer


```

6.858 Lab2

9/27
8:55PM

Committed

Should have done earlier

¿Is this our fixed lab code

Don't get what Git did

¿Did we use make setup last time?

Edit chroot-setup.sh instead

It appears zookld has changed since last update

¿view diff to see their changes?

✓ Ran

Won't look at yet

Play w/ app

- register user
- transfer credits (zoobars)
- lookup zoobar balance, profile, transaction history

Who are the other users?

Where is the backend db

↳ not a concern now I think

Part 1 Web Server Setup

Read zookws web server code

└ same as last time?

Where is simple_svc?

└ not in any code files

in zook.conf - only mention on whole file

Oh that is actually <u>zookfs</u>

dir = /jail

└ w/ chroot

└ runs command or interactive shell w/ special root directory

So one article says not really secure
 Ls too hard

---

chroot-copy.sh copies stuff into /jail

```
echosvc
etc
index.htm
lib └python
password.cgi
tmp
usr
  zoobar
  zookd
   zookfs
   zooksvc
```

39M

How much do people do this in pratice
on real websites
  – besides OkWS

zookld still under root

↳ so can bind to pt 80

zookd is running under root

↳ so fix it!

Exercise 1. Modify config files for zookd + other services

(what ones?

So in zook.conf

Can set uids

How get list of uids?

Set to httpd?

(5)

less /etc/passwd

root	0
daemon	1
bin	2
sys	3
www-data	33
nobody	65534
httpd	1000

¿Where is zookd running under?

Permissions set when copied

Normally httpd - so do it?

make setup

↳ must quit server 1st

Can't save — access db?

Have chares take effect?

~~Ok~~: Users get deleted each time

? Do we jst type it in?

Conf w/ Jwang → So 2 fold

1. Permissions when run in zookld
   ↳ from conf
2. Actual files

does 2 matter?

And do we run the /jail vesion?
↳ no error no loader...

? Run as httpd or new user?
Might as well new user
Try not 1st

2

Running test

⊗ Zookd is running as root

Oh didn't copy

Now didn't run

exec svc 4 permission denied

in 61010

Ok set ~~[illegible]~~ file permission to true

Not setting file permission - still runs?
Or not

but no error on ~~[illegible]~~ start up

but no pg load

W/ file permission check gives error
but seems to work in browser

8

Oh fails when writing the db

What the * does password.cgi do?

Or is password just in memory

Are py errors logged anywhere?

from flask? ∟ micro framework

But not persistent

So keep file non editable

~~else~~ non owned by cook

except db

So is ~~[illegible]~~ sqlite directory

The /zoobar/db directory

Make 777 or owned by httpd?

After file creation!

9

Ok files changed

Why no work¿

even w/ 777 server error

¿any logging¿

Oh its a db

✓ works

✗ but still fails

Why does it work for me, but not the test

Note on Piazza @ 77

Did you make any manual changes to /jail

I don't think so...

tail /tmp/zookld.out

Ah there is the error

10

So clearing my /jail gives an error

→ Can't open db

Who is it running as?

777 fails too

? set dir as well?

✓ works

✓ Doesn't crash

Fail: zookd + zookfs w/ same uid

So they should be more seperate?

So can't interfer?

✓ Pass exercise 1

(that was more complicated then it looked!)

aug 11

Exercise 2

Try to furter priv seperate the simple_svc service that handles

Since a dynamic script cold delete static files

But aren't those files not writable

Asked on Piazza @98

So split into 2 → static
→ dynamic

Should only run index.cgi

Not password.cgi

└ is this file needed at all?
just delete it!

~~Or be an~~

Or be able to fetch persons.db /transfer.db
over the web

Oh it prints password.cgi

And tries to execute the .db file

[illegible] 12

└ but an attacker still can't read it

So what is going wrong?

lol lab code won't let you remove or re~~un~~set 755

- stupid

---

2 new HTTP services

URL filtering

Oh I need to ~~re~~ write

How to match multiple?

~~Which ones to pick~~ ?

What regex format do they use?

All go to ~~A~~ index.cgi

But what changes in main code

we have http-~~servi~~ svcs

13

So how do we tell it to do both?

This a dumb formatting q....

Asked ⓤ 99

no one else had a problem...

Oh I see → parse from (

Think before asking

✓ 2 services started

But no static or dynamic files served

— regex looks correct...

regex tester...

$\underbrace{.}_{\text{any char}}\underbrace{*}_{\infty\ \text{times}}$

✓ img working

Need trailing / on URL

dispatches - but does not show anything..

↳ is it since db wrong?

(14)

¿why did it break the other?

The fs ~~file~~ now has permission denied
from 100~~0~~3 to 1002

Can't chroot that - why?
~~own process~~ ?
process reuse?

What did I change since before?
Or it never worked...?
But after 1st one

Had this permission ~~error~~ denied error earlier
└ oh must update .sh!
Copying it twice
~~[illegible]~~
└ Nope must be directory

[illegible] 15

So usr 1002
grp 1003 ?

how does that scale?

✓ img worked

✓ dynamic worked
Site still seems to work
? So try make check

⊗ password cgi accessible

↳ # zoobar/../password.cgi

So regex for this
likely from media
but code doesn't compile

No the ; in file → index.cgi
Hmm

Nov 16

Or specific media files

Not truly correct

↳ do for other

untested - but do

Stillwise checker errors

site works!

Something about checking vids

Hmm sometimes does fail - why?

Socket IO Error

↳ read line too long?

Oh w/ no slash

✓ Pass ex 2

17

Exercise 3

Now start fixing application code w/ transfer.py

1. - Zombies
2. Fail to log
3. Not atomic

Deliverable

Must write them all up

So since we run as non root - other files non ~~[illegible]~~ writable

Prob could tighten up more

To be extra conservative....

✓ Handed in

```python
from flask import g, render_template, request
import time

from login import requirelogin
from zoodb import *
from debug import *

@catch_err
@requirelogin
def transfer():
    warning = None
    try:
        if 'recipient' in request.form:
            recipient = g.persondb.query(Person).get(request.form['recipient'])
            zoobars = eval(request.form['zoobars'])
            sender_balance = g.user.person.zoobars - zoobars
            recipient_balance = recipient.zoobars + zoobars

            if sender_balance < 0 or recipient_balance < 0:
                raise ValueError()

            g.user.person.zoobars = sender_balance
            recipient.zoobars = recipient_balance
            transfer = Transfer()
            transfer.sender = g.user.person.username
            transfer.recipient = recipient.username
            transfer.amount = zoobars
            transfer.time = time.asctime()
            g.transferdb.add(transfer)
            warning = "Sent %d zoobars" % zoobars
    except (KeyError, ValueError, AttributeError) as e:
        log("Transfer exception: %s" % str(e))
        warning = "Transfer to %s failed" % request.form['recipient']

    return render_template('transfer.html', warning=warning)
```

? get #

~~does not Check balance~~

← or yes it does

then does this

commits 1st

Can be neg

----

lab2: exercise 1

I set zookfs service to execute as a different user in zook.conf. I had to change the file permissions in /jail so these new users could read these files, but not change them; which I did in chroot-setup.sh. I made the file readable only by this user. I didn't change the /zoobar folder, since those were already readable by this new user. I also had to give this user writing permissions on the two database files.

----

lab2: exercise 2

I split the two services up in zook.conf. I had to change the file permissions again in chroot-setup.sh to make the file accessible by the new user id. To prevent access to the very stupid password.cgi file (as well as other files) I listed all of the valid files in zook.conf using regex. I don't know how this hopes to scale though.

----

lab2: exercise 3

The number of zoobars can be negitive to steal the positive balance of other zoobars

Does not commit the log as an atomic unit - transfer could fail to record, but transfer still happened.

Does not commit the zoobars as an atomic unit - subject to race condition on multiple requests in short time frame

----

lab2: part1 feedback

I would have better explained the user permission thing. There was a bit of exploring best pratices here - but I feel that best practices should be explained, not discovered. Now that I see the answer, it makes sense.

Static Detection of Security Vulnerabilities in Scripting Languages

Read 9/30

Yichen Xie Alex Aiken
Computer Science Department
Stanford University
Stanford, CA 94305
{yxie,aiken}@cs.stanford.edu

date?

ABSTRACT

Cool

We present a static analysis algorithm for detecting security vulnerabilities in PHP, a popular server-side scripting language for building web applications. Our analysis employs a novel three-tier architecture to capture information at decreasing levels of granularity at the intrablock, intraprocedural, and interprocedural level. This architecture enables us to handle dynamic features unique to scripting languages such as dynamic typing and code inclusion, which have not been adequately addressed by previous techniques.

We demonstrate the effectiveness of our approach by running our tool on six popular open source PHP code bases and finding 105 previously unknown security vulnerabilities, most of which we believe are remotely exploitable.

?I [illegible] know what this means

1. INTRODUCTION

Web-based applications have experienced exponential growth during the past few years and have become the *de facto* standard for delivering online services ranging from discussion forums to security sensitive areas such as banking and retailing. As such, security vulnerabilities in these applications represent an increasing threat to both the providers and the users of such services. During the second half of 2004, Symantec cataloged 670 vulnerabilities affecting web applications, an 81% increase over the same period in 2003 [16]. This trend is likely to continue for the foreseeable future.

According to the same report, these vulnerabilities are typically caused by programming errors in input validation and improper handling of submitted requests [16]. Since vulnerabilities are usually deeply embedded in the program logic, traditional network-level defense (e.g., firewalls) does not offer adequate protection against such attacks. Testing is also largely ineffective because attackers typically use the least expected input to exploit these vulnerabilities and compromise the system.

in the Command Code

A natural alternative is to find these errors using static analysis, but it is widely believed that scripting languages are too difficult to analyze statically. The main message of this paper is that this folk wisdom is false: we show by example that a static analysis, suitably designed to address the unique aspects of scripting languages, can identify many serious security vulnerabilities in scripts. Given the importance of scripting in real world applications, we believe there is an opportunity for static analysis to have a significant impact in this new domain.

In this paper, we apply static analysis to finding security vulnerabilities in PHP, a server-side scripting language that has become one of the most widely adopted platforms for developing web applications[1]. Our goal is a bug detection tool that automatically finds serious vulnerabilities with high confidence. This work, however, does not aim to verify the absence of bugs.

This paper makes the following contributions:

- We present an interprocedural static analysis algorithm for PHP. A language as dynamic as PHP presents unique challenges for static analysis: language constructs (e.g., include) that allow dynamic inclusion of program code, variables whose types change during execution, operations with semantics that depend on the runtime types of the operands (e.g., <), pervasive use of hash tables and regular expression matching are just some features that must be modeled well to produce useful results.

  To faithfully model program behavior in such a language, we use a unique three-tier analysis that captures information at decreasing levels of granularity at the intrablock, intraprocedural, and interprocedural levels. For example, we use symbolic execution to model dynamic features inside basic blocks and use block summaries to hide that complexity from intra- and inter-procedural analysis. We believe the same techniques can be easily applied to other scripting languages (e.g., Perl). To the best of our knowledge, this paper is the first to recognize and model complex program features that are specific to scripting languages.

I have no clue how it works internally

- We show how to use our static analysis algorithm to find SQL injection vulnerabilities. Once configured, the analysis is fully automatic. Although we focus on SQL injections in this work, we believe that, with small modifications, the same techniques can be applied to detecting other vulnerabilities such as cross site scripting (XSS) and code injection in web applications.
- We experimentally validate our approach by implementing the analysis algorithm and running it on six popular web applications written in PHP. Our tool found 105 previously unknown security vulnerabilities. We further investigated two reported vulnerabilities in PHP-fusion, a mature, widely deployed content management system, and constructed exploits for both that allow an attacker to control or damage the system.

[1] Installed on over 23 million Internet domains [13], and is ranked fourth on the TIOBE programming community index [17].

The rest of the paper is organized as follows. We start with a brief introduction to PHP and show examples of SQL vulnerabilities in web application code (Section 2). We then describe our analysis algorithm in detail and show how we use it to find SQL injection vulnerabilities (Section 3). Section 4 describes the implementation and experimental results and show two case studies of exploitable vulnerabilities in PHP-fusion. Section 5 discusses related work, and Section 6 concludes.

2. BACKGROUND

This section briefly introduces the PHP language and shows examples of SQL injection vulnerabilities in PHP.

PHP was created a decade ago by Rasmus Lerdorf as a simple set of Perl scripts for tracking accesses to his online resume. It has since evolved into one of the most popular server-side scripting languages for building web applications. According to a recent Security Space survey, PHP is installed on 44.6% of Apache web servers [15], adopted by millions of developers, and used or supported by Yahoo, IBM, Oracle, and SAP, among others [13].

Although the PHP language has undergone two major redesigns over the past decade, it retains a Perl-like syntax and dynamic (interpreted) nature, which contributes to its most cited advantage of being simple and flexible.

PHP has a suite of programming constructs and special operations that makes web development easy. We give three examples below:

1. **Natural integration with SQL:** PHP provides nearly native support for database operations. For example, using inline variables in strings, most SQL queries can be concisely expressed with a simple function call

   ```
   $rows=mysql_query("UPDATE users SET
        pass='$pass' WHERE userid='$userid'");
   ```

   Contrast this code with Java, where a database is typically accessed through *prepared statements*: one creates a statement template and fills in the values (along with their types) using *bind variables*:

   ```
   PreparedStatement s = con.prepareStatement
     ("UPDATE users SET pass = ?
       WHERE userid = ?");
   s.setString(1, pass); s.setInt(2, userid);
   int rows = s.executeUpdate();
   ```

2. **Dynamic types and implicit casting to and from strings:** PHP, like other scripting languages, has extensive support for string operations and automatic conversions between strings and other types. These features are handy for web applications because strings serve as the common medium between the browser, the web server, and the database backend. For example, we can convert a number into a string without an explicit cast:

   not much to worry about

   ```
   if ($userid < 0) exit;
   $query = "SELECT * from users
             WHERE userid = '$userid'";
   ```

3. **Variable scoping and the environment:** PHP has a number of mechanisms that minimize redundancy when accessing values from the execution environment. For example, HTTP *get* and *post* requests are automatically imported into the global name space as hash tables $_GET and $_POST. To access the "name" field of a submitted form, one can simply use variable $_GET['name'] directly in the program.

   If this still sounds like too much typing, PHP provides an extract operation that automatically imports all key-value pairs of a hash table into the current scope. In the example above, one can use extract(_GET, EXTR_OVERWRITE) to import data submitted using the HTTP get method. To access the "name" field, one now simply types $name, which is preferred by some to $_GET['name'].

However, these conveniences come with security implications:

1. **SQL injection made easy:** bind variables in Java have the benefit of assuring the programmer that any data passed into a SQL query remains data. The same cannot be said for the PHP example where malformed data from a malicious attacker may change the meaning of a SQL statement and cause unintended operations to the database. These are commonly called *SQL injection* attacks.

   In the example above (case 1), suppose $userid is controlled by the attacker and has value

   ```
   ' OR '1' = '1
   ```

   The query string becomes

   ```
   UPDATE users SET pass='...'
   WHERE userid='' OR '1'='1'
   ```

   which has the effect of updating the password for all users in the database.

2. **Unexpected conversions:** Consider the following code:

   ```
   if ($userid == 0) echo $userid;
   ```

   One would expect that if the program prints anything, it should be "0". Unfortunately, PHP implicitly casts string values into numbers before comparing them with an integer. Non-numerical values (e.g. "abc") convert to 0 without complaint, so the code above can print anything other than a non-zero number. We can imagine a potential SQL injection vulnerability if $userid is subsequently used to construct a SQL query as in the previous case.

   nice

3. **Uninitialized variables under user control:** In PHP, uninitialized variable defaults to null. Some programs rely on this fact for correct behavior; consider the following code:

   ```
   1  extract($_GET, EXTR_OVERWRITE);
   2  for ($i=0;$i<=7;$i++)
   3     $new_pass .= chr(rand(97, 122));
   4  mysql_query("UPDATE ... $new_pass ...");
   ```

   This program generates a random password and inserts it into the database. However, due to the extract operation on line 1, a malicious user can introduce an arbitrary initial value for $new_pass by adding an unexpected new_pass field into the submitted HTTP form data.

```
CFG := build_control_flow_graph(AST);
foreach (basic_block b in CFG)
  summaries[b] := simulate_block(b);
return make_function_summary(CFG, summaries);
```

Figure 1: Pseudo-code for the analysis of a function.

3. ANALYSIS

Given a PHP source file, our tool carries out static analysis in the following steps:

- We parse the PHP source into abstract syntax trees (ASTs). Our parser is based on the standard open-source implementation of PHP 5.0.5 [12]. Each PHP source file contains a *main* section (referred to as the *main* function hereon although it is not part of any function definition) and zero or more user-defined functions. We store the user-defined functions in the environment, and start the analysis from the main function.

- For each function in the program, the analysis performs a standard conversion from the abstract syntax tree (AST) of the function body into a control flow graph (CFG). The nodes of the CFG are maximal *basic blocks*: single entry, single exit sequences of statements. The edges of the CFG are the jump relationships between blocks. For conditional jumps, the corresponding CFG edge is labeled with the branch predicate.

- Each basic block is simulated using symbolic execution. The goal is to understand the collective effects of statements in a block on the global state of the program, and summarize their effects into a concise *block summary* (which describes, among other things, the set of variables that must be sanitized before entering the block). We describe the simulation algorithm in Section 3.1.

- After computing a summary for each basic block, we use a standard reachability analysis to combine block summaries into a *function summary*. The function summary describes the pre- and post-conditions of a function (e.g., the set of sanitized input variables after calling the current function). We discuss this step in Section 3.2.

- During the analysis of a function, we might encounter calls to other user-defined functions. We discuss modeling function calls, and the order in which functions are analyzed, in Section 3.3.

3.1 Simulating Basic Blocks

3.1.1 Outline

Figure 2 gives pseudo-code outlining the symbolic simulation process. Recall each basic block contains a linear sequence of statements with no jumps or jump targets in the middle. The simulation starts in an *initial state*, which maps each variable x to a symbolic initial value x_0. It processes each statement in the block in order, updating the simulator state to reflect the effect of that statement. The simulation continues until it encounters any of the following:

```
function simulate_block(BasicBlock b) : BlockSummary
{
  state := init_simulation_state();
  foreach (Statement s in b) {
    state := simulate(s, state);
    if (state.has_returned || state.has_exited)
      break;
  }
  summary := make_block_summary(state);
  return summary;
}
```

Figure 2: Pseudo-code for intra-block simulation.

$$
\begin{array}{rcl}
\text{Type } (\tau) & ::= & \text{str} \mid \text{bool} \mid \text{int} \mid \bot \\
\text{Const } (c) & ::= & \text{string} \mid k \mid \text{true} \mid \text{false} \mid \text{null} \\
\text{L-val } (lv) & ::= & x \mid \texttt{Arg\#i} \mid l[e] \\
\text{Expr } (e) & ::= & c \mid lv \mid e \text{ binop } e \mid \text{unop } e \mid (\tau)e \\
\text{Stmt } (S) & ::= & lv \leftarrow e \mid lv \leftarrow f(e_1, \ldots, e_n) \\
 & & \mid \text{return } e \mid \text{exit} \mid \text{include } e
\end{array}
$$

$$
\begin{array}{rcl}
\text{binop} & \in & \{+, -, \text{concat}, ==, != , <, >, \ldots\} \\
\text{unop} & \in & \{-, \neg\}
\end{array}
$$

Figure 3: Language Definition

1. the end of the block;

2. a return statement. In this case, the current block is marked as a "return" block, and the simulator evaluates and records the return value;

3. an exit statement. In this case the current block is marked as an "exit" block;

4. a call to a user-defined function that exits the program. This condition is automatically determined using the function summary of the callee (see Sections 3.2 and 3.3).

Note that in the last case, execution of the program has effectively terminated and therefore we remove any ensuing statements and outgoing CFG edges from the current block.

After a basic block is simulated, we use information contained in the final state of the simulator to summarize the effect of the block into a *block summary*, which we store for use during the intraprocedural analysis (see Section 3.2). The state itself is discarded after simulation.

The following subsections describe the simulation process in detail. We start with a definition of the subset of PHP that we currently model (§3.1.2) and discuss the representation of the simulation state and program values (§3.1.3, §3.1.4) during symbolic execution. Using the value representation, we describe how the analyzer simulates expressions (§3.1.5) and statements (§3.1.6). Finally, we describe how we represent and infer block summaries (§3.1.7).

3.1.2 Language

Figure 3 gives the definition of a small imperative language that captures a subset of PHP constructs that we believe is relevant to SQL injection vulnerabilities. Like PHP, the language is dynamically typed. We model three basic types of PHP values: strings, booleans and integers. In addition, we introduce a special ⊥ type to describe objects whose static types are undetermined (e.g. input parameters, etc.).[2]

Expressions can be *constants*, *l-values*, *unary* and *binary operations*, and *type casts*. The definition of l-values is worth mentioning because in addition to variables and function parameters, we include a named subscript operation to give limited support to the array and hash table accesses that are used extensively in PHP programs.

A statement can be an *assignment*, *function call*, *return*, *exit*, or *include*. The first four types of statement require no further explanation. The include statement is a commonly used feature unique to scripting languages, which allows programmers to dynamically insert code into the program. In our language, include evaluates its string argument, and executes the program file designated by the string as if it is inserted at that program point (e.g., it shares the same scope). We describe how we simulate such behavior in Section 3.1.6. Oh

3.1.3 State

Figure 4(a) gives the definition of values and states during simulation. The simulation state maps memory locations to their value representations, where a memory location is either a program variable (e.g. x), or an entry in a hash table accessed via another location (e.g. $x[key]$).

On entry to the function, each location l is implicitly initialized to a symbolic initial value l_0, which makes up the initial state of the simulation. The values we represent in the state can be classified into three categories based on type:

Strings: Strings are the most fundamental type in many scripting languages, and precision in modeling strings directly determines the analysis precision. Strings are typically constructed through concatenation. For example, user inputs (via HTTP get and post methods) are often concatenated with a pre-constructed skeleton to form a SQL query. Similarly, results from the query can be concatenated with HTML templates to form output. Modeling concatenation well enables an analysis to better understand information flow in a script. Thus, our representation of a string is based on the concept of concatenation: string values are represented as an ordered concatenation of *string segments*, which can be one of the following: a string constant, the initial value of a memory location on entry to the current block (l_0), or a string that contains initial values of zero or more elements from a set of memory locations (contains(σ)). We use the last representation to model return values from function calls, which may non-deterministically contain a combination of global variables and input parameters. For example, in

```
1 function f($a, $b) {
2     if (...) return $a;
3     else return $b;
4 }
5 $ret = f($x.$y, $z);
```

we represent the return value on line 5 as contains({x, y, z}) to model the fact that it may contain any element in the set as a sub-string.

The string representation described above has the following benefits:

$\boxed{\text{Value Representation}}$

$$\begin{aligned}
\text{Loc } (l) &::= x \mid l[\text{string}] \mid l[\bot] \\
\text{Init-Values } (o) &::= l_0 \\
\text{Segment } (\beta) &::= \text{string} \mid \text{contains}(\sigma) \\
\text{String } (s) &::= [\beta_1, \ldots, \beta_n] \\
\text{Boolean } (b) &::= \text{true} \mid \text{false} \mid \text{untaint}(\sigma_0, \sigma_1) \\
\text{Loc-set}(\sigma) &::= \{l_1, \ldots, l_n\} \\
\text{Integer } (i) &::= k \\
\text{Value } (v) &::= s \mid b \mid i \mid o \mid \bot
\end{aligned}$$

$\boxed{\text{Simulation State}}$

$\text{State } (\Gamma) : \text{Loc} \rightarrow \text{Value}$

(a) Value representation and simulation state.

$\boxed{\text{Locations}}$

$$\frac{}{\Gamma \vdash x \overset{\text{Lv}}{\Rightarrow} x}\ \text{var} \qquad \frac{}{\Gamma \vdash \texttt{Arg\#n} \overset{\text{Lv}}{\Rightarrow} \texttt{Arg\#n}}\ \text{arg}$$

$$\frac{\begin{array}{l}\Gamma \vdash e \overset{\text{E}}{\Rightarrow} l_0 \\ \Gamma \vdash e' \overset{\text{E}}{\Rightarrow} v' \quad v'' = \text{cast}(v', \text{str})\end{array}}{\Gamma \vdash e[e'] \overset{\text{Lv}}{\Rightarrow} \begin{cases} l[\alpha] & \text{if } v'' = [\text{“}\alpha\text{”}] \\ l[\bot] & \text{otherwise} \end{cases}}\ \text{dim}$$

(b) L-values.

$\boxed{\text{Expressions}}$

Type casts:

$$\text{cast}(k, \text{bool}) = \begin{cases} \text{true} & \text{if } k \neq 0 \\ \text{false} & \text{otherwise} \end{cases}$$

$$\text{cast}(\text{true}, \text{str}) = [\text{“1”}]$$

$$\text{cast}(\text{false}, \text{str}) = []$$

$$\begin{aligned}&\text{cast}(v = [\beta_1, \ldots, \beta_n], \text{bool}) \\ &\quad = \begin{cases} \text{true} & \text{if } (v \neq [\text{“0”}]) \wedge \bigvee_{i=1}^{n} \neg\text{is\_empty}(\beta_i) \\ \text{false} & \text{if } (v = [\text{“0”}]) \vee \bigwedge_{i=1}^{n} \text{is\_empty}(\beta_i) \\ \bot & \text{otherwise} \end{cases}\end{aligned}$$

...

Evaluation Rules:

$$\frac{\Gamma \vdash lv \overset{\text{Lv}}{\Rightarrow} l}{\Gamma \vdash lv \overset{\text{E}}{\Rightarrow} \Gamma(l)}\ \text{L-val}$$

$$\frac{\begin{array}{l}\Gamma \vdash e_1 \overset{\text{E}}{\Rightarrow} v_1 \quad \text{cast}(v_1, \text{str}) = [\beta_1, \ldots, \beta_n] \\ \Gamma \vdash e_2 \overset{\text{E}}{\Rightarrow} v_2 \quad \text{cast}(v_2, \text{str}) = [\beta_{n+1}, \ldots, \beta_m]\end{array}}{\Gamma \vdash e_1 \text{ concat } e_2 \overset{\text{E}}{\Rightarrow} [\beta_0, \ldots, \beta_m]}\ \text{concat}$$

$$\frac{\Gamma \vdash e \overset{\text{E}}{\Rightarrow} v \quad \text{cast}(v, \text{bool}) = v'}{\Gamma \vdash \neg e \overset{\text{E}}{\Rightarrow} \begin{cases} \text{true} & \text{if } v' = \text{false} \\ \text{false} & \text{if } v' = \text{true} \\ \text{untaint}(\sigma_1, \sigma_0) & \text{if } v' = \text{untaint}(\sigma_0, \sigma_1) \\ \bot & \text{otherwise} \end{cases}}\ \text{not}$$

(c) Expressions.

Figure 4: Intrablock simulation algorithm.

[2] In general, in a dynamically typed language, a more precise static approximation in this case would be a sum (aka. soft typing) [1, 19]. We have not found it necessary to use type sums in this work.

Q How are scripting languages "compiled"?

First, we get automatic constant folding for strings within the current block, which is often useful for resolving hash keys and distinguishing between hash references (e.g., in $key = "key"; return $hash[$key];).

☆ Second, we can track how the contents of one input variable flow into another by finding occurrences of initial values of the former in the final representation of the latter. For example, in: $a = $a . $b, the final representation of $a is $[a_0, b_0]$. We know that if either $a or $b contains unsanitized user input on entry to the current block, so does $a upon exit.

Finally, interprocedural dataflow is possible by tracking function return values based on function summaries using contains(σ). We describe this aspect in more detail in Section 3.3.

Booleans: In PHP, a common way to perform input validation is to call a function that returns true or false depending on whether the input is well-formed or not. For example, the following code sanitizes $userid:

```
    $ok = is_numeric($userid);
    if (!$ok) exit;
```

The value of Boolean variable $ok after the call is undetermined, but it is correlated with the validity of $userid. This motivates untaint(σ_0, σ_1) as a representation for such Booleans: σ_0 (resp. σ_1) represents the set of validated l-values when the Boolean is false (resp. true). In the example above, $ok has representation untaint({}, {userid}).

Besides untaint, representation for Booleans also include constants (true and false) and unknown ($\perp$).

Integers: Integer operations are relatively less emphasized in our simulation. We track integer constants and binary and unary operations between them. We also support type casts from integers to Boolean and string values.

So what do we do w/ this

3.1.4 *Locations and L-values*

don't get

In the language definition in Figure 3, hash references may be aliased through assignments and l-values may contain hash accesses with non-constant keys. The same l-value may refer to different memory locations depending on the value of both the host and the key, and therefore, l-values are not suitable as memory locations in the simulation state.

Figure 4(b) gives the rules we use to resolve l-values into memory locations. The var and arg rules map each program variable and function argument to a memory location identified by its name, and the dim rule resolves hash accesses by first evaluating the hash table to a location and then appending the key to form the location for the hash entry.

These rules are designed to work in the presence of simple aliases. Consider the following program:

```
  1  $hash = $_POST;
  2  $key = 'userid';
  3  $userid = $hash[$key];
```

The program first creates an alias ($hash) to hash table $_POST and then accesses the userid entry using that alias. On entry to the block, the initial state maps every location to its initial value:

$$\Gamma = \{\mathsf{hash} \Rightarrow \mathsf{hash}_0, \mathsf{key} \Rightarrow \mathsf{key}_0, \mathsf{\_POST} \Rightarrow \mathsf{\_POST}_0, \mathsf{\_POST[userid]} \Rightarrow \mathsf{\_POST[userid]}_0\}$$

According to the var rule, each variable maps to its own unique location. After the first two assignments, the state is:

$$\Gamma = \{\mathsf{hash} \Rightarrow \mathsf{\_POST}_0, \mathsf{key} \Rightarrow [\text{`userid'}], \ldots\}$$

We use the dim rule to resolve $hash[$key] on line 3: $hash evaluates to $\mathsf{\_POST}_0$, and $key evaluates to constant string 'userid'. Therefore, the l-value $hash[$key] evaluates to location _POST[userid], and thus the analysis assigns the desired value $\mathsf{\_POST[userid]}_0$ to $userid.

3.1.5 *Expressions*

We perform abstract evaluation of expressions based on the value representation described above. Because PHP is a dynamically typed language, operands are automatically cast into appropriate types for binary and unary operations in an expression. Figure 4(c) gives a representative subset of cast rules that simulates cast operations in PHP. For example, Boolean value true, when used in a string context, evaluates to "1". false, on the other hand, is converted to the empty string instead of "0". In cases where exact representation is not possible, the result of the cast is unknown ($\perp$).

Figure 4(c) also gives three representative rules for evaluating expressions. The first rule handles l-values, and the result is obtained by first resolving the l-value into a memory location, and then looking up the location in the evaluation context (recall that $\Gamma(l) = l_0$ on entry to the block).

The second rule models string concatenation. We first cast the value of both operands into string values, and the result is the concatenation of both.

The final rule handles Boolean negation. The interesting case involves untaint values. Recall that untaint(σ_0, σ_1) denotes an unknown Boolean value that is false (resp. true) if l-values in the set σ_0 (resp. σ_1) are sanitized. Given this definition, the negation of untaint(σ_0, σ_1) is untaint(σ_1, σ_0).

The analysis of an expression is $\perp$ if we cannot determine a more precise representation, which is a potential source of false negatives.

3.1.6 *Statements*

We model assignments, function calls, return, exit, and include statements in the program. The assignment rule resolves the left-hand side into a memory location l, and evaluates the right-hand side into a value v. The updated simulation state after the assignment maps l to the new value v:

$$\frac{\Gamma \vdash lv \overset{\mathsf{Lv}}{\Rightarrow} l \qquad \Gamma \vdash e \overset{\mathsf{E}}{\Rightarrow} v}{\Gamma \vdash lv \leftarrow e \overset{\mathsf{S}}{\Rightarrow} \Gamma[l \mapsto v]} \ \mathsf{assignment}$$

Function calls are similar. The return value of a function call $f(e_1, \ldots, e_n)$ is modeled using either contains(σ) (if f returns a string) or untaint(σ_0, σ_1) (if f returns a Boolean) depending on the inferred summary for f. We defer discussion of the function summaries and the return value representation to Sections 3.2 and 3.3. For the purpose of this section, we use the uninterpreted value $f(v_1, \ldots, v_n)$ as a place holder for the actual representation of the return value:

$$\frac{\Gamma \vdash lv \overset{\mathsf{Lv}}{\Rightarrow} l \quad \Gamma \vdash e_1 \overset{\mathsf{E}}{\Rightarrow} v_1 \ \ldots \ \Gamma \vdash e_n \overset{\mathsf{E}}{\Rightarrow} v_n}{\Gamma \vdash lv \leftarrow f(e_1, \ldots, e_n) \overset{\mathsf{S}}{\Rightarrow} \Gamma[l \mapsto f(v_1, \ldots, v_n)]} \ \mathsf{fun}$$

In addition to the return value, certain functions have pre- and post-conditions depending on the operation they perform. Pre- and post-conditions are inferred and stored in the callee's summary, which we describe in detail in Sections 3.2 and 3.3. Here we show two examples to illustrate their effects:

```
1  function validate($x) {
2   if (!is_numeric($x)) exit;
3   return;
4  }
5  function my_query($q) {
6    global $db;
7    mysql_db_query($db, $q);
8  }
9  validate($a.$b);
10 my_query("SELECT ... WHERE a = '$a' AND c = '$c'");
```

The validate function tests whether the argument is a number (thus safe) and aborts if it is not. Therefore, line 9 sanitizes both $a and $b. We record this fact by first inspecting the value representation of the actual parameter (in this case $[a_0, b_0]$), and remembering the set of non-constant segments that are sanitized.

The second function my_query uses the argument as a database query string. To prevent SQL injection attacks, we require that any user input be sanitized before it becomes part of the first parameter. Again, we enforce this requirement by inspecting the value representation of the actual parameter. We record any unsanitized non-constant segments (in this case $c, since $a is sanitized on line 9) and require they be sanitized as part of the pre-condition for the current block.

Sequences of assignments and function calls are simulated by using the output environment of the previous statement as the input environment of the current statement:

$$\frac{\Gamma \vdash s_1 \overset{S}{\Rrightarrow} \Gamma' \quad \Gamma' \vdash s_2 \overset{S}{\Rrightarrow} \Gamma''}{\Gamma \vdash (s_1; s_2) \overset{S}{\Rrightarrow} \Gamma''}\ \textsf{seq}$$

The final simulation state is the output state of the final statement.

The return and exit statements terminate control flow[3] and require special treatment. For a return, we evaluate the return value and use it in calculating the function summary. In case of an exit statement, we mark the current block as an *exit block*.

Finally, include statements are a commonly used feature unique to scripting languages allowing programmers to dynamically insert code and function definitions from another script. In PHP, the included code inherits the variable scope at the point of the include statement. It may introduce new variables and function definitions, and change or sanitize existing variables before the next statement in the block is executed.

We process include statements by first parsing the included file, and adding any new function definitions to the environment. We then splice the control flow graph of the main function at the current program point by a) removing the include statement, b) breaking the current basic block into two at that point, c) linking the first half of the current block to the start of the main function, and all return blocks (those containing a return statement) in the included CFG to the second half, and d) replacing the return statements in the included script with assignments to reflect the fact that control flow resumes in the current script.

3.1.7 Block summary

The final step for the symbolic simulator is to characterize the behavior of a CFG block into a concise summary. A block summary is represented as a six-tuple $\langle \mathcal{E}, \mathcal{D}, \mathcal{F}, \mathcal{T}, \mathcal{R}, \mathcal{U} \rangle$:

- **Error set ($\mathcal{E}$):** the set of input variables that must be sanitized before entering the current block. These are accumulated during simulation of function calls that require sanitized input.
- **Definitions ($\mathcal{D}$):** the set of memory locations defined in the current block. For example, in

  $a = $a.$b; $c = 123;

  we have $\mathcal{D} = \{a, c\}$.
- **Value flow ($\mathcal{F}$):** the set of pairs of locations (l_1, l_2) where the string value of l_1 on entry becomes a substring of l_2 on exit. In the example above, $\mathcal{F} = \{(a, a), (b, a)\}$.
- **Termination predicate ($\mathcal{T}$):** true if the current block contains an exit statement, or if it calls a function that causes the program to terminate.
- **Return value ($\mathcal{R}$):** records the representation for the return value if any, undefined otherwise. Note that if the current block has no successors, either $\mathcal{R}$ has a value or $\mathcal{T}$ is true.
- **Untaint set ($\mathcal{U}$):** for each successor of the current CFG block, we compute the set of locations that are sanitized if execution continues onto that block. Sanitization can occur via function calls, casting to safe types (e.g., int, etc), regular expression matching, and other tests. The untaint set for different successors might differ depending on the value of branch predicates. We show an example below.

  ```
  validate($a);
  $b = (int) $c;
  if (is_numeric($d))
    ...
  ```

  As mentioned earlier, validate exits if $a is unsafe. Casting to integer also returns a safe result. Therefore, the untaint set is $\{a, b, d\}$ for the true branch, and $\{a, b\}$ for the false branch.

3.2 Intraprocedural Analysis

Based on block summaries computed in the previous step, the intraprocedural analysis computes the following summary $\langle \mathcal{E}, \mathcal{R}, \mathcal{S}, \mathcal{X} \rangle$ for each function:

1. **Error set ($\mathcal{E}$):** the set of memory locations (variables, parameters, and hash accesses) whose value may flow into a database query, and therefore must be sanitized before invoking the current function. For the main function, the error set must not include any user-defined variables (e.g. $_GET['...'] or $_POST['...'])—the analysis emits an error message for each such violation.

   We compute $\mathcal{E}$ by a backwards reachability analysis that propagates the error set of each block (using the $\mathcal{E}, \mathcal{D}, \mathcal{F}$, and $\mathcal{U}$ components in the block summaries) to the start block of the function.

[3] So do function calls that exits the program, in which case we remove any ensuing statements and outgoing edges from the current CFG block. See Section 3.3.

2. **Return set ($\mathcal{R}$):** the set of parameters or global variables whose value may be a substring of the return value of the function. $\mathcal{R}$ is only computed for functions that may return string values. For example, in the following code, the return set includes both function arguments and the global variable $table (i.e. $\mathcal{R} = \{\mathsf{table}, \mathtt{Arg\#1}, \mathtt{Arg\#2}\}$).

   ```
   function make_query($user, $pass) {
     global $table;
     return "SELECT * from $table ".
       "where user = $user and pass = $pass";
   }
   ```

   We compute the function return set by using a forward reachability analysis that expresses each return value (recorded in the block summaries as $\mathcal{R}$) as a set of function parameters and global variables.

3. **Sanitized values ($\mathcal{S}$):** the set of parameters or global variables that are sanitized on function exit. We compute the set by using a forward reachability analysis to determine the set of sanitized inputs at each return block, and we take the intersection of those sets to arrive at the final result.

   If the current function returns a Boolean value as its result, we distinguish the sanitized value set when the result is true versus when it is false (mirroring the untaint representation for Boolean values above). The following example motivates this distinction:

   ```
   function is_valid($x) {
     if (is_numeric($x)) return true;
     return false;
   }
   ```

   The parameter is sanitized if the function returns true, and the return value is likely to be used by the caller to determine the validity of user input. In the example above,

   $$\mathcal{S} = (\mathsf{false} \Rightarrow \{\}, \mathsf{true} \Rightarrow \{\mathtt{Arg\#1}\})$$

   For comparison, the validate function defined previously has $\mathcal{S} = (* \Rightarrow \{\mathtt{Arg\#1}\})$. In the next section, we describe how we make use of this information in the caller.

4. **Program Exit ($\mathcal{X}$):** a Boolean which indicates whether the current function terminates program execution on all paths. Note that control flow can leave a function either by returning to the caller or by terminating the program. We compute the exit predicate by enumerating over all CFG blocks that have no successors, and identify them as either return blocks or exit blocks (the $\mathcal{T}$ and $\mathcal{R}$ component in the block summary). If there are no return blocks in the CFG, the current function is an exit function.

The dataflow algorithms used in deriving these facts are fairly standard fix-point computations. We omit the details for brevity.

3.3 Interprocedural Analysis

This section describes how we conduct interprocedural analysis using summaries computed in the previous step. Assuming f has summary $\langle \mathcal{E}, \mathcal{R}, \mathcal{S}, \mathcal{X} \rangle$, we process a function call $f(e_1, \ldots, e_n)$ during intrablock simulation as follows:

1. **Pre-conditions:** We use the error set ($\mathcal{E}$) in the function summary to identify the set of parameters and global variables that must be sanitized before calling this function. We substitute actual parameters for formal parameters in $\mathcal{E}$ and record any unsanitized non-constant segments of strings in the error set as sanitization pre-condition for the current block.

2. **Exit condition:** If the callee is marked as an exit function (i.e. $\mathcal{X}$ is true), we remove any statements that follow the call and delete all outgoing edges from the current block. We further mark the current block as an exit block.

3. **Post-conditions:** If the function unconditionally sanitizes a set of input parameters and global variables, we mark this set of values as safe in the simulation state after substituting actual parameters for formal parameters.

   If sanitization is conditional on the return value (e.g., the is_valid function defined above), we record the intersection of its two component sets as being unconditionally sanitized (i.e., $\sigma_0 \cap \sigma_1$ if the untaint set is $(\mathsf{false} \Rightarrow \sigma_0, \mathsf{true} \Rightarrow \sigma_1)$).

4. **Return value:** If the function returns a Boolean value and it conditionally sanitizes a set of input parameters and global variables, we use the untaint representation to model that correlation:

   $$\frac{\begin{array}{l}\Gamma \vdash lv \stackrel{\mathtt{Lv}}{\Rightarrow} l \quad \Gamma \vdash e_1 \stackrel{\mathtt{E}}{\Rightarrow} v_1 \ \ldots \ \Gamma \vdash e_n \stackrel{\mathtt{E}}{\Rightarrow} v_n \\ \mathtt{Summary}(f) = \langle \mathcal{E}, \mathcal{R}, \mathcal{S}, \mathcal{X} \rangle \\ \mathcal{S} = (\mathsf{false} \Rightarrow \sigma_0, \mathsf{true} \Rightarrow \sigma_1) \quad \sigma_* = \sigma_0 \cap \sigma_1 \\ \sigma_0' = \mathsf{subst}_{\bar{v}}(\sigma_0 - \sigma_*) \quad \sigma_1' = \mathsf{subst}_{\bar{v}}(\sigma_1 - \sigma_*)\end{array}}{\Gamma \vdash lv \leftarrow f(e_1, \ldots, e_n) \stackrel{\mathtt{S}}{\Rightarrow} \Gamma[l \mapsto \mathsf{untaint}(\sigma_0', \sigma_1')]} \ \mathsf{fun\text{-}bool}$$

   In the rule above, $\mathsf{subst}_{\bar{v}}(\sigma)$ substitutes actual parameters (v_i) for formal parameters in σ.

   If the callee returns a string value, we use the return set component of the function summary ($\mathcal{R}$) to determine the set of input parameters and global variables that might become a substring of the return value:

   $$\frac{\begin{array}{l}\Gamma \vdash lv \stackrel{\mathtt{Lv}}{\Rightarrow} l \quad \Gamma \vdash e_1 \stackrel{\mathtt{E}}{\Rightarrow} v_1 \ \ldots \ \Gamma \vdash e_n \stackrel{\mathtt{E}}{\Rightarrow} v_n \\ \mathtt{Summary}(f) = \langle \mathcal{E}, \mathcal{R}, \mathcal{S}, \mathcal{X} \rangle \quad \sigma' = \mathsf{subst}_{\bar{v}}(\mathcal{R})\end{array}}{\Gamma \vdash lv \leftarrow f(e_1, \ldots, e_n) \stackrel{\mathtt{S}}{\Rightarrow} \Gamma[l \mapsto \mathsf{contains}(\sigma')]} \ \mathsf{fun\text{-}str}$$

Since we require the summary information of a function before we can analyze its callers, the order in which functions are analyzed is important. Due to the dynamic nature of PHP (e.g., include statements), we analyze functions on demand—a function f is analyzed and summarized when we first encounter a call to f. The summary is then memoized to avoid redundant analysis. Recursive function calls are rare in PHP programs. If we encounter a cycle during the analysis, our current implementation uses a dummy "no-op" summary as a model for the second invocation.

4. EXPERIMENTAL RESULTS

The analysis described in Section 3 has been implemented as two separate parts: a frontend based on the open source PHP 5.0.5 distribution that parses the source files into abstract syntax trees and a backend written in O'Caml that reads the ASTs into memory and carries out the analysis. This separation ensures maximum compatibility while minimizing dependence on the PHP implementation.

The decision to use different levels of abstraction in the intrablock, intraprocedural, and interprocedural levels enabled us to fine tune the amount of information we retain at one level independent of the algorithm used in another and allowed us to quickly build a usable tool. The checker is largely automatic and requires little human intervention for use. We seed the checker with a small set of query functions (e.g. mysql_query) and sanitization operations (e.g. is_numeric). The checker infers the rest automatically.

Regular expression matching presents a challenge to automation. Regular expressions are used for a variety of purposes including, but not limited to, input validation. Some regular expressions match well-formed input while others detect malformed input; assuming one way or the other results in either false positives or false negatives. Our solution is to maintain a database of previously seen regular expressions and their effects, if any. Previously unseen regular expressions are assumed by default to have no sanitization effects, so as not to miss any errors due to incorrect judgment. To make it easy for the user to specify the sanitization effects of regular expressions, the checker has an interactive mode where the user is prompted when the analysis encounters a previously unseen regular expression and the user's answers are recorded for future reference. Practically, we found this approach to be very effective and it helped us find at least two vulnerabilities caused by overly lenient regular expressions being used for sanitization.[4]

?What is prompted?

The checker detects errors by using information from the summary of the main function—the checker marks all variables that are required to be sanitized on entry as potential security vulnerabilities. From the checker's perspective, these variables are defined in the environment and used to construct SQL queries without being sanitized. In reality, however, these variables are either defined by the runtime environment or by some language constructs that the checker does not fully understand (e.g., the extract operation in PHP which we describe in the case study below). The tool emits an *error* message if the variable is known to be easily controlled by the user (e.g. $_GET['...'], $_POST['...'], $_COOKIE['...'], etc). For others, the checker emits a *warning*.

Weird Ones

We conducted our experiments on the latest versions of six open source PHP code bases: e107 0.7, Utopia News Pro 1.1.4, mybloggie 2.1.3beta, DCP Portal v6.1.1, PHP Webthings 1.4patched, and PHP fusion 6.00.204. Table 1 summarizes our findings for the first five. Our checker emitted a total of 99 error messages for the first five applications, where unsanitized user input (from $_GET, $_POST, etc) may flow into SQL queries. We manually inspected the error reports and believe all 99 represent real vulnerabilities. We have notified the developers about these errors and will publish security advisories once the errors have been fixed. We have not inspected warning messages—unsanitized variables of unresolved origin (e.g. from database queries, configuration files, etc) that are subsequently used in SQL queries due to the high likelihood of false positives.

	Err Msgs	Bugs	(FP)	Warn
e107	16	16	(0)	23
News Pro	8	8	(0)	8
myBloggie	16	16	(0)	23
DCP Portal	39	39	(0)	55
PHP Webthings	20	20	(0)	6
Total	99	99	(0)	115

Table 1: Summary of experiments. Err Msgs: number of reported errors. Bugs: number of confirmed bugs from error reports. FP: number of false positives. Warn: number of unique warning messages for variables of unresolved origin (uninspected).

PHP-fusion is different from the other five code bases because it does not directly access HTTP form data from input hash tables such as $_GET and $_POST. Instead it uses the extract operation to automatically import such information into the current variable scope. We describe our findings for PHP-fusion in the following subsection.

4.1 Case Study: Two Exploitable SQL Injection Attacks in PHP-fusion

In this section, we show two case studies of exploitable SQL injection vulnerabilities in PHP-fusion detected by our tool. PHP-fusion is an open-source content management system (CMS) built on PHP and MySQL. Excluding locale specific customization modules, it consists of over 16,000 lines of PHP code and has a wide user-base because of its speed, customizability and rich features. Browsing through the code, it is obvious that the author programmed with security in mind and has taken extra care in sanitizing input before use in query strings.

Our experiments were conducted on the then latest 6.00.204 version of the software. Unlike other code bases we have examined, PHP-fusion uses the extract operation to import user input into the current scope. As an example, extract($_POST, EXTR_O has the effect of introducing one variable for each key in the $_POST hash table to the current scope, and assigning the value of $_POST[key] to that variable. This feature reduces typing, but introduces confusion to the checker and security vulnerabilities to the software—both of the exploits we constructed involve use of uninitialized variables whose values can be manipulated by the user because of the extract operation.

Since PHP-fusion does not directly read user input from input hashes such as $_GETor $_POST, there are no direct error messages generated by our tool. Instead we inspect warnings (recall the discussion about errors and warnings above), which correspond to security sensitive variables whose definition is unresolved by the checker (e.g., introduced via the extract operation, or read from configuration files).

We ran our checker on all top level scripts in PHP-fusion. The tool generated 22 unique warnings, a majority of which relate to configuration variables that are used in the construction of a large number of queries[5]. After filtering those out, we arrive at 7 warnings in 4 different files.

[4] For example, Utopia News Pro misused "[0-9]+" to validate some user input. This regular expression only checks the *existence* of a number, instead of ensuring that the input *is* actually a number. The correct regular expression in this case is "^[0-9]+$".

[5] Data base configuration variables such as $db_prefix accounted for 3 false positives, and information derived from the database queries and configuration settings (e.g. locale settings) caused the remaining 12.

misses alarm when fire

~~the right~~

wait what is the underlying problem

Error could pre set won't get reset

all confused the 2 alarm when no fire

We believe all but one of the 7 warnings may result in exploitable security vulnerabilities. The lone false positive arises from an unanticipated sanitization:

```
/* php-files/lostpassword.php */
if (!preg_match("/^[0-9a-z]{32}$/", $account))
        $error = 1;
if (!$error) { /* database access using $account */ }
if ($error) redirect("index.php");
```

Instead of terminating the program immediately based on the result from preg_match, the program sets the $error flag to true and delays error handling, which is in general not a good practice. This idiom can be handled by adding slightly more information in the block summary.

We investigated the first two of the remaining warnings for potential exploits, and confirmed that both are indeed exploitable on a test installation. Unsurprisingly both errors are made possible because of the extract operation. We explain these two errors in detail below.

1) Vulnerability in script for recovering lost password. This is a remotely exploitable vulnerability that allows any registered user to elevate his privileges via a carefully constructed URL. We show the relevant code below:

```
1  /* php-files/lostpassword.php */
2  for ($i=0;$i<=7;$i++)
3     $new_pass .= chr(rand(97, 122));
4  ...
5  $result = dbquery("UPDATE ".$db_prefix."users
6     SET user_password=md5('$new_pass')
7     WHERE user_id='".$data['user_id']."'");
```

Our tool issued a warning for $new_pass, which is uninitialized on entry and thus defaults to the empty string during normal execution. The script proceeds to add seven randomly generated letters to $new_pass (lines 2-3), and uses that as the new password for the user (lines 5-7). The SQL request under normal execution takes the following form:

```
UPDATE users SET user_password=md5('???????')
  WHERE user_id='userid'
```

However, a malicious user can simply add a new_pass field to his HTTP request by appending, for example, the following string to the URL for the password reminder site:

&new_pass=abc%27%29%2cuser_level=%27103%27%2cuser_aim=%28%27

The extract operation described above will magically introduce $new_pass in the current variable scope with the following initial value:

abc'), user_level =' 103', user_aim = ('

The SQL request is now constructed as:

```
UPDATE users SET user_password=md5('abc'),
     user_level='103', user_aim=('???????')
  WHERE user_id='userid'
```

Here the password is set to "abc", and the user privilege is elevated to 103, which means "Super Administrator." The newly promoted user is now free to manipulate any content on the website.

2) Vulnerability in the messaging sub-system. This vulnerability exploits another use of potentially uninitialized variable $result_where_message_id in the messaging sub system. We show the relevant code in Figure 5.

Our tool warns about unsanitized use of $result_where_message_id. On normal input, the program initializes $result_where_message_id using a cascading if statement. As shown in the code, the author is very careful about sanitizing values that are used to construct $result_where_message_id. However, the cascading sequence of if statements does not have a fall back branch. And therefore, $result_where_message_id might be uninitialized on malformed input. We exploit this fact, and append

&request_where_message_id=1=1/*

The query string submitted on line 11-13 thus becomes:

```
DELETE FROM messages WHERE 1=1 /* AND ...
```

Whatever follows "/*" is treated as comments in MySQL and thus ignored. The result is loss of all private messages in the system. Due to the complex control and data flow, this error is unlikely to be discovered via code review or testing.

We reported both exploits to the author of PHP-fusion, who immediately fixed these vulnerabilities and released a new version of the software.

```
1  if (isset($msg_view)) {
2    if (!isNum($msg_view)) fallback("messages.php");
3    $result_where_message_id="message_id=".$msg_view;
4  } elseif (isset($msg_reply)) {
5    if (!isNum($msg_reply)) fallback("messages.php");
6    $result_where_message_id="message_id=".$msg_reply;
7  }
8  ... /* ~100 lines later */ ...
9  } elseif (isset($_POST['btn_delete']) ||
10   isset($msg_delete)) { // delete message
11   $result = dbquery("DELETE FROM ".$db_prefix.
12    "messages WHERE ".$result_where_message_id. // BUG
13    " AND ".$result_where_message_to);
```

Figure 5: An exploitable vulnerability in PHP-fusion 6.00.204.

5. RELATED WORK

5.1 Static techniques

WebSSARI is a type qualifier based analyzer for PHP [7]. It uses a standard intraprocedural tainting analysis to find cases where user controlled values flow into functions that require trusted input (*sensitive functions*). The analysis relies on three user written "prelude" files to provide information regarding: 1) the set of *all* sensitive functions–those require sanitized input; 2) the set of *all* untainting operations; and 3) the set of untrusted input variables. Incomplete specification will result in both false positives and false negatives.

The key limitation of WebSSARI is its analysis power: 1) the analysis is intraprocedural and does not infer function pre- and post-conditions, thus requiring extensive annotations to use; 2) it does not model predicates and conditional branches, which is a key mechanism for testing and sanitizing input variables in PHP; and 3) it uses a generic type based algorithm which does not model dynamic features in scripting languages like PHP. For example, dynamic typing may introduce subtle errors that WebSSARI misses. The include statement dynamically inserts code to the program which may contain, induce, or prevent errors.

Livshits and Lam [8] developed a static detector for security vulnerabilities (e.g. SQL injection, cross site scripting, etc) in Java applications. The algorithm uses a BDD-based context-sensitive pointer analysis [18] to find potential flow from untrusted sources (e.g. user input) to trusting sinks (e.g. SQL queries). One limitation of this analysis is that it does not model control flow in the program and therefore may misflag sanitized input that subsequently flows into SQL queries. Sanitization with conditional branching is common in PHP programs, so techniques that ignore control flow are likely to cause large numbers of false positives on such code bases.

Other tainting analysis that are proven effective on C code include CQual [4], MECA [20], and MC [6, 2]. Collectively they have found hundreds of previously unknown security errors in the Linux Kernel.

Christensen *et. al.* [3] developed a string analysis that approximates string values in a Java program using a context free grammar. The result is then widened into a regular language and can be checked against a specification of expected output to determine syntactic correctness. However, syntactic correctness does not entail safety, and therefore it is unclear how one can adapt this work to the detection of SQL injection vulnerabilities. Minamide [9] extended the approach and constructed a string analyzer for PHP. It cited SQL injection detection as a possible application. However, the analyzer models a small set of string operations in PHP (e.g. concatenation, string matching and replacement), and ignores more complex features such as dynamic typing, casting, and predicates. Furthermore, the framework only seems to model sanitization with string replacement, which represents a small subset of all sanitization in real code. Therefore, accurately pinpointing injection attacks remains challenging.

Gould *et. al.* [5] combines string analysis with type checking to ensure not only syntactic correctness but also type correctness for SQL queries constructed by Java programs. However, type correctness does not guarantee safety, which is the focus of our analysis.

5.2 Dynamic Techniques

Scott and Sharp [14] proposed an application-level firewall to centralize sanitization of client input. Firewall products are also commercially available from companies such as NetContinuum, Imperva, Watchfire, etc. Some of these firewalls detect and guard against previously known attack patterns, while others maintain a white list of valid inputs. The main limitation here is that the former is susceptible to both false positives and false negatives, and the latter is reliant on correct specifications, which are difficult to come by.

The Perl taint mode [11] enables a set of special security checks during execution in an unsafe environment. It prevents the use of untrusted data (e.g. all command line arguments, environment variables, data read from files, etc) in operations that require trusted input (e.g. any command that invokes a sub-shell). Nguyen-Tuong [10] proposed a taint mode for PHP. It employs a set of heuristics to determine whether a query is safe when it contains fragments of user input. The limitation of a heuristics based approach is that it is susceptible to both false positives and false negatives, which presents an obstacle for deployment in a production environment.

In general, the advantage of a static analysis is that it finds the root cause of a security vulnerability and prevents the attack before the application is deployed.

6. CONCLUSION

We have presented a static analysis algorithm for detecting security vulnerabilities in PHP. Our analysis employs a novel three-tier architecture that enables us to handle dynamic features unique to scripting languages such as dynamic typing and code inclusion. We demonstrated the effectiveness of our approach by running our tool on six popular open source PHP code bases and finding 105 previously unknown security vulnerabilities, most of which we believe are remotely exploitable.

7. REFERENCES

[1] A. Aiken, E. Wimmers, and T. Lakshman. Soft typing with conditional types. In *Proceedings of the 21st Annual Symposium on Principles of Programming Languages*, 1994.

[2] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *2002 IEEE Symposium on Security and Privacy*, 2002.

[3] A. Christensen, A. Moller, and M. Schwartzbach. Precise analysis of string expressions. In *Proceedings of the 10th Static Analysis Symposium*, 2003.

[4] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, June 2002.

[5] C. Gould, Z. Su, and P. Devanbu. Static checking of dynamically generated queries in database applications. In *Proceedings of the 26th International Conference on Software Engineering*, 2004.

[6] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.

[7] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th International World Wide Web Conference*, 2004.

[8] V. Livshits and M. Lam. Finding security vulnerabilities in Java applications with static analysis. In *Proceedings of the 14th Usenix Security Symposium*, 2005.

[9] Y. Minamide. Approximation of dynamically generated web pages. In *Proceedings of the 14th International World Wide Web Conference*, 2005.

[10] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In *Proceedings of the 20th International Information Security Conference*, 2005.

[11] Perl documentation: Perlsec.
`http://search.cpan.org/dist/perl/pod/perlsec.pod`.

[12] PHP: Hypertext Preprocessor. `http://www.php.net/`.

[13] PHP usage statistics. `http://www.php.net/usage.php`.

[14] D. Scott and R. Sharp. Abstracting application-level web security. In *Proceedings of the 11th International World Wide Web Conference*, 2002.

[15] Security space apache module survey (Oct 2005).
`http://www.securityspace.com/s_survey/data/man.200510/apachemods.html`.

[16] Symantec Internet security threat report: Vol. VII. Technical report, Symantec Inc., Mar. 2005.

[17] TIOBE programming community index for November 2005.
`http://www.tiobe.com/tpci.htm`.

[18] J. Whaley and M. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, 2004.

[19] A. Wright and R. Cartwright. A practical soft type system for Scheme. *ACM Trans. Prog. Lang. Syst.*, 19(1):87–152, Jan. 1997.

[20] J. Yang, T. Kremenek, Y. Xie, and D. Engler. MECA: an extensible, expressive system and language for statically checking security properties. In *Proceedings of the 10th Conference on Computer and Communications Security*, 2003.

Notation for Yichen's paper

Figure 4 in Yichen's paper uses some notation that might not be familiar. Here are some notes to help you along:

- The turnstile symbol ⊢, in "G ⊢ x", basically means "x is true in the environment G".
- The horizontal line is called a "rule": if all the things above the line are true, then the thing below the line must be true.

The Wikipedia page on sequents has more details.

Sequent

From Wikipedia, the free encyclopedia

In proof theory, a **sequent** is a formalized statement of provability that is frequently used when specifying calculi for deduction. In the sequent calculus, the name *sequent* is used for the construct which can be regarded as a specific kind of judgment, characteristic to this deduction system.

Contents

Explanation

A sequent has the form

$$\Gamma \vdash \Sigma$$

where both Γ and Σ are sequences of logical formulae (i.e., both the number and the order of the occurring formulae matter). The symbol $\vdash$ is usually referred to as *turnstile* or *tee* and is often read, suggestively, as "yields" or "proves". It is not a symbol in the language, rather it is a symbol in the metalanguage used to discuss proofs. In a sequent, Γ is called the antecedent and Σ is said to be the succedent of the sequent.

Intuitive meaning

The intuitive meaning of the sequent $\Gamma \vdash \Sigma$ is that under the assumption of Γ the conclusion of Σ is provable. Classically, the formulae on the left of the turnstile can be interpreted conjunctively while the formulae on the right can be considered as a disjunction. This means that, when all formulae in Γ hold, then at least one formula in Σ also has to be true. If the succedent is empty, this is interpreted as falsity, i.e. $\Gamma \vdash$ means that Γ proves falsity and is thus inconsistent. On the other hand an empty antecedent is assumed to be true, i.e., $\vdash \Sigma$ means that Σ follows without any assumptions, i.e., it is always true (as a disjunction). A sequent of this form, with Γ empty, is known as a logical assertion.

Of course, other intuitive explanations are possible, which are classically equivalent. For example, $\Gamma \vdash \Sigma$ can be read as asserting that it cannot be the case that every formula in Γ is true and every formula in Σ is false (this is related to the double-negation interpretations of classical into intuitionistic logic, such as Glivenko's theorem).

In any case, these intuitive readings are only pedagogical. Since formal proofs in proof theory are purely syntactic, the meaning of (the derivation of) a sequent is only given by the properties of the calculus that provides the actual rules of inference.

Barring any contradictions in the technically precise definition above we can describe sequents in their introductory logical form. Γ represents a set of assumptions that we begin our logical process with, for example "Socrates is a man" and "All men are mortal". The $\sum$ represents a logical conclusion that follows under these premises. For example "Socrates is mortal" follows from a reasonable formalization of the above points and we could expect to see it on the $\sum$ side of the *turnstile*. In this sense, $\vdash$ means the process of reasoning, or "therefore" in English.

Example

A typical sequent might be:

$$\phi, \psi \vdash \alpha, \beta$$

This claims that either α or β can be derived from ϕ and ψ.

Property

Since every formula in the antecedent (the left side) must be true to conclude the truth of at least one formula in the succedent (the right side), adding formulas to either side results in a weaker sequent, while removing them from either side gives a stronger one.

Rules

Most proof systems provide ways to deduce one sequent from another. These inference rules are written with a list of sequents above and below a line. This rule indicates that if everything above the line is true, so is everything under the line.

A typical rule is:

$$\frac{\Gamma, \alpha \vdash \Sigma \qquad \Gamma \vdash \Sigma, \alpha}{\Gamma \vdash \Sigma}$$

This indicates that if we can deduce that Γ, α yields $\sum$, and that Γ yields Σ, α, then we can also deduce that Γ yields $\sum$.

Variations

The general notion of sequent introduced here can be specialized in various ways. A sequent is said to be an **intuitionistic sequent** if there is at most one formula in the succedent. This form is needed to obtain calculi for intuitionistic logic. Similarly, one can obtain calculi for dual-intuitionistic logic (a type of paraconsistent logic) by requiring that sequents be singular in the antecedent.

In many cases, sequents are also assumed to consist of multisets or sets instead of sequences. Thus one disregards the order or even the number of occurrences of the formulae. For classical propositional logic this does not yield a problem, since the conclusions that one can draw from a collection of premises does not depend on these data. In substructural logic, however, this may become quite important.

History

Historically, sequents have been introduced by Gerhard Gentzen in order to specify his famous sequent calculus. In his German publication he used the word "Sequenz". However, in English, the word "sequence" is already used as a translation to the German "Folge" and appears quite frequently in mathematics. The term "sequent" then has been created in search for an alternative translation of the German expression.

This article incorporates material from Sequent on PlanetMath, which is licensed under the Creative Commons Attribution/Share-Alike License.

Retrieved from "http://en.wikipedia.org/w/index.php?title=Sequent&oldid=487274092"
Categories: Proof theory | Logical expressions

- This page was last modified on 14 April 2012 at 02:26.

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 lec*n*.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 8

alarm when no error

The paper only mentions one potential false positives arising because of the use of regular expression. Explain why it is indeed a false positive.

(2nd word is if there is a problem)

~~A well formed~~

2 ways to detect

- look for only correct strings
  - Some string may be missed
- look for attacks

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

Top // **6.858 home** // *Last updated Saturday, 29-Sep-2012 10:55:47 EDT*

9/30

Paper Question 8

Michael Plasmeier

You can write regular expressions in one of two ways. You can either make sure that input matches the structure of good input, or that there is no known attack strings in the input. Both are subject to false positives and false negatives. A false positive is when there is an alarm, but no fire. A false negative is a fire, but no alarm. Either structure could label good input as suspect, or miss some crucial attack string. The matching of well-formed inputs is more susceptible to a false positive, while looking for attack strings is more susceptible to a false negative.

The false positive in PHP-fusion on page 9 arises from the fact that the order of the code shows the SQL statement is executed first, even though it actually would not execute when run, since `$error` would be set `true`. This caused the first version of the static analysis to raise an alarm when the code was fine. As the authors said, this was easy to fix.

6.858
L8 Static Analysis

10/1

Fix bugs in applications

Give reports to developers

PHP

- like python
- very dynamic

harder to build a tool

What type of bugs?

SQL injection

```
$id = $_GET['id'];
mysql_query("UPDATE ___ WHERE id=
                $id");
```

They are only looking to make sure sanatized

└ Not that you could pass in an arbitary id

②

```
echo "Hi $id";
```

↳ it better be sanatized by now

also directory traversal

```
open("../../../foo/$id");
```

eval
- common in JS

```
eval("... $id")
```

↑ arbitrary string

These are all ~~unched~~ unchecked input bugs

3

Prof: Sorry for the notation in the paper

```
function check($a) {
        $ok = is_numeric($a);
        return $ok
}
function check2($b) {
        $v = check($b);
        if (!$v) exit
        else return;
}
$c = $_GET['x'];
check2($c);
$q = "xx $c yy";
mysql_query($q);
```

9

Goal: How can it possibly behave under all possible inputs

fuzzing → only the inputs you can come up w/

but it had   if (sha1(arg) != "....") exit

we don't know when true

Though neither fuzzing or static work

Must give up on precision - but hope close enough

3 level of analysis

basic block

no if, return, branches, etc

Very detailed analysis

Come up w/ summary

function combine summary for each basic block

→ look at summary for main function

Check() | basic block

don't know ~~$~~a will be

but symbolic execution

run in simulation env

variables don't have concrete value

~~could be~~ but symbols instead

Γ = Gamma = map memory location → symbolic value

like variable names → values might be

→ how relates to either

$$\Gamma = \left\{ \begin{array}{l} a \to a_0 \\ ok \to ok_0 \end{array} \right\}$$

~~$~~ok = is_numeric(~~$~~a);

$$\Gamma = \left\{ \begin{array}{l} a \to a_0 \\ ok \to \text{boolean, but don't know value} \end{array} \right.$$

if true, some variables (a) are sanitized

if false, nothing sanitized

Ok $\exists$ $\text{untaint}(\underset{\substack{?\\ \text{false}}}{\{\}}, \{a_0\})$

All they care about is SQL injection
Not logic correctness, etc

$\boxed{\perp = \text{unknown}}$

How do they know is_numeric is doing right thing
└ well its built in, primitive

So the pre-annotate functions for sanitizing

This would be a lot to store
↳ so want to summarize

4 magic variables

E = error set
Variables that might flow into SQL query
R = return set
Variables that might end up here
S = Which values sanatized if return is boolean
Only strings (missed) here
X = (missed)

7

Here

$E = \{\}$

$R = \{\}$ not a string

$S = \{ T \to \{Arg1\} , F \to \{ \} \}$ ← do renamed

? look at untaint

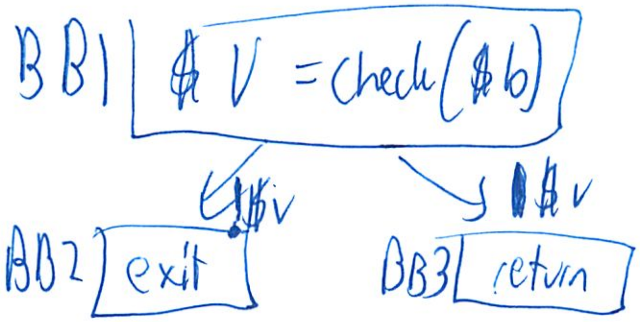
$X = false$

(missed something about global variables)

So we just have this description

Check 2

a few more blocks here



(8)

BB1 $G = \{b \to b_0, v \to v_0\}$

$Bv = \text{Check}(b)$

$G = \{b \to b_0, v \to \text{untaint}(\overset{F}{\{\}}, \overset{T}{\{b_0\}})$

↑just look at summary
no string
but S, so untaint

Summary $E = \{\}$, $D = \{v\}$, $F = \{\}$, $T = \text{false}$

↑variables that were defined in this block

↑flow map which variables in input move to output to track where strings end up

↑exit statement

~~BB2~~

$R = \text{undef}$, $V = \{V_{BB2}, V_{BB3})$

↑specify untained values for each control flow edge of basic block

$V_{BD2} = \{\}$ ← left

$V_{BD3} = \{b\}$ ← right

BB2

E = {}, D = {}, F = {}

T = true, R = undef, V = ~~nothing~~ none

↑ terminates

BB3

E = {}, D = {}, F = {}

T = false, R = {}, V = none

Now put togethe

Check 2

E = {}, R = {}, S = {x → {Arg#1}}

↑ nothing passes anything to SQL query

↑ backtrack the return values turns out we are ~~not sanitizing anything~~ is sanitizing ✓

X = false

↑ before we fixed if(!$v)
↑
then it would have been
S = {}

So

$G = \{ c \to c_0, q \to q_0, \_GET \to \_GET_0,$

$\_GET[x] \to \_GET[x]_0 \}$

$\$c = \ldots$

$G = \{ c \to \_GET[x]_0, \ldots \}$, sanitized $= \{\}$

Check2($\$c$)

$G = \{ \ldots \}$, sanitized $= \{\}$ or $\{\_GET[x]_0\}$

Some extra piece of state

└ set of things that are sanitized

Sanit'

$\$q = \ldots$

all the care is which strings appear in mix

$G = \{ q \to [\text{'xx'}, \_GET[x]_0, \text{'yy'}], \ldots \}$

(11)

mysql query()

↳ $G = \{ \ldots \}$ but everything better be sanitized! for q

↑unchanged

Summary main

$E = \{ \_GET[x] \}$ { things that end up in sql_query just care about Global variables not "xx" or "yy"

$R = \{ \}$

$S = \{ \ldots \}$

$X = \text{true}$

So look at it + see if bug

↳ is main's E empty?

No! So might be bug (this is w/o if(!$v))
↑

Might let subfunction have non empty E

But main must be right

w/ if(!$v) $E = \{ \}$

Other variations

└ does it still work

1) Call Check2($_GET["x"])

will get into sanitized set

it notices its the same thing

2.)
```
function Check3($f) {
    $b = $_GET[$f];
    Check2($b);
}
```

Will this prevent SQL injection?

It should

Static analysis → will treat x as a string

Need a summery

E={} R={} S=?

X = false

$$G = \left\{ \begin{array}{l} f \to f_0 \\ b \to b_0 \end{array} \right\}$$

$$\$b \cancel{\#}= \$\_GET[\underbrace{\cancel{\$}f}]$$

if can figure out, use it
otherwise _GET[⊥]

$$G = \{ b \to \_GET[\bot], \}$$

? Something, but I don't know what

So then when we call check3('x')

$Sanitized = \{\_GET[\bot]_0\}$

So E will not be empty

And it will be flagged

So their summary language is not detailed enough to handle this

if summary was too detailed → ~~[illegible]~~ would be whole program

tradeoff of false pos + false~~o~~ neg (missed details)

Built in functions

have summary for each built in fn

ie: substr($s, ...)

E={}
R={Arg#1}
S={}
X =false

Mysqlquery($q)

E = {Arg#1} ← will put in SQL query
R={} Caller better be sanatizing
S={}
x = false

is_numeric($v)

E={} S={T→{Arg#1}, F→{}}
R={} X=false

(15)

Regular Expressions

they don't know what they do

```
$ok = preg_match("[0-9]+", $s)
```

Insted: Can prompt user "Does it sanitize?"

No → must be

```
"^[0-9]+$"
```

~~Or else~~

Otherwise would have matched "aa 0 bb c"

So is it good

- ~~The~~ No ~~real~~ harm in running

They distinguish b/w error + warning

16

(missed)

Looking at DELETE WHERE id = "bob"
OR 1=1

Programmer might actually want to do
So must look at history

Less SQL injection now
- prepared statement
- Or magic-object model

Also runtime versions
attach extra bit
but also false positive
but those run on users site

10/1

Static analysis

```
What's the goal of this paper?
  Help developers fix vulnerabilities, by finding security bugs in PHP code.
  Show that static analysis is feasible for PHP.

What kinds of vulnerabilities are they looking for?
  Unchecked input vulnerabilities: missing sanitization checks.
  Most specifically, tool targeted at SQL injection.
    $rows = mysql_query("UPDATE users SET pass='$pass' WHERE userid='$userid'")
  Authors also claim their approach might work for cross-site scripting.
    echo "Hello $userid\n";
  Might work for other cases of unsanitized input.
    open("/foo/bar/$filename") may be vulnerable to filenames with ".." or "/".
    eval($_GET['x']).

Example program [bug: missing "not" in check2's if statement]:

  function check($a) {
    $ok = is_numeric($a);
    return $ok;
  }

  function check2($b) {
    $v = check($b);
    if ($v) exit;
    else return;
  }

  $c = $_GET['x'];
  check2($c);
  $q = "xx $c yy";
  mysql_query($q);

Static analysis goal: understand how some code behaves, given any inputs.
  In our simple example, can cover all behavior with a few inputs.
  For large applications, impractical to enumerate possible inputs.
  Control flow depends on inputs, need to consider all possible paths.
  Recursive functions, loops, etc make enumeration difficult.
  Hard to decide some questions statically.
  E.g., suppose we add this before mysql_query():

    if (sha1($c) != '...') exit();

  Is there an input that hashes to this value and triggers SQL injection?
  .. hopeless ..

Approach: lose precision but gain scalability.
  Analyze smaller pieces in detail: typically basic blocks or functions.
  Summarize interesting aspects of that piece in a concise way.
  Use the summary when other pieces call this piece.
  This tool actually applies this summary-based idea at two levels.
    First analyze basic block, generate BB summary.
    Then combine BBs in a function, generate function summary.
    Function summary for the main section/function used to flag errors.

Analyzing check(): only one basic block.
  How do we figure out what's going on in this function without running it?
  This tool: simulate execution on "symbolic" values.
    The current set of symbolic values is denoted with the symbol G (UTF-8: Γ).
    Maps memory locations (variables, hash table entries) to symbolic values.
  Initially, each memory location is assigned to initial symbolic value.
    G = { a -> a0, ok -> ok0 }
  What happens when $ok=is_numeric($a) "runs" in our simulation?
    Need to assign some value to the "ok" memory location.
    No idea what value will be returned.
    But what we care about is how it might relate to validating input.
    This tool's model has a special kind of value in simulation: untaint.
    Symbolic value untaint({x, ..}, {y, ..}) means:
      If actual value is false, then {x, ..} are all validated.
      If actual value is true, then {y, ..} are all validated.
      Still don't know if it's true or false, but will help analysis shortly.
    "Tainted" means could be a "malicious" value.
    Simulation also supports "true", "false", and "unknown" (UTF-8: ⊥) bools.
  Thus, after $ok=is_numeric($a), new state is:
    G = { a -> a0, ok -> untaint({}, {a0}) }
    [ Section 3.1.3 suggests it should be location {a} instead of value {a0},
      but that doesn't make much sense, and step 4 in section 3.3 seems to
      suggest it should be a value instead of a memory location. ]
  How does the tool know what is_numeric does?  What is "malicious"?
    Hard-coded by tool maker / developer for SQL injection.
    No underlying proof that is_numeric does the right thing.
    Would similarly hard-code functions that sanitize SQL strings.
  Return statement finishes basic block, done with BB analysis.

How to summarize check()'s behavior?  [Section 3.2]
  This tool's function-level summary boils down to <E, R, S, X>:
  E (error set): memory locations that might flow into an SQL query,
                 but aren't sanitized in the function itself.
    For check(), E={}.
  R (return set): what parameters or global variables might be in retval?
    For check(), R={}.
  S (sanitized values): what parameters or global variables are sanitized?
    Can be conditional on boolean return value, or unconditional.
    For check(), S={ T => {Arg#1}, F => {} }
  X (exit): does this function always exit?
    For check(), X=false.

Analyzing check2(): three basic blocks.
                               /-> [ exit ]
  entry -> [ call to check ]
                               \-> [ return ]

First basic block of check2().
  Initial: G = { b -> b0, v -> v0 }.
  Call to check() -- what symbolic value does v get?  [Section 3.3, step 4]
    Figure out what values are passed as arguments to check(): Arg#1 ==E==> b0
    Figure out what the sanitized value summary is: S={T=>{Arg#1}, F={}}
    Plug in arguments into summary, convert to an untaint value:
      v -> untaint({}, {b0})
  Two control flow transitions from BB: one to exit, another to return.

How to summarize basic block?
  Slightly more complex than a function:  [Section 3.1.7]
  E (error set): memory locations that might flow to SQL query.
    For check2's first BB, E={}.
  D (definitions): which memory locations were defined in this BB?
    For check2's first BB, D={v}.
  F (value flow): how did this BB move strings between memory locations?
    For check2's first BB, F={}.
  T (termination): is there an exit statement?
    For check2's first BB, T=false.
  R (return value): what's being returned (if anything)?
    For check2's first BB, R=undefined.
    U (untaint set): for each successor BB, what's sanitized?
      For the exit BB ($v==true): U={b}.
      For the return BB ($v==false): U={}.
      [ Or, if we fix bug, vice-versa. ]

  What happens in the exit BB?
    E={}, D={}, F={}, T=true, R=undef, No successors -> no U's.

  What happens in the return BB?
    E={}, D={}, F={}, T=false, R=undef, No successors -> no U's.

  How to summarize the check2() function?
    E={}, R={}, X=false.
    How to compute S?  [ Section 3.2 step 3 ]
      Find all return blocks: just the return BB.
      Find the set of sanitized inputs for that BB (chain of U's from entry).
      Take intersection.
    S={* => {}}  [ or, if we fix bug, S={* => {Arg#1}} ]

  Finally, what happens in the main function?  One basic block.
    Initial:     G = { c -> c0, q -> q0, _GET -> _GET0, _GET[x] -> _GET[x]0 }
    Assign $c:   G = { c -> _GET[x]0, .. }
    Call check2: use S to mark validated locations.  [ Section 3.3 step 3]
      If we fix bug, this marks location c's value (_GET[x]0) as validated.
    Compute $q:  G = { q -> [ 'xx', _GET[x]0, 'yy' ], .. }
    Call mysql_query: need to ensure that all parts of q's value are validated.

  More complicated examples:
    Suppose we replace check2($c) with check2($_GET['x'])?
      Works fine: summary for check2 says _GET[x]0 becomes validated.

    What if we replace $c = $_GET['x'] with:  [ Section 3.1.4 ]
      $d = $_GET;
      $e = 'x';
      $c = $d[$e];

      Init:      G = { c -> c0, d -> d0, e -> e0, .. }
      Assign $d: G = { d -> _GET0, .. }
      Assign $e: G = { e -> ['x'], .. }
      Assign $c: G = { c -> _GET[x]0, .. }

    What if we instead use check3('x'):
      function check3($f) {
        $b = $_GET[$f];
        $v = check($b);
        if (!$v) exit;  # bug fixed
        else return;
      }

      Init:      G = { b -> b0, f -> f0, .. }
      Assign $b: G = { b -> _GET[bottom]0, .. }
      Check:     G = { v -> untaint({}, {_GET[bottom]0}) }
      Summary:   S = { * => {_GET[bottom]0} }

      In main function, _GET[bottom]0 marked validated, not _GET[x]0.

    Built-in functions, like substr()?
      Presumably need to hard-code summaries for them.
      What would the summary for substr() look like?
        E={}, R={Arg#1}, S={}, X=false.

  What does the tool need to hard-code about SQL injection?
    Sink sites: mysql_query().
```

```
      What would the summary for mysql_query() look like?
      E={Arg#1}, R={}, S={}, X=false.

    Sanitization functions: is_numeric(), presumably others.
      What would the summary for is_numeric() look like?
        E={}, R={}, S={ T => {Arg#1}, F => {} }, X=false.
      What would the summary for mysql_escape_string() look like?
        E={}, R={}, S={}, X=false.

    Source sites: any "external" values that aren't assigned in program code.
      Flag external inputs from $_GET, $_POST, etc as errors.
      Others are just warnings -- false positives.
    Why do they run into external variables other than $_GET, $_POST, etc?
      Analysis doesn't understand / deal with extract().
    What about unvalidated values as return values from built-in functions?
      E.g., read from socket or return from mysql_query()?
      Paper doesn't say anything about this case.
      May want to represent return value as unvalidated.
      Could invent a fake global string variable for this.

  Why do they make such a big deal about regexps?
    Often used for pattern-matching to validate inputs.
    Their analysis doesn't know ahead of time which regexps validate correctly.
    For each new regexp, ask user if it validates input properly.
    Seems like an error-prone step: easy to declare incorrect regexp as correct.
    Nice example from Utopia News Pro: "[0-9]+" should be "^[0-9]+$".

  How well does this work for SQL injection?
    For 5 applications, 99 error reports (unsanitized $_GET params, etc).
    Manually checked all 99 error reports, decided they were real bugs.
    Didn't get acknowledgment from developers if bugs are real or not.
    Also got 115 warnings for those 5 apps, suggest mostly false positives.

    Other application: PHP-fusion, uses extract() -> no errors, 22 warnings.
      15 false positives: unvalidated config variables used to construct queries.
      7 remaining warnings, authors say 6 are real bugs and 1 FP (below).
      Developer acknowledged and fixed two bugs.  (Unclear what about other 4..)

  PHP-fusion false positive example:
    if (!preg_match("...", $account))
      $error = 1;
    if (!$error) { mysql_query("xx $account xx"); }

    Why does this result in a false positive?
    How could we augment BB summary to handle this correctly?

  How would you modify this tool to catch cross-site scripting?
    Sinks?
    Sources?
    Sanitization functions?
    Current design keeps track of only one kind of sanitization per analysis.

  What are the sources of false positives?
    Tool doesn't understand some sanitization function.
      (But it does ask when it sees a new regexp.)
    Tool doesn't understand subtle sanitization / checking plan.
      Doesn't conform to return value patterns used by this tool.
      Sources aren't actually malicious (e.g., config file inputs).

  What are the sources of false negatives?
    Code in eval or dynamically-generated PHP filenames.
    Can't determine array index (e.g., computed at runtime).
    Recursive functions.
```

```
Are these ideas applicable to static analysis for other languages?
  Python?  Seems reasonably close, fewer implicit conversions.
  C?  Less string-oriented, hard to distinguish types, aliasing.
  Java?  Well-defined strings, much stricter than Python.
    PQL is a similar tool for Java, looks for source-to-sink paths.

Advantages / disadvantages of different approaches to fixing vulnerabilities.
  Prevent bugs in the first place.
    Prepared SQL statements: Java example from the paper.
    Cross-site scripting: HTML templates?
    Avoid error-prone functions, such as PHP's extract().
      [ Similar bug affected github: Ruby-on-Rails mass assignment ]
    Privilege separation might work, but requires significant re-design (lab 2).
    +: good to prevent bugs when possible.
    -: programmers may still make mistakes even with less error-prone APIs.
    -: developers prefer simple APIs, can be hard to combine simple+secure.
      Lab 2's privilege separation, SQL prepared statements, etc.
    -: applications evolve, can't always predict developer's needs.
  Test cases / fuzzing.
    Good idea but tests cover only situations programmer already thought about.
    Fuzzing helps but might not trigger bugs that require complex inputs.
    Might require a lot of time to get coverage.
  Runtime checking: taint tracking.
    Potentially fewer false positives than static analysis.
    Adds runtime overhead.
    Bugs flagged at runtime may show up for the user, rather than developer.
    Taint propagation can be too conservative (false alarms = false positives).
    Taint propagation can be too lax (missed alarms = false negatives).
  Grep.
    What if we just "grep mysql_query *.php"?
  Static analysis.
    +: no need to run code, might get more coverage than testing / fuzzing.
    +: can run at development time, help developers fix bugs.
    +: no runtime overhead in deployment.
    -: potential for false positives.
    -: precise analysis impossible in all cases (decidability, halting prob).
    -: might miss bugs since analysis is not perfectly precise.
    -: might require some non-trivial time to analyze code.

Are static analysis tools actually used?
  Doesn't appear that this specific tool got much immediate use.
    However, the underlying ideas seem pretty good.
    Other tools use / build on similar techniques.
    Summary-based analysis is a common approach.
  Static analysis for C and Java programs pretty common.
    Linux kernel developers use some static analysis tools (sparse, smatch).
    Coverity provides commercial static analysis tools.
    Java static analysis tools are reasonably common.
      E.g., IBM helps customers find bugs in Java software.
  Static analyses look for all kinds of bugs.
    Memory leaks.
    Buffer overflows in C.
    Unchecked inputs in Java.
    Missing access control checks.
```

Read ~~at~~ 10/3

Run-Time Enforcement of Secure JavaScript Subsets

Sergio Maffeis
Imperial College London
maffeis@doc.ic.ac.uk

John C. Mitchell
Dep. of Computer Science
Stanford University
mitchell@cs.stanford.edu

Ankur Taly
Stanford University
ataly@stanford.edu

Abstract

Many Web-based applications such as advertisement, social networking and online shopping benefit from the interaction of trusted and unstrusted content within the same page. If the untrusted content includes JavaScript code, it must be prevented from maliciously altering pages, stealing sensitive information, or causing other harm. We study filtering and rewriting techniques to control untrusted JavaScript code, using Facebook FBJS as a motivating example. We explain the core problems, provide JavaScript code that enforces provable isolation properties at run-time, and compare our results with the techniques used in FBJS.

1 Introduction

Many contemporary web sites incorporate untrusted content. For example, many sites serve third-party advertisements, allow users to post comments that are then served to others, or allow users to add their own applications to the site. Although untrusted content can be placed in an isolating `iframe` [3], this is not always done because of limitations imposed on communication between trusted and untrusted code. Instead, Facebook [18], for example, preprocesses untrusted content, applying filters and source-to-source rewriting before the content is served. While some of these methods make intuitive sense, JavaScript [7, 9] provides many subtle ways for malicious code to subvert language-based isolation methods, as shown here and in our previous work [14].

In this paper, we review some previous filtering methods for managing untrusted JavaScript [14] and explore ways of replacing some aspects of these restrictive static code filters with more flexible run-time instrumentation that is implementable as source-to-source translation. Our previous efforts uncovered problems and vulnerabilities with the then-current versions of FBJS and ADsafe [5], Yahoo's safe advertising proposal. We then developed a formal foundation for proving isolation properties of JavaScript programs [14], based on our operational semantics of the full ECMA-262 Standard language (3rd Edition) [6], available on the web [12] and described previously in [13]. The language subsets defined in [14] provided a foundation for code filtering – any JavaScript filter that only allows programs in a meaningful sublanguage will guarantee any semantic properties associated with it. More specifically, we developed proofs that certain subsets of the ECMA-262 Standard language make it possible to syntactically identify the object properties that may be accessed, make it possible to safely rename variables used in the code, and/or make it possible to prevent access to scope objects (including the global object). However, these syntactic subsets are more restrictive than the solution currently employed by Facebook, which uses run-time instrumentation to restrict the semantic behavior of code that would not pass our filters. In this paper, we therefore focus on subsets of JavaScript and semantic restrictions that model the effect of rewriting JavaScript source code with "wrapper" functions. Our main contribution is the definition of JavaScript code that implements secure, semantic preserving run-time checks that enforce isolation of untrusted JavaScript code. We also compare our methods with the solutions employed by Facebook a the time of our submission. In particular, we describe a previously unknown Facebook vulnerability that we discovered thanks to our analysis, and the fix adopted in the current version of FBJS following our disclosure to them.

Related work on language-based methods for isolating the effects of potentially malicious web content include [16], which examines ways to inspect and cleanse dynamic HTML content, and [24], which modifies questionable JavaScript, for a more restricted fragment of JavaScript than we consider here. A short workshop paper [23] also gives an architecture for server-side code analysis and instrumentation, without exploring details or specific methods for constraining JavaScript. The Google Caja [4] project follows instead a different approach, based on transparent compilation of JavaScript code into a capability-based JavaScript subset, with libraries that emulate DOM objects.

I think I remember this ...

Additional related work on rewriting based methods for controlling the execution of JavaScript include [11]. Foundational studies of limited subsets of JavaScript and dynamic languages in general are reported in [2, 21, 24, 10, 17, 1, 22]; see [13].

2 JavaScript Isolation Problems

In this Section, we summarize the Facebook isolation mechanism. Over time, several teams of researchers have discovered flaws in the Facebook protection mechanisms that were promptly addressed by the Facebook team [8, 15, 14]. Specific handling of $FBJS.ref described below, for example, is the result of vulnerabilities reported to Facebook [14]. Based on past evidence, we believe it is important to develop a foundation for proving isolation properties. Without careful scrutiny and reliable semantic methods, it is simply not possible to reliably reason about a programming language as complex as JavaScript.

2.1 Facebook JavaScript

Facebook is a web-based social networking application. Registered and authenticated users store private and public information on the Facebook website in their Facebook profile, which may include personal data, list of friends (other Facebook users), photos, and other information. Users can share information by sending messages, directly writing on a public portion of a user profile (called the wall), or interacting with Facebook applications.

Facebook applications can be written by any user and can be deployed in various ways: as desktop applications, as external web pages displayed inside a frame within a Facebook page, or as integrated components of a user profile. Integrated applications are by far the most common, as they affect the way a user profile is displayed.

Facebook applications are written in FBML [20], a variant of HTML designed to make it easy to write applications and also to restrict their possible behavior. A Facebook application is retrieved from the application publisher's server and embedded as a subtree of the Facebook page document. Since Facebook applications are intended to interact with the rest of the user's profile, they are not isolated inside an `iframe`. However, the actions of a Facebook application must be restricted so that it cannot maliciously manipulate the rest of the Facebook display, access sensitive information (including the browser cookie) or take unauthorized actions on behalf of the user. As part of the Facebook isolation mechanism, the scripts used by applications must be written in a subset of JavaScript called FBJS [19] that restricts them from accessing arbitrary parts of the DOM tree of the larger Facebook page. The source application code is checked to make sure it contains valid FBJS, and some rewriting is applied to limit the application's behavior before it is rendered in the user's browser.

FBJS. While FBJS has the same syntax as JavaScript, a preprocessor consistently adds an application-specific prefix to all top-level identifiers in the code, isolating the effective namespace of an application from the namespace of other applicantions and of the rest of the Facebook page. For example, a statement document.domain may be rewritten to a12345_document.domain, where a12345_ is the application-specific prefix. Since this renaming will prevent application code from directly accessing most of the host and native JavaScript objects, such as the document object, Facebook provides libraries that are accessible within the application namespace. For example, the libraries include the object a12345_document, which mediates interaction between the application code and the true document object.

Additional steps are used to restrict the use of the special identifier this in FBJS code. The expression this, executed in the global scope, evaluates to the window object, which is the global scope itself. Without further restrictions, an application could simply use an expression such as this.document to break the namespace isolation and access the document object. Since renaming this would drastically change the meaning of JavaScript code, occurrences of this are replaced with the expression $FBJS.ref(this), which calls the function $FBJS.ref to check what object this refers to when it is used. If this refers to window, then $FBJS.ref(this) returns null.

Clever

Other, indirect ways that malicious content might reach the window object involve accessing certain standard or browser-specific predefined object properties such as __parent__ and constructor. Therefore, FBJS blacklists such properties and rewrites any explicit access to them in the code into an access to the useless property __unknown__. Since the notation o[e] denotes the access to the property of object o whose name is the result of evaluating expression e to a string, FBJS rewrites that term to a12345_o[$FBJS.idx(e)], where $FBJS.idx enforces blacklisting on the string value of e. Note that this technique is not vulnerable to standard obfuscation, because $FBJS.idx is run on the string obtained as the final result of evaluating e.

Finally, FBJS code runs in an environment where properties such as valueOf, which may access (indirectly) the window object, are redefined to something harmless.

2.2 Formalizing JavaScript Isolation

FBJS illustrates two fundamental issues with mashup isolation. (i) Regardless of the technique adopted to enforce isolation, the ultimate goal is usually very simple: make sure that a piece of untrusted code does not access a certain set of global variables (typically the DOM). (ii) While enforcing this constraint may seem easy, there are a number of subtleties related to the expressiveness and complexity of JavaScript.

Common isolation techniques include blacklisting certain properties, separating the namespaces corresponding to code in different trust domains, inserting run-time checks to prevent illegal accesses, and wrapping sensitive objects to limit their accessibility.

In the remainder of this paper, we study how combining run-time checks (analogous to $FBJS.idx and $FBJS.ref) with syntactic restrictions leads to expressive and provably secure subsets of JavaScript. While we use FBJS as a running example, the ideas illustrated in this paper also apply to JavaScript isolation in other settings.

3 Syntactic JavaScript Subsets

In this Section, we describe two secure subsets of JavaScript (first defined in [14]) that enforce isolation exclusively by means of syntactic restrictions, so that the user code is directly executed in the browser. The informal properties stated in this section are all fully supported by formal proofs available in [14]. These earlier results are included in the present paper both as background for modifications to them we present in Section 4, and as motivation for more permissive, run-time checks in the user code.

Two JavaScript Isolation Problems. If we can solve the problem of determining the set of properties that a piece of code can access, then we can isolate global variables by a simple syntactic check.

Our first subset, *Jt*, is designed to solve this problem without restricting the use of this. A JavaScript program can get hold of its own scope by way of this. For example, the expression var x; this.x=42 effectively assigns 42 to variable x. In fact, manipulating the scope leads to a confusion of the boundary between variables (which are properties of scope objects) and properties of regular object. Hence, *Jt* code must be prevented from using as property name any of the global variable names to be protected. In theory, this does not constitute a significant limitation of expressiveness. Effectively, *Jt* is a good subset for isolating the code of a single untrusted application from a library of functions whose names may be all prefixed by a designated string such as $. On the other hand, *Jt* is not suited to run several applications with separate namespaces, since the sets of property names used by each one needs to be disjoint.

To better support multiple applications, the next problem we have to solve is to prevent code from explicitly manipulating the scope, so that variables are effectively separated from regular object properties. To this end, we propose a refinement of *Jt*, which we call *Js*, that forbids the use of this. Hence, only the global variable names of each application, and of the page libraries, need to be distinct from one another. Moreover, *Js* enjoys the property that the semantics of its terms does not change after a safe renaming of variables. Hence, isolation can be enforced by an automatic rewriting pass (with suitable side-conditions).

3.1 Isolating property names: *Jt*

The problem of determining the set of properties names that may be accessed by a piece of code is intractable for JavaScript in general, because property names can be computed using string operations, as in o={prop:42}; m=*"pr"*; n=*"op"*; o[m + n], which returns 42. However, we can determine a finite set containing all accessed properties if we eliminate operations that can convert strings to property names, such as eval and e[e]. In doing so, we must also consider implicit access to native properties that may not be mentioned explicitly in the code. For example, the code fragment var o = { }; *"an␣"*+ o causes an implicit type conversion of object o to a string, by an implicit call to the toString property of object o, evaluating to the string *"an␣[object␣Object]"*. (If o does not have the toString property, then it is inherited from its prototype). Fortunately, the property names that can be accessed implicitly are only the natural numbers used to index arrays and a finite set of native property names [13].

Definition 1 *The set $\mathcal{P}_{nat}$ of all the property names that can be accesses implicitly is* $\{0,1,2,...\}$ $\bigcup$

$$\left\{\begin{array}{l} \mathit{toString, toNumber, valueOf, constructor, prototype,} \\ \mathit{length, arguments, message, Object, Array, RegExp} \end{array}\right\}$$

This list is exhaustive for an ECMA-262-compliant implementation. Other properties may be added to $\mathcal{P}_{nat}$ to account for browser-specific JavaScript extensions.

Our first subset, called *Jt*, is designed to make property access (whether for read or for write) decidable.

Definition 2 *Jt is defined as JavaScript minus all terms containing the identifiers eval, Function, hasOwnProperty, propertyIsEnumerable and constructor; the expressions e[e], e in e; the statement for (e in e) s.*

Since we consider checking for the existence of a property as a read access, we exclude from *Jt* also the e in e and for (e in e) s statements, even though they cannot be used to read the actual contents of the corresponding property.

From the usability point of view, the only serious restrictions of *Jt* are the lack of eval, and e[e]. The former, although has practical uses, is commonly considered *evil*, and is excluded from most subsets. The latter constitutes the natural way to access arrays elements. The dynamic subset *Jb* of Section 4.1 addresses this limitation.

Jt lends itself naturally to enforce *whitelisting* of properties and variable. It can also be used to enforce *blacklisting*. A *Jt* piece of code cannot read or write any variable or property, except for those in $\mathcal{P}_{nat}$, that does not appear explicitly in its code or in a function pre-loaded in the run-time environment (Theorem 1 of [14]). A simple static analysis can be used to screen the actual code for blacklisted properties. Since the initial JavaScript environment is defined by the specification, blacklisting can be effectively enforced as long as the code of any pre-loaded, user-defined function is known *a priori* (such is the case for Facebook).

3.2 Protecting the Scope: *Js*

In ECMA-262-compliant JavaScript implementations there are three ways to obtain a pointer to a scope object. The simplest way, supported by all JavaScript implementations, is by referencing the global object, for example by evaluating the expression this in the global scope. Another way to get a pointer to a scope object is by the statement

```
try {throw (function(){return this})}
catch(get_scope){scope=get_scope(); ...};
```

When the code is executed, the function thrown as an exception in the try block is bound to the identifier get_scope in a new scope object that becomes the scope for the catch block. Hence, when we call get_scope(), the this identifier of the function is bound to the enclosing scope object, which we make available to arbitrary code by saving it in variable scope. Although this behaviour conforms to the ECMA-262 standard, as far as we are aware Safari, Opera and Chrome are the only browser where this example works. Other browsers, such as for example Internet Explorer and Firefox bind the global object instead of the catch scope object to the this of the call to get_scope in the catch clause. Finally, we can get a pointer to a scope object by the expression

```
(function get_scope(x){if (x==0) {return this}
                     else {scope = get_scope(0); ...}})(1)
```

Here we use a named function expression. As this function executes, the static scope of the recursive function is a fresh scope object where the identifier get_scope is bound to the function itself, making recursion possible. When in the else branch we recursively call get_scope(0), then this is once again bound to the scope object, which is saved in scope for later usage. Once again, although ECMA-262-compliant, this example works only in Firefox and Safari. Internet Explorer, Opera and Chrome instead bind the global object to the this of get_scope in the recursive call.

We now define the subset *Js* which keeps variables distinct from property names by preventing manipulation of explicit scope objects (Theorem 2 of [14]).

Definition 3 *The subset Js is defined as Jt minus all terms containing this, with(e){s} and the identifiers valueOf, sort, concat and reverse.*

First and foremost the subset forbids any use of this, which can be used to access scope objects as described above. Just like in FBJS, we need to remove also the with construct because it gives another (direct) way to manipulate the scope. For example, the code var o = {x:null}; with(o){x=42} assigns 42 to the property o.x. Since we eliminate this and with, scope objects are only accessible via internal JavaScript properties which in turn can only be accessed as a side effect of the execution of other instructions. For example, the internal scope pointer of a scope object is accessed during identifier resolution, in order to search along the scope chain. However, its value is never returned as the result of evaluating a term. Similarly, the scope pointer stored in a function closure is never returned as a result. The internal @this property is returned only by the reduction rule for this, which cannot be triggered in *Js*, and by the native functions concat, sort or reverse of Array.prototype, and valueOf of Object.prototype. For example, the expression valueOf() evaluates to window (which is also the initial scope). By defining *Js* as a subset of *Jt*, we can blacklist these dangerous properties.

Closure under renaming The goal of variable renaming is to isolate the namespaces of different applications without requiring all of the property names to be distinct. Therefore, we want o.p to be renamed to a12345_o.p, and not to a12345_o.a12345_p. Due to implicity property access, and the fact that variables are effectively undistinguishable from properties of scope objects, the definition of variable renaming in JavaScript is subtle. In particular, one should not rename all the variables that correspond to native properties of a scope object, including the ones inherited via the prototype chain. These properties in fact have a predefined semantics that cannot be preserved by renaming. For example toString() evaluates to *"[object_Window]"*, but throws a "reference error" exception when evaluated as a12345_toString() after renaming.

Since *Js* does not contain with, only the global object, internal activation objects or freshly allocated objects (in the case of try-catch and named functions) can play the role of scope objects. Hence, the only (non-internal) inherited native properties are the ones present in Object.prototype, and the pre-defined properties of the global object. The complete set of properties that should not be renamed, denoted by $\mathcal{P}_{noRen}$ is:

$$\left\{\begin{array}{l}\text{NaN,Infinity,undefined,eval,parseInt,parseFloat,IsNaN,}\\ \text{IsFinite,Object,Function,Array,String,Number,Boolean,}\\ \text{Date,RegExp,Error,RangeError,ReferenceError, TypeError,}\\ \text{SyntaxError,EvalError,constructor,toString,toLocaleString,}\\ \text{valueOf,hasOwnProperty,propertyIsEnumerable,isPrototypeOf}\end{array}\right\}$$

Bowser implementations contain additional properties such as document,setTimeout,etc..

Let a *safe renaming* be a partial injective function that renames identifiers (not in $\mathcal{P}_{noRen}$) without introducing clashes. In [14], we prove that the intended meaning of a *Js* program does not change under renaming. *Jt* instead does not support the semantics preserving renaming of variables. The counterexample try {throw (function(){return this});} catch(y){y().x=42; x;} is valid *Jt* code that, according to the JavaScript semantics, evaluates to 42. If we rename x to $x, in the catch clause is rewritten to catch(y){y().x=42; $x} which raises an exception because $x is undefined.

3.3 Comparison with FBJS

A purely syntactic solution to the FBJS isolation problem, justified by our analysis, is to restrict Facebook applications to *Js*. While this could be an attractive solution for isolating user-supplied applications in contexts where code is written from scratch, it is more restrictive than the solutions proposed in Section 4. Since *Js* preserves safe renamings, we can separate the namespaces of different applications, and of the FBJS libraries, without altering their semantics. Since it is a subset of *Jt*, a simple syntactic check on application code guarantees that it cannot escape its namespace or access blacklisted properties (which need to include also browser-specific extensions such as caller, __proto__, getters, setters, etc.).

FBJS is more expressive than *Js*, because it includes a (sanitized) version of this and of the member access e[e] notation. On the other hand, FBJS does not correctly support renaming because it does not prevent explicit manipulation of the scope, and because it renames the properties in $\mathcal{P}_{noRen}$. The toString and try-catch counterexamples of Section 3.2 apply to FBJS as well. In Section 4 we shall propose better subsets that preserve renaming and are as expressive as FBJS.

4 Semantic JavaScript Subsets

In this Section, we present three JavaScript subsets that, by virtue of using run-time checks, are more expressive than *Jt* and *Js* yet still enforce strong insolation properties. The informal claims put forward in this Section are proven in the Appendix A.

JavaScript Isolation Problems Revisited. While the subset *Jt* of Section 3 makes it possible to statically determine all the properties accessed during execution of given code, this subset prevents e1[e2], which is often useful in programming. We therefore define a subset *Jb* with modified semantics (wrapper function) that allows e1[e2] and guarantees the weaker property that no program accesses properties that are explicitly blacklisted.

Our second semantic subset, called Js^s, is the semantic counterpart to *Js*. It solves the same problem of preventing the direct manipulation of scope objects, but it is more expressive, because Js^s programs can use this when it does not evaluate to a scope object. Disallowing this altogether would break many existing JavaScript libraries, and entail extensive rewriting.

The last semantic subset of this section, called *Jg* (first defined in [14]), solves the problem of isolating the window object, hence the global scope, while permitting to use this, even when it is bound to other scope objects. Indeed, we shall see that for some purposes the ability to explicitly manipulate the scope can be a desirable.

4.1 Blacklisting Properties: *Jb*

We now define the subset *Jb* that prevents user code from accessing any property included in a blacklist (or excluded from a whitelist). Note that if a property in $\mathcal{P}_{nat}$ is blacklisted it can still be accessed implicitly as a side effect.

Definition 4 *Let $\mathcal{B}$ be a set of blacklisted properties. The subset $Jb(\mathcal{B})$ is defined as Jt plus the construct e[e], minus all terms containing property names or identifiers in $\mathcal{B}$.*

In order for $Jb(\mathcal{B})$ to effectively achieve its isolation goal, $\mathcal{B}$ must contain at least the properties eval, Function and constructor blacklisted also by *Jt*, and a small number of private identifiers beginning with $, as explained below.

Enforcing *Jb*. The idea is to insert a run-time check in each occurrence of e1[e2] to make sure that e2 does not evaluate to a blacklisted property name. We transform every access to a blacklisted property of an object into an access to the property *"bad"* of the same object (we assume that $\mathcal{B}$ does not contain *"bad"*). A different option, clashing with the JavaScript *silent failure* philosophy is to throw an exception when a blacklisted property is accessed.

A faithful implementation of *Jb* is complicated by subtle details of the JavaScript semantics for the expression e1[e2]. In fact, the execution of e1[e2] goes through several steps involving evaluation of expressions to values, and possibly type conversions executed in a very specific order. Roughly, first e1 is evaluated to a value va1, then e2 to va2, then if va1 is not an object it is converted into an object o, and similarly if va2 is not a string it is converted into a string m:

$$\mathsf{e1[e2]} \longrightarrow \mathsf{va1[e2]} \longrightarrow \mathsf{va1[va2]} \longrightarrow \mathsf{o[va2]} \longrightarrow \mathsf{o[m]}$$

Each of these steps, which precede the actual access of property m in o, may raise an exception or have other side effects. Therefore, their execution order must be preserved.

The simplest and most efficient faithful implementation of this run-time check that we could find is to rewrite e1[e2] to e1[IDX(e2)], where IDX(e2) is the expression

```
($=e2,{toString:function(){return($=TOSTRING($),FILTER($))}})
```

The IDX code evaluates once and for all e2 to a value va2 that is saved in the variable $, and returns an object value va so that effectively the internal execution steps so far are

$$\mathsf{e1[IDX(e2)]} \longrightarrow \mathsf{va1[IDX(e2)]} \longrightarrow \mathsf{va1[va]} \longrightarrow \mathsf{o[va]}$$

Since va is an object and not a string, its toString method is invoked next. The expression TOSTRING($), which is defined as (new $String($)).valueOf() converts va2 into a string. In fact, the most direct way to convert a value into a string exactly as o[va2] would do, is by passing va2 to the original String constructor (which we assume to have saved in a variable $String), and invoking the valueOf method of the resulting string object. Finally, the expression FILTER($), defined as

```
($ == "$String" ? "bad" :
            ($ == "$" ? "bad" :
                    ($ == "constructor" ? "bad" : $))
```

uses nested conditional expressions to return the string saved in $ if it is not in the blacklist $\mathcal{B}$, and *"bad"* otherwise. For this filtering to work $, $String and constructor must always be blacklisted (and cannot appear as identifiers or property names in the source code). While these are the only blacklisted properties in the code above, it is straightforward to nest further conditional expressions to blacklist other properties. An alternative implementation of FILTER($) is the expression ($blacklist[$]?*"bad"*:$), where $blacklist is a (blacklisted) global variable containing an object with the properties to be blacklisted initialized to true. Note that all the properties of Object.prototype that are not overridden by $blacklist, and that do not contain values (such as null,0,"",false) that evaluate to false in a boolean context, will be automatically blacklisted. Hence, in our case $blacklist should actually be the object

```
{$:true,$String:true,$blacklist:true,
 toString:false,toLocaleString:false,...}
```

Our run time check is correct with either choice of FILTER.

Claim 1 *For every blacklist $\mathcal{B}$ containing the property names $ and $String, and for every JavaScript program $P \in Jb(\mathcal{B})$, the program $String=String;Q where Q is obtained by rewriting every instance of e1[e2] in P to e1[IDX(e2)] (adapted to include all of $\mathcal{B}$), behaves exactly like P when P accesses non-blacklisted properties. If P accesses a blacklisted property m of an object o, Q accesses instead o["bad"].*

In many practical cases, one can use simpler variants of IDX, sacrificing their correspondence to the original semantics of e1[e2].

When the order of the side-effects (including exceptions) caused by the evaluation of e1 and e2 can be ignored (say because the exceptions are not caught, or the expressions are side-effect free) we can simplify IDX(e2) to be ($=TOSTRING(e2),FILTER($)).

If e2 evaluates to an object va2, converting va2 to a string in the expression o[va2] involves invoking first its toString method, and if that fails, its valueOf method. The opposite happens when converting va2 to a string by the expression va2+"". If va2.toString() returns the same value as va2.valueOf(), or if the latter does not return a string, we can redefine TOSTRING(e2) in IDX to be the expression e2+"".

Combining these two simplifications, we can define IDX(e2) as ($=e2+"",($blacklist[$]?*"bad"*:$)). which is remarkably simple and efficient, and in particular implements correctly the JavaScript semantics in the most common case when the expression e2 is just a string or a number.

These latest variants of IDX do not enjoy Claim 1 because there are some (corner) cases in which their behaviour departs from that of e1[e2]. Yet, they are secure, because they still prevent any blacklisted property from being accessed.

4.2 Protecting the Scope: Js^s

In *Js*, we exclude this because it can be used to obtain a scope object. Now, we reinstate this and look for dynamic ways to prevent it to be bound to scope objects.

Definition 5 *The subset Js^s is defined as Jt minus all terms containing with(e){s}, the identifiers valueOf, sort, concat and reverse and property names or identifiers beginning with $.*

Js^s still excludes valueOf, sort, concat and reverse because those native functions can return the window object, if called in the appropriate context.

Enforcing Js^s. Unfortunately, it is not possible to enforce Js^s in an ECMA-262-compliant implementation of JavaScript. In the general case, there is no JavaScript expression that can detect if an object has an internal scope pointer, or test for its existence directly. Only code that has a handle to a scope object *that is present in the scope chain* can test such object and detect that it is a scope object. Recall the two ways of obtaining a scope object described in Section 3.2. In the case of the recursive function, the scope object that we obtain is active in the scope chain just below the activation object of the function returning its this. Therefore, we can insert a run-time check that detects it and replaces it with null. In the try-catch case instead the function returning its this is defined before the scope of the catch branch is created, so when at run-time the catch scope object is bound to the this, it is not active in the (static) scope chain of the function, and cannot be detected.

Hence, our implementation is useful to prevent direct scope manipulation in Firefox, which as discussed in Section 3.2 returns a scope only in the recursive function case, but not in Safari or other strictly ECMA-262-compliant implementations, which return the scope also in the try-catch.

To enforce Js^s in Firefox all we need to do is to initialize a global (blacklisted) variable $ with true, and replace each instance of this with the expression NOSCOPE(this), defined as (this.$=false,$?(delete this.$,this):(delete this.$,$=true,null)). When this is bound to the global object, the expression this.$=false overrides the global declaration, which needs to be restored by the $=true expression in the last branch of the conditional. In the case of a local scope object, this expression leaves behind a useless (but unharmful) local binding of $ to true. In the case of regular objects, the temporary variable $ is correctly removed.

Claim 2 *For every Firefox-JavaScript program $P \in Js^s$ that does not contain $, the program $=true;Q where Q is obtained by rewriting every instance of this in P to NOSCOPE(this), behaves exactly like P when P never accesses a this bound to a scope object. If P evaluates the expression this to a scope object then Q evaluates the same expression to null.*

4.3 Isolating the Global Object: *Jg*

In Section 4.2 we argued that, in general, it is not possible to detect a scope object in an ECMA-262-compliant JavaScript implementation. What we can do instead, is to prevent this to be bound to the global object. This solution is effectively equivalent to Js^s for Internet Explorer, because in that browser local scope objects cannot be accessed anyway, as discussed in Section 3.2. In the other browsers, keeping at least the global variables separate from generic property names still supports flexible isolation policies, as discussed for *Js*.

Arguably, the ability to manipulate scope objects directly may be a desirable feature. For example, it can be used to implement *open closures* which are a concept that we discovered after understanding direct scope manipulation via the examples given in Section 3.2. The idea is to write expressions that return a number of functions sharing some private state (like normal closures), *plus* an object that effectively embodies that shared state. A software architecture may distribute such functions, guaranteeing the encapsulation of the shared state, plus retain a handle to the shared state itself. In particular, in the case where the functions participating in the closure return results by updating shared variables, the shared state is ready to be used as a result object, without need to do any copying. For example, given

```
var oc = (function scope(x){if (x==0) {return this}
  else {shared=scope(0);shared.y=7;
  return [function(){y+=23},function(){y+=12},shared]}})(1)
```

the expression oc[0]();oc[1]();oc[2].y evaluates to 42. Traditional closures could encode less efficiently some of this behaviour by providing a dedicated function to access and update the shared state.

The subset *Jg* contains this and isolates the global object.

Definition 6 *The subset Jg is defined as Js plus the this expression and minus all terms containing property names or identifiers beginning with $.*

Note that *Jg* still excludes valueOf, sort, concat and reverse, that can return the window object.

Since the local scope can still be directly manipulated, in general variables can be confused with property names, and therefore variable renaming does not preserve the meaning of programs. Yet, this rarely happens accidentally, and does not constitute a security problem. On the other hand, since variables defined in the global scope *are* effectively separated from property names, *Jg* can be used to isolate the namespaces of different applications.

Enforcing *Jg*. In practice, the semantic restriction can be implemented by rewriting every occurrence of this in the user code into the expression NOGLOBAL(this) defined as (this==$?null;this). $ is a blacklisted global variable, initialized with the address of the global object.

Claim 3 *For every JavaScript program $P \in Jg$ that does not contain $, the program $=this;Q where Q is obtained by rewriting every instance of this in P to NOGLOBAL(this), behaves exactly like P if P does not access a this bound to the global object. If P evaluates this to the global object then Q evaluates NOGLOBAL(this) to null.*

4.4 Comparison with FBJS

We now compare our run-time checks with the corresponding ones in FBJS. Below, we denote by $FBJS^v_{09}$ the version of FBJS ~~deployed~~ on Facebook at the time of our analysis, in March 2009. The $FBJS^v_{09}$ $FBJS.ref function carries out ~~a check~~ equivalent to NOGLOBAL, plus some additional filtering needed to wrap DOM objects exposed to user code (we reserve to study the secure wrapping of libraries in future work). Since $FBJS is effectively blacklisted in $FBJS^v_{09}$, we are satisfied that ref prevents the this identifier to be evaluated to the window object, and the check is semantically faithful in the spirit of Claim 3.

The $FBJS^v_{09}$ $FBJS.idx function instead does not preserve the semantics of the member access notation, and as a result can be compromised. In the context of our explanation of Section 4.1, $FBJS.idx is in fact equivalent to the expression ($=e2,($instanceof Object||$blacklist[$])?"*bad*":$), where $blacklist is the object {caller:true,$:true,$blacklist:true}. The main problem is that, differently from our definition of IDX, the expression $blacklist[$]?"*bad*":$ converts va (that in principle could be an object) to a string two times. The object

```
{toString:function(){this.toString=function(){return "caller"};
                      return "good"}}
```

can fool the blacklisting by first returning the good property *"good"*, and then returning the bad property *"caller"* (we found a similar attack, which has since been fixed, on [11]). To avoid this problem, $FBJS^v_{09}$ inserts the check $ instanceof Object that tries to detect if $ contains an object. In general, this check is not sound. According to the JavaScript semantics, any object with a null prototype (such as Object.prototype) escapes this check. Moreover, in Firefox, Internet Explorer and Opera also the window object escapes the check.

In $FBJS^v_{09}$, Object.prototype and window are not accessible by user code, so cannot be used to implement this attack. We found instead that the scope objects described in Section 3.2 have a null prototype in Safari, and therefore we were able to mount attacks on the $FBJS.idx that effectively let user application code escape the Facebook sandbox. (See [14] for examples of exploit code, and a discussion on the security implications.) Shortly after our notification of this problem, Facebook has modified the $FBJS.ref function to include code that detects if the current browser is Safari, and in that case checks if this is bound to an object able to escape the instanceof check described above.

Unfortunately this solution is not very robust, and is unnecessarily restrictive. First, some browsers may have other host objects that have a null prototype, and that can be accesses without using this. Such objects could still be used to subvert $FBJS.idx, which has not been changed. Second, $FBJS.idx prevents objects to be used as arguments of member expressions. This restriction is unnecessary for the safety of blacklisting, as shown by our IDX.

Another minor problem with $FBJS.idx is that it deals inconsistently with the blacklisting of inherited properties such as toString. While the expression ({}).toString() is valid FBJS code returning *"[object␣Object]"*, the expression ({})[*"toString"*]() raises an exception because toString is implicitly blacklisted.This problem can be easily fixed, as described in Section 4.1, by setting $blacklist.toString=false.

5 Conclusions

We reviewed previous filtering methods for managing untrusted JavaScript and developed ways of replacing restrictive static code filters with more flexible run-time instrumentation that is implementable as source-to-source translation. We defined a subset with modified semantics (wrapper functions) that allows e1[e2] and guarantees that no program accesses properties that are explicitly blacklisted. Our second semantic subset prevents the direct manipulation of scope objects, but allows programs to use this when it does not evaluate to a scope object. Our third semantic subset isolates the window object, and hence the global scope, while permitting code to use this, even when it is bound to other scope objects. We have applied our results to analyze FBJS, which apart from some minor problems discovered by our analysis, has proven to be a remarkably sound and efficient practical JavaScript subset. We hope that our semantics-based study will convince developers of the value of programming language methods for evaluating language-based isolation.

Acknowledgments. Sergio Maffeis is supported by EPSRC grant EP/E044956 /1. Mitchell and Taly acknowledge the support of the National Science Foundation.

References

[1] Irem Aktug, Mads Dam, and Dilian Gurov. Provably correct runtime monitoring. In *Proc. of FM 2008*, volume 5014 of *LNCS*, pages 262–277. Springer, 2008.

[2] C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for JavaScript. In *Proc. of ECOOP'05*, pages 429–452, 2005.

[3] A. Barth, C. Jackson, and J.C. Mitchell. Securing browser frame communication. In *Proc. of USENIX Security*, 2008.

[4] Google Caja Team. Google-Caja: A source-to-source translator for securing JavaScript-based web. `http://code.google.com/p/google-caja/`.

[5] Douglas Crockford. ADsafe: Making JavaScript safe for advertising. `http://www.adsafe.org/`, 2008.

[6] ECMA International. ECMAScript language specification. stardard ECMA-262, 3rd Edition. `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf`, 1999.

[7] B. Eich. JavaScript at ten years. `http://www.mozilla.org/js/language/ICFP-Keynote.ppt`.

[8] A. Felt, P. Hooimeijer, D. Evans, and W. Weimer. Talking to strangers without taking their candy: isolating proxied content. In *Proc. of SocialNets '08*. ACM, 2008.

[9] D. Flanagan. *JavaScript: The Definitive Guide*. O'Reilly, 2006.

[10] P. Heidegger and P. Thiemann. Recency types for dynamically-typed, object-based languages. Proc. of FOOL'09, 2009.

[11] P. H.Phung, D. Sands, and A. Chudnov. Lightweight self protecting JavaScript. In *Proc. of ASIACCS 2009*. ACM Press, 2009.

[12] S. Maffeis, J.C. Mitchell, and A. Taly. Complete ECMA 262-3 operational semantics. `http://jssec.net/semantics/`.

[13] S. Maffeis, J.C. Mitchell, and A. Taly. An operational semantics for JavaScript. In *Proc. of APLAS'08*, volume 5356 of *LNCS*, pages 307–325, 2008. See also: Dep. of Computing, Imperial College London, Technical Report DTR08-13, 2008.

[14] S. Maffeis and A. Taly. Language-based isolation of untrusted Javascript. In *Proc. of CSF'09*, IEEE, 2009. See also: Dep. of Computing, Imperial College London, Technical Report DTR09-3, 2009.

[15] J. Pynnonen. Facebook script injection vulnerabilities. `http://seclists.org/fulldisclosure/2008/Jul/0023.html`.

[16] C. Reis, J. Dunagan, H. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-driven filtering of Dynamic HTML. *ACM Transactions on the Web*, 1(3), 2007.

[17] A. Sabelfeld and A. Askarov. Tight enforcement of flexible information-release policies for dynamic languages. Proc. of PCC'08, 2008.

[18] The FaceBook Team. FaceBook. `http://www.facebook.com/`.

[19] The FaceBook Team. FBJS. `http://wiki.developers.facebook.com/index.php/FBJS`.

[20] The FaceBook Team. FBML. `http://wiki.developers.facebook.com/index.php/FBML`.

[21] P. Thiemann. Towards a type system for analyzing javascript programs. In *Proc. of ESOP'05*, volume 3444 of *LNCS*, pages 408–422, 2005.

[22] P. Thiemann. A type safe DOM API. In *Proc. of DBPL'05*, pages 169–183, 2005.

[23] K. Vikram and M. Steiner. Mashup component isolation via server-side analysis and instrumentation. In *Proc. of W2SP'08*, 2008.

[24] D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript instrumentation for browser security. In *Proc. of POPL'07*, pages 237–249, 2007.

A Appendix: Correctness Proofs

In this Section we formally prove the correctness of our enforcement mechanisms. In order to prove the correctness, we make use of the operational semantics of JavaScript. The main purpose of this appendix is to substantiate the claims made in Section 4. The reader does not need to read this Appendix in order to understand the main body of the paper. Unless otherwise stated, below we assume that the semantics of JavaScript is compliant with the ECMA-262 standard.

A.1 Operational Semantics of JavaScript

We briefly summarize our formalization of the operational semantics of JavaScript [12, 13] based on the ECMA-262 standard [6], and introduce some auxiliary notation and definitions. In [13], we proved properties of JavaScript that address the internal consistency of the semantics itself, and memory reachability properties needed for garbage collection, but did not address the kind isolation properties.Discussion of the relation between this semantics and current browsers implementations appear in [13]

Our operational semantics consists of a set of rules written in a conventional meta-notation suitable for rigorous but (currently) unautomated proofs. Given the space constraints, we describe only the main semantic functions and some representative axioms and rules.

Semantic Functions and Contexts. Expressions, statements and programs each have a corresponding small-step semantic relation denoted respectively by $\xrightarrow{e}, \xrightarrow{s}, \xrightarrow{P}$. Each semantic function transforms a heap H, a pointer in the heap to the current scope l, and the current term being evaluated t into a new heap-scope-term triple.

The semantics of programs depends on the semantics of statements which in turn depends on the semantics of expressions which in turn, for example by evaluating a function, depends circularly on the semantics of programs. These dependencies are made explicit by contextual rules, that specify how a transition derived for a term can be used to derive a transition for a larger term including the former as a sub-term. The premises of each semantic rule are predicates that must hold in order for the rule to be applied, usually built of very simple mathematical conditions such set membership, inequality and semantic function application.

An atomic transition is described by an axiom. For example, the axiom $\mathsf{H,l,(v)} \longrightarrow \mathsf{H,l,v}$ describes that brackets can be removed when they surround a value (as opposed to an expression, where brackets are still meaningful). Contextual rules propagate such atomic transitions. For example, if program H,l,P evaluates to H1,l1,P1 then also H,l,@FunExe(l2,P) (an internal expression used to evaluate the body of a function) evaluates to H1,l1,@FunExe(l2,P1).

The rule below shows that: @FunExe(l,−) is one of the contexts eCp for evaluating programs.

$$\frac{\mathsf{H,l,P} \xrightarrow{P} \mathsf{H1,l1,P1}}{\mathsf{H,l,eCp[P]} \xrightarrow{e} \mathsf{H1,l1,eCp[P1]}}$$

Expressions. We distinguish two classes of expressions: internal expressions, which correspond to specification artifacts needed to model the intended behavior of user expressions, and user expressions, which are part of the user syntax of JavaScript. Internal expressions include addresses, references, exceptions and functions such as @GetValue,@PutValue used to get or set object properties, and @Call,@Construct used to call functions or to construct new objects using constructor functions.

Statements. Similarly to the case for expressions, the semantics of statements contains a certain number of internal statements, used to represent unobservable execution steps, and user statements that are part of the user syntax of JavaScript. A completion is the final result of evaluating a statement.

co ::= "("ct,vae,xe")"   vae::=&empty|va   xe::=&empty|x
ct ::= Normal | Break | Continue | Return | Throw

The completion type indicates whether the execution flow should continue normally, or be disrupted. The value of a completion is relevant when the completion type is Return (denoting the value to be returned), Throw (denoting the exception thrown), or Normal (propagating the value to be return during the execution of a function body). The identifier of a completion is relevant when the completion type is either Break or Continue, denoting the program point where the execution flow should be diverted to.

Programs. Programs are sequences of statements and function declarations.

P ::= fd [P] | s [P]   fd ::= function x "("[x~]"){"[P]"}"

As usual, the execution of statements is taken care of by a contextual rule. If a statement evaluates to a break or continue outside of a control construct, an SyntaxError exception is thrown (rule (i)). The run-time semantics of a function declaration instead is equivalent to a no-op (rule (ii)). Function (and variable) declarations should in fact be parsed once and for all, before starting to execute the program text. In the case of the main body of a JavaScript program, the parsing is triggered by rule (iii) which adds to the initial heap NativeEnv first the variable and then the function declarations (functions VD,FD).

$$\frac{\begin{array}{l}\mathsf{ct} < \{\mathsf{Break,Continue}\} \\ \mathsf{o = new\_SyntaxError()} \quad \mathsf{H1,l1 = alloc(H,o)}\end{array}}{\mathsf{H,l,(ct,vae,xe)\ [P]} \xrightarrow{P} \mathsf{H1,l,(Throw,l1,\&empty)}} \quad \text{(i)}$$

$$\text{H,l,function x ([x\~{}])\{[P]\} [P1]} \xrightarrow{P} \text{H,l,(Normal,\&empty,\&empty) [P1]} \quad \text{(ii)}$$

$$\frac{\begin{array}{l}\text{VD(NativeEnv,\#Global,\{DontDelete\},P) = H1}\\ \text{FD(H1,\#Global,\{DontDelete\},P) = H2}\end{array}}{\text{P} \xrightarrow{P} \text{H2,\#Global,P}} \quad \text{(iii)}$$

Native Objects. NativeEnv is the initial heap of core JavaScript. It contains native objects for representing predefined functions, constructors and prototypes, and the global object @Global that constitutes the initial scope, and is always the root of the scope chain. In web browsers, the global object is called window. For example, the global object defines properties to store special values such as &NaN and &undefined, functions such as eval and constructors to build generic objects, functions, numbers, booleans and arrays. Since it is the root of the scope chain, its @Scope property points to null. Its @this property points to itself. None of the non-internal properties are read-only or enumerable, and most of them can be deleted.

A.2 Preliminaries

We now define some notation and state some properties of the semantics that support the formal analysis of JavaScript subsets defined in Section 4.

A *state* S is a triple (H, l, t). We use the notation $\mathcal{H}(S)$, $\mathcal{S}(S)$ and $\mathcal{T}(S)$ to denote each component of the state. We denote by H_0 the "empty" heap, that contains only the native objects, and no user code. We use l_G to denote the heap address of the global object #Global. If a heap, a scope pointer and a term are well-formed then the corresponding state is also well-formed (see the Appendix of [14] for a formal definition). In [13], we show that the evaluation of well-formed terms, if it terminates, yields either a value or an exception (for expressions), or a completion (for statements and programs). A state S is *initial* if it is well-formed, $\mathcal{H}(S) = H_0$, $\mathcal{S}(S) = l_G$ and $\mathcal{T}(S)$ is a user term. A *reduciton trace* τ is the (possibly infinite) maximal sequence of states $S_1, \ldots, S_n, \ldots$ such that $S_1 \rightarrow \ldots \rightarrow S_n \rightarrow \ldots$. Give a state S, we denote by $\tau(S)$ the (unique) trace originating from S and, if $\tau(S)$ is finite, we denote by $Final(S)$ the final state of $\tau(S)$.

To ease our analysis, we add a separate sort mp to distinguish property names from strings and identifiers in the semantics. We make all the implicit conversions between these sorts explicit, by adding the identity functions Id2Prop: x ⟶ mp, Prop2Id: mp ⟶ x; Str2Prop: m ⟶ mp, Prop2Str: mp ⟶ m. The semantics already contained explicit conversion of strings to programs: ParseProg, ParseFunction, ParseParams. In order to keep track of the names appearing in a state S, we define functions that collect respectively the identifiers and the property names of the term and the heap of S.

$$\mathcal{N}^T_I(S) = \{x | x \in \mathcal{T}(S)\} \quad \mathcal{N}^T_P(S) = \{mp \mid mp \in \mathcal{T}(S)\}$$
$$\mathcal{N}^H_I(S) = \{x \mid x \in P,\ P \in \mathcal{H}(S)\}$$
$$\mathcal{N}^H_P(S) = \{mp \mid \exists l : mp \in \mathcal{H}(S)(l)\}$$
$$\mathcal{N}_I(S) = \mathcal{N}^T_I(S) \cup \mathcal{N}^H_I(S) \quad \mathcal{N}_P(S) = \mathcal{N}^T_P(S) \cup \mathcal{N}^H_P(S)$$

Finally, we define the set of all the identifiers and property names appearing in a state S by $\mathcal{N}(S) = \mathcal{N}_I(S) \cup \text{Prop2Id}(\mathcal{N}_P(S))$. From these definitions, follows that for any initial state S_0, $\mathcal{N}(S_0) = \mathcal{N}^T_I(S_0) \cup \mathcal{N}^H_P(S)$. $\mathcal{N}^H_P(S)$ is the set of property names present in the initial heap H_0. This is a fixed set, and will henceforth be denoted by $\mathcal{N}^0_P$.

We define *meta-call* a pair $(f, (args))$ where f is a semantic function or predicate appearing in the premise of a reduction rule, and $(args)$ is the list of its actual arguments as instantiated by a reduction step using that rule. For every state S, we denote by $\mathcal{C}_1(S)$ the set of the meta-calls triggered directly by a one step transition from state S. Since each meta-call may in turn trigger other meta-calls during its evaluation, we denote by $\mathcal{C}(S)$ the set of all the meta-calls involved in a reduction step. We denote by $\mathcal{F}_H$ the set of functions that can read or write to the heap: $\mathcal{F}_H = \{$Dot(H, l, mp), Get(H, l, mp), Update(H, l,mp), Scope(H, l, mp), Prototype(H, l, mp)$\}$,

Definition 7 *(Property access) For any state S, we define the set of all property names accessed during a single transition by $\mathcal{A}(S) \triangleq \{mp \mid \exists f \in \mathcal{F}_H\ \exists H, l : (f, (H, l, mp)) \in \mathcal{C}(S)\}$. In the case of a trace τ, $\mathcal{A}(\tau) \triangleq \bigcup_{S_i \in \tau} \mathcal{A}(S_i)$.*

In Sections 3 and 4, we considered syntactic subsets of JavaScript. A syntactic subset J is essentially a subset of JavaScript user terms. For a given subset J, we denote by $Initial(J)$, the set of all well-formed initial states for J. We denote by J^* the set

$$J^* = \{t' \mid t \in J \ \wedge\ \exists H, l : H_0, l_G, t \rightarrow H, l, t'\}$$

of all terms that are reachable by reducing terms in J. We denote by $Wf_J(S)$ the well-formedness predicate for a state in the subset J, defined exactly like $Wf(S)$ except that $Wf_{\mathcal{T}}(\mathcal{T}(S))$ instead of checking if a term is derivable by the grammar, checks if the term is in J^*.

A.3 Proof of Claim 1

In this Subsection we prove Claim 1, which states that given a black list $\mathcal{B}$, for all initial states S_0 in the set $Jb(\mathcal{B})$, for which the $\mathcal{T}(S_0)$ is appropriately rewritten, no property from the blacklist $\mathcal{B}$ is accessed. We start by giving a few notations and definitions that will be used in the lemmas and theorems that come later. We define $Jb^r(\mathcal{B})$ as the subset $Jb(\mathcal{B})$ where for every term $t \in Jb(\mathcal{B})$, the subexpression e1[e2] (if it is present in t) is replaced with e1[IDX(e2)]. We formalize the property that if the execution of a program P involves accessing property mp of some object then either $mp \in \mathcal{P}_{nat}$ or $mp \notin \mathcal{B}$ as follows:

Definition 8 *(Pb) Given a well-formed state $S \in Jb^r(\mathcal{B})$, $Pb(S)$ holds iff $\mathcal{A}(\tau(S)) \cap (\mathcal{B} \setminus \mathcal{P}_{nat}) = \emptyset$*

Theorem 1 *For all well-formed states S_0 in $Initial(Jb^r(\mathcal{B}))$, $Pb(S_0)$ holds.*

It is easy to see that theorem 1 proves Claim 1. In this rest of this Subsection we sketch out the proof of Theorem 1. We split the proof into the following two main steps

Step 1: We define a state predicate $Pb^{strong}(S)$ and show that for all initial states S_0 in JavaScript, $Pb^{strong}(S_0) \Rightarrow Pb(S_0)$

Step 2: For all initial states S_0 in the subset $Jb^r(\mathcal{B})$, $Pb^{strong}(S_0)$ holds.

A.3.1 Step (1)

Given a blacklist $\mathcal{B}$ we define the following whiteness predicate on states:

Definition 9 *(State Whiteness) For a well-formed state S in the subset $Jb^r(\mathcal{B})^*$, $White(S)$ is true iff $(\mathcal{N}_P^T(S_1) \cup \mathsf{Id2Prop}(\mathcal{N}_I(S_1))) \cap (\mathcal{B} \setminus \mathcal{P}_{nat}) = \emptyset$.*

Consider any reduction rule from the operational semantics. The general structure of such a rule is $\frac{(Premise)}{S_1 \rightarrow S_2}$. We define the following goodness property on rules.

Definition 10 *(Rule goodness) A reduction rule of the form $\frac{(Premise)}{S_1 \rightarrow S_2}$ is good iff for all applicable S_1, S_2*

$$White(S_1) \Rightarrow White(S_2)$$

Based on the above definition of rule goodness we try to enumerate the set of good rules. If the initial state is white and the final state is not white then it is necessarily the case that some additional property names or identifier names get dynamically generated during the particular reduction step. According to our semantics, for most of the reduction rules, all the property names and identifier names that appear in the final state are a subset of those that appear in the initial state. If any identifier present in the final state is not present in the initial state then it must have been obtained by conversion from a string value present in the initial state. Similarly if any property name that appears in the final state, does not appear as a property name or an identifier in the initial state and also does not appear in the set $\mathcal{P}_{nat}$, then it must have been obtained by conversion from a string value present in the initial state. Therefore we claim that if a rule is good then it must not involve any of the following conversions: (1) strings to property names: rule E−ctx−Str−Pname (2) strings to identifiers: rule N−Funparse−StrId; (3) strings to programs: rules N−Funparse−StrProg, E−Eval−StrProg.

The rule E−ctx−Str−Pname also includes the context l[−]. We argue that the context l[−] is not bad if the result of converting the string to a property name is outside the blacklist. In order to make the analysis simpler, we split the rule for the term l∗m in two:

$$\frac{\begin{array}{l}\mathsf{mp = convStrPname(m)}\\ \mathsf{mp\ !< Blacklist\ OR\ mp = \textit{"bad"}}\end{array}}{\mathsf{H,l,l1[m]} \longrightarrow \mathsf{H,l,l1{*}mp}} \quad [\mathsf{E{-}AccGood}]$$

$$\frac{\mathsf{mp = convStrPname(m)\ AND\ mp < Blacklist}}{\mathsf{H,l,l1[m]} \longrightarrow \mathsf{H,l,l1{*}mp}} \quad [\mathsf{E{-}AccBad}]$$

Therefore the rule E−AccGood is good as it applies to only those cases where the resulting property names are outside the blacklist. We define the set $\mathcal{R}^{good}$ as all rules minus the set { E−AccBad, E−ctx−Str−Pname, N−Funparse−StrId, N−Funparse−StrProg, E−Eval−StrProg}. The detailed description for these rules is given in Figure 1.

Lemma 1 *All reduction rules present in the set $\mathcal{R}^{good}$ are good.*

Proof. We divide the set of good rules into two sets: $\mathcal{R}^{good} \setminus \{E - AccGood\}$ and $\{E - AccGood\}$.
Case 1: $\mathcal{R}^{good} \setminus \{E - AccGood\}$
For all rules in this set, we make use of Lemma 1 from [14] which states that all rules of the form $\frac{(Premise)}{S_1 \rightarrow S_2}$ in the set $\mathcal{R}^{good} \setminus \{E - AccGood\}$ have the property:

$$\begin{aligned} \mathcal{A}(S_1) &\subseteq \mathcal{N}_P^T(S_1) \cup \mathcal{P}_{nat} \bigwedge && (1)\\ \mathcal{N}_I(S_2) &\subseteq \mathcal{N}_I(S_1) \bigwedge && (2)\\ \mathcal{N}_P^T(S_2) &\subseteq \mathcal{N}_P^T(S_1) \cup \mathsf{Id2Prop}(\mathcal{N}_I(S_1)) \cup \mathcal{P}_{nat} && (3) \end{aligned}$$

By definition,

$$White(S_1) \Rightarrow (\mathcal{N}_P^T(S_1) \cup \mathsf{Id2Prop}(\mathcal{N}_I(S_1))) \bigcap (\mathcal{B} \backslash \mathcal{P}_{nat}) = \emptyset$$

From conditions (2) and (3) we have,

$$\mathcal{N}_P^T(S_2) \cup \mathsf{Id2Prop}(\mathcal{N}_I(S_2)) \subseteq \mathcal{N}_P^T(S_1) \cup \mathsf{Id2Prop}(\mathcal{N}_I(S_1)) \cup \mathcal{P}_{nat}$$

Therefore,

$$\begin{aligned} (\mathcal{N}_P^T(S_2) \cup \mathsf{Id2Prop}(\mathcal{N}_I(S_2))) &\bigcap (\mathcal{B} \setminus \mathcal{P}_{nat}) \subseteq\\ (\mathcal{N}_P^T(S_1) \cup \mathsf{Id2Prop}(\mathcal{N}_I(S_1)) \cup \mathcal{P}_{nat}) &\bigcap (\mathcal{B} \setminus \mathcal{P}_{nat}) =\\ (\mathcal{N}_P^T(S_1) \cup \mathsf{Id2Prop}(\mathcal{N}_I(S_1))) &\bigcap (\mathcal{B} \setminus \mathcal{P}_{nat}) \end{aligned}$$

Figure 1. List of bad reduction rules

$$\text{StrP(\_) ::= \_ in l} \mid \text{\#OPhasOwnProperty.@Exe(l1,\_)} \mid \text{\#OPpropertyIsEnumerable.@Exe(l,\_)}$$

$$\frac{\text{mp = convStrPname(m)}}{\text{H,l,StrP(m)} \longrightarrow \text{H,l,StrP(mp)}}\ [\text{E}-\text{Ctx}-\text{Str}-\text{Pname}] \qquad \frac{\begin{array}{c}\text{ParseFunction(m) = P}\\ \text{H,Function(fun(x}^{\sim}\text{)P,\#Global) = H1,l1}\end{array}}{\text{H,l,@FunParse(x}^{\sim}\text{,m)} \longrightarrow \text{H1,l,l1}}\ [\text{N}-\text{FunParse}-\text{StrProg}]$$

$$\frac{\text{ParseParams(m1) = x}^{\sim}}{\text{H,l,@FunParse(m1,m2)} \longrightarrow \text{H1,l,@FunParse(x}^{\sim}\text{,m2)}}\ [\text{N}-\text{FunParse}-\text{StrId}]$$

$$\frac{\text{ParseProg(m) = P}}{\text{H,l,\#GEval.@Exe(l1,m)} \longrightarrow \text{H2,l,\#GEval.@Exe(l1,P)}}\ [\text{E}-\text{Eval}-\text{StrProg}]$$

$$\frac{\text{mp = convStrPname(m) AND mp} < \text{Blacklist}}{\text{H,l,l1[m]} \longrightarrow \text{H,l,l1}*\text{mp}}\ [\text{E}-\text{AccBad}]$$

Hence, using definition of state whiteness we can conclude that $White(S_1) \Rightarrow White(S_2)$.
Case 2: $\{E - AccGood\}$. The rule $\{E - AccGood\}$ is

$$\frac{\begin{array}{c}\text{mp = convStrPname(m)}\\ \text{mp !< Blacklist OR mp = \textit{"bad"}}\end{array}}{\text{H,l,l1[m]} \longrightarrow \text{H,l,l1}*\text{mp}}\ [\text{E}-\text{AccGood}]$$

Let $S_1 = H, l, l1[m]$ and $S_2 = H, l, l1 * mp$. The only additional property names in $(\mathcal{N}_P^T(S_2)\text{Uld2Prop}(\mathcal{N}_I(S_2)))\backslash(\mathcal{N}_P^T(S_1)\text{Uld2Prop}(\mathcal{N}_I(S_1)))$ would be the property name mp. The premise of the rule ensures that this property is not in the blacklist. Therefore $White(S_1) \Rightarrow White(S_2)$ follows in this case as well. □

We denote the set of all rules not in $\mathcal{R}^{good}$ as $\mathcal{R}^{bad}$. Finally, given a reduction trace τ, we define $\mathcal{R}(\tau)$ as the set of all axioms R_i used to derive the transitions $S_i \longrightarrow S_{i+1}$ in τ (for all i). We are now ready to define the property $Pb^{strong}(S)$

Definition 11 *(Pb^{strong}) For a given state well-formed S we define $Pb^{strong}(S)$ as true iff $\mathcal{R}(\tau(S)) \subseteq \mathcal{R}^{good}$.*

The above definition basically says that a state has the property Pb^{strong} if only reduction rules from the set $\mathcal{R}^{good}$ are involved during its reduction.

Lemma 2 *For all initial states S_0 in Jb^r, $Pb^{strong}(S_0) \Rightarrow Pb(S_0)$.*

Proof. If the initial state S_0 corresponds to a value then $Pt(S_0)$ is trivially true. Therefore we consider initial states S_0 which have at least one reduction step in their trace. Let $\tau^n(S_0)$ denote the n step partial reduction trace of the state S_0, that is, $\tau^n(S_0)$ consists of the first $n + 1$ terms of the sequence $\tau(S_0)$.

In order to prove the above theorem we prove that $Pt^{strong}(S_0)$ implies the stronger property:-
$\forall n \geq 1 : \mathcal{P}(S_0, n)$ is true, where $\mathcal{P}(S_0, n)$ is defined as

$$\mathcal{A}(\tau^n(S_0)) \cap (\mathcal{B} \setminus \mathcal{P}_{nat}) = \emptyset \quad (4)$$
$$White(S_n) \text{ is true.} \quad (5)$$

where we assume (without loss of generality) $\tau^n = S_0, S_1, \ldots, S_n$.
Clearly, for all S_0 which have at least one reduction step, $\forall n \geq 1 : \mathcal{P}(S_0, n)) \Rightarrow Pt(S_0)$.
Given that $Pt^{strong}(S_0)$ holds, we prove $\forall n \geq 0 : \mathcal{P}(S_0, n)$ by induction over n.

Base Case: $n = 1$. Let $\tau^1(S_0) = S_0, S_1$. By definition of the subset Jb^r, $White(S_0)$ holds. Since $Pt^{strong}(S_0)$ holds, the reduction rule that applies to S_0 has to be good. From goodness property of rules, $White(S_1)$ holds. From our semantics, for all reduction rules $\frac{(Premise)}{S_1 \rightarrow S_2}$ we know that:

$$\mathcal{A}(S_1) \subseteq \mathcal{N}_P^T(S_1) \cup \mathcal{P}_{nat} \quad (6)$$

Therefore $\mathcal{A}(S_0) \subseteq \mathcal{N}_P^T(S_0) \cup \mathcal{P}_{nat}$. By definition of state whiteness, it follows that $\mathcal{A}(S_0) \cap (\mathcal{B} \setminus \mathcal{P}_{nat}) = \emptyset$. Hence property $\mathcal{P}(S_0, 1)$ holds.

Induction hypothesis: Assume $\mathcal{P}(S_0, n)$ is true for $n = k$. Therefore we have

$$\mathcal{A}(\tau^k(S_0)) \cap (\mathcal{B} \setminus \mathcal{P}_{nat}) = \emptyset \quad (7)$$
$$White(S_k) \text{ is true.} \quad (8)$$

Induction Step: Consider $n = k + 1$. Let $\tau^{k+1}(S_0) = S_0, S_1, \ldots, S_k, S_{k+1}$. By definition, $\mathcal{A}(\tau^{k+1}(S_0)) = \mathcal{A}(\tau^k(S_0)) \cup \mathcal{A}(S_k)$. Using condition (6) we get $\mathcal{A}(S_k) \subseteq$

$\mathcal{N}_P^T(S_k) \cup \mathcal{P}_{nat}$. Therefore,

$$\begin{aligned}\mathcal{A}(S_k) \cap (\mathcal{B} \setminus \mathcal{P}_{nat}) \;&\subseteq\; (\mathcal{N}_P^T(S_k) \cup \mathcal{P}_{nat}) \cap \\ &\quad\; (\mathcal{B} \setminus \mathcal{P}_{nat})\end{aligned}$$

This is equivalent to

$$\mathcal{A}(S_k) \cap (\mathcal{B} \setminus \mathcal{P}_{nat}) \;\subseteq\; \mathcal{N}_P^T(S_k) \cap (\mathcal{B} \setminus \mathcal{P}_{nat})$$

Since $White(S_k)$ is true $\mathcal{N}_P^T(S_k) \cap (\mathcal{B} \setminus \mathcal{P}_{nat}) = \emptyset$. Therefore we have $\mathcal{A}(S_k) \cap (\mathcal{B} \setminus \mathcal{P}_{nat}) = \emptyset$. Combining this with condition (7) we get

$$\mathcal{A}(\tau^{k+1}(S_0)) \cap (\mathcal{B} \setminus \mathcal{P}_{nat}) \;=\; \emptyset \qquad (9)$$

From condition (8) we know that $White(S_k)$ is true. Therefore using Pb^{strong} and the goodness property of rules, $White(S_{k+1})$ is true. Combining this with condition (9), we get that the predicate $\mathcal{P}(S_0, k+1)$ is true. Therefore $\forall n \geq 1 : \mathcal{P}(S_0, n)$ is true by induction. □.

A.3.2 Step (2)

We need to show that for all initial states S_0 in the subset $Jb^r(\mathcal{B})^*$, $Pb^{strong}(S_0)$ holds. This is also the basis on which the subset $Jb^r(\mathcal{B})^*$ was obtained, that is, no term should ever move to a state where a rule from $\mathcal{R}^{bad}$ becomes applicable. We prove this property by defining a "goodness" property (inductive invariant) on heaps and terms such that: (1) For all states with a good heap and term, no reduction rule from $\mathcal{R}^{bad}$ applies. (2) Heap goodness and term goodness are always preserved during reduction.

Before defining these properties, we state a few notations. Let $l_{Function}, l_{eval}, l_{hOP}, l_{pIE}$ denote the heap addresses of the constructor Function and methods eval, hasOwnProperty and propertyIsEnumerable of Object.prototype.

Definition 12 *(State goodness for $Jb^r(\mathcal{B})$) We say that a state S is good, denoted by $Good_{Jb^r(\mathcal{B})}(S)$, iff the term is good and the heap is good. The conditions for term goodness and heap goodness are given as follows.*

Term goodness:

(1) *Structure of t does not contain any property name or identifier from the set $\mathcal{B} \cup$ {eval, Function, hasOwnProperty, propertyIsEnumerable constructor }.*

(2) *\$ only appears inside a subexpression of the form IDX(e) for some e.*

(3) *Structure of t does not contain any sub terms with any contexts of the form eforin(), pforin(), cEval(), FunParse() contexts and any constructs of the form e in e, for (e in e) s*

(4) *Structure of t does not contain any of the heap addresses $l_{Function}, l_{eval}, l_{hOP}, l_{pIE}$*

(5) *If Structure of t contains a sub-expression e1[e2] then for all well-formed states S such that $Good_{Jb^r(\mathcal{B})}(\mathcal{H}(S))$ holds and $\mathcal{T}(S)$ = e1[e2]: $Pb^{strong}(S)$ is true.*

Heap goodness:

$$\begin{array}{lcl}\forall l, p : H(l).p = l_{Function} & \Rightarrow & p = \textit{constructor} \\ & & \quad \vee\; p = \textit{Function} \\ \forall l, p : H(l).p = l_{eval} & \Rightarrow & p = \textit{eval} \\ \forall l, p : H(l).p = l_{hOP} & \Rightarrow & p = \textit{hasOwnProperty} \\ \forall l, p : H(l).p = l_{pIE} & \Rightarrow & p = \textit{propertyIsEnumerable} \\ \forall l, p : p \in H(l) \wedge & \Rightarrow & l = l_G \\ \quad isPrefix(\$, p) & & \end{array}$$

$isPrefix(\$, p)$ is true iff \$ is a prefix of p. The contexts pforin() and eforin are internal continuation contexts used to express the internal states obtained during the reduction of a for (e in e) s statement. FunParse() is an internal continuation context which is entered during a call to the Function constructor, in order to parse the argument string. In the rest of this Section, we will apply the predicate $Good_{Jb^r}$ to heaps, terms and states; the interpretation for each of them being the corresponding goodness definition.

Lemma 3 *For all well-formed states S_1 and S_2 in the subset $Jb^r(\mathcal{B})^*$, $S_1 \rightarrow S_2 \wedge Good_{Jb^r(\mathcal{B})}(S_1) \Rightarrow Good_{Jb^r(\mathcal{B})}(S_2)$*

Proof. We prove this lemma by an induction over the set of all reduction rules. Since state goodness holds for the initial state S_1, by Lemma 5 it is sufficient to consider only the set of good rules ($\mathcal{R}^{good}$)). All context rules which have a reduction in their premise form the inductive cases and the transition axioms form the base cases. For the base cases we prove the theorem by a detailed case analysis. For the inductive case, consider any context rule of the form $\frac{S_1 \rightarrow S_2}{C(S_1) \rightarrow C(S_2)}$ (Recall that if $S = (H, l, t)$ then $C(S) = (H, l, C(t)))$. We divide the set of reduction contexts into the following three cases:

(1) C = va[−] For any state $S_1 = (H_1, l_1, t_1)$, $\mathcal{T}(C(S_1))$ = va[t_1]. Therefore, by definition of state goodness, $Good_{Jb^r}(C(S_1))$ holds iff $Pb^{strong}(C(S_1))$ holds . By definition of Pb^{strong}, $C(S_1 \rightarrow C(S_2) \wedge Pb^{strong(C(S_1))} \Rightarrow Pb^{strong(S_2)}$. Therefore we have $Good_{Jb^r(\mathcal{B})}(C(S_1)) \Rightarrow Good_{Jb^r(\mathcal{B})}(C(S_2))$

(2) C = −[e] For any state $S_1 = (H_1, l_1, t_1)$, $\mathcal{T}(C(S_1)) =$ t_1[e]. Therefore, by definition of state goodness, $Good_{Jb^r}(C(S_1))$ holds iff $Pb^{strong}(C(S_1))$

holds. By definition of Pb^{strong}, $C(S_1 \rightarrow C(S_2) \wedge Pb^{strong(C(S_1))} \Rightarrow Pb^{strong(S_2)}$. Therefore we have $Good_{Jb^r(\mathcal{B})}(C(S_1)) \Rightarrow Good_{Jb^r(\mathcal{B})}(C(S_2))$

(3) All other reduction contexts. For any term t, t' and an appropriate C from this set, we have the following (easy to prove) propositions.

- *Proposition 1* $Good_{Jb^r}(C(t)) \Rightarrow Good_{Jb^r}(t)$
- *Proposition 2* $(\exists t : Good_{Jb^r}(C(t))) \wedge Good_{Jb^r}(t') \Rightarrow Good_{Jt}(C(t'))$

From the induction hypothesis we know that

$Good_{Jb^r}(S_1) \Rightarrow Good_{Jb^r}(S_2)$. Therefore,

$Good_{Jb^r}(\mathcal{T}(S_1)) \Rightarrow Good_{Jb^r}(\mathcal{T}(S_2))$ and

$Good_{Jb^r}(\mathcal{H}(S_1)) \Rightarrow Good_{Jb^r}(\mathcal{H}(S_2))$

For all states S,

$Good_{Jb^r}(\mathcal{H}(S)) = Good_{Jb^r}(\mathcal{H}(C(S)))$. Therefore,

$Good_{Jb^r}(\mathcal{H}(C(S_1))) \Rightarrow Good_{Jb^r}(\mathcal{H}(C(S_2)))$ holds.

As a result we only need to show

$Good_{Jb^r}(\mathcal{T}(C(S_1))) \Rightarrow Good_{Jb^r}(\mathcal{T}(C(S_2)))$. This can be shown by using Propositions 1 and 2 and the induction hypothesis: $Good_{Jb^r}(\mathcal{T}(S_1)) \Rightarrow Good_{Jb^r}(\mathcal{T}(S_2))$.

□

Lemma 4 *For all well-formed expressions* e_1 *and* e_2 *in the subset* Jb^r *such that* $Good_{Jb^r(\mathcal{B})}(e_1)$, $Good_{Jb^r(\mathcal{B})}(e_2)$ *holds, and for all well-formed states* S *such that* $Good_{Jb^r(\mathcal{B})}(\mathcal{H}(S))$ *holds and* $\mathcal{T}(S) = e1[IDX(e2)]$*;* $Pb^{strong}(S)$ *is true. In other words term goodness holds for* e1[IDX(e2)]

Proof. Let $S = (H, l, e1[IDX(e2)])$ be any state such that $Good_{Jb^r(\mathcal{B})}(H)$ holds. We need to show that $\mathcal{R}(\tau(S)) \subseteq \mathcal{R}^{good}$. According to our semantics.

$$\frac{\text{H,l,e1} \xrightarrow{*} \text{H1,l1,va1}}{\text{H,l,e1[IDX(e2)]} \xrightarrow{*} \text{H1,l1,va1[IDX(e2)]}} \quad [\text{E}-\text{eCgv}]$$

Since $Good_{Jb^r(\mathcal{B})}(H)$ and $Good_{Jb^r(\mathcal{B})}(e1)$ hold, $Good_{Jb^r(\mathcal{B})}(H, l, e1)$ is true and $Pb^{strong}(H, l, e1)$ is true. So all the transition axioms involved in $\mathcal{R}(\tau(H, l, e1))$ would be from the set of good rules. By Lemma 3, we get that $Good_{Jb^r(\mathcal{B})}(H_1)$ holds. Now we only need to argue that $\mathcal{R}(\tau(H1, l1, va1[IDX(e2)])) \subseteq \mathcal{R}^{good}$.
From our semantics we deduce that

$$\frac{\text{H1,l1,e2} \xrightarrow{*} \text{H2,l2,va2}}{\text{H1,l1,va1[IDX(e2)]} \xrightarrow{*} \text{H2,l2,va1[IDX(va2)]}} \quad [\text{E}-\text{eCgv}]$$

Again $Good_{Jb^r(\mathcal{B})}(H1)$ and $Good_{Jb^r(\mathcal{B})}(e2)$ hold, therefore $Good_{Jb^r(\mathcal{B})}(H1, l, e2)$ is true and $Pb^{strong}(H1, l, e2)$ is true. So all the transition axioms involved in $\mathcal{R}(\tau(H1, l, e2))$ would be from the set of good rules. By Lemma 3, we get that $Good_{Jb^r(\mathcal{B})}(H_2)$ holds. Now we only need to argue that $\mathcal{R}(\tau(H2, l2, val[IDX(va2)])) \subseteq \mathcal{R}^{good}$.
This can be done using a straightforward symbolic execution based on the operational semantics rules. We deduce that $H2, l2, val[IDX(va2)] \rightarrow H3, l3, l4 * mp$
where mp ∈ $\mathcal{B}$ or mp= "bad". In either case the axiom E−AccGood applies to the state and reduces it to a final value. Therefore all transition axioms involved in the entire trace are from the set of $\mathcal{R}^{good}$. Therefore $Pb^{strong}(S)$ is true. □

Lemma 5 *For all well-formed states* S *in the subset* $Jb^r(\mathcal{B})^*$ *such that* $Good_{Jb^r(\mathcal{B})}(\mathcal{T}(S))$ *is true, no reduction rule from* $\mathcal{R}^{bad}$ *applies to* S*.*

Proof. We prove this result by a detailed case analysis over the set of rules in $\mathcal{R}^{bad}$ and show that no rule from $\mathcal{R}^{bad}$ is applicable to any term with the term goodness property. □

Lemma 6 *For all well-formed states* S_0 *in* $Initial(Jb^r(\mathcal{B}))$*,* $Good_{Jb^r(\mathcal{B})}(S_0)$ *is true.*

Proof. For any initial state S_0, $\mathcal{H}(S_0)$ is the initial heap and only consists of native objects. Therefore from the semantics and the definition of heap goodness, we show that $Good_{Jb^r}(\mathcal{H}(S_0))$ holds. We show $Good_{Jb^r}(\mathcal{T}(S_0))$ by structural induction over the set of user terms in Jb^r is contained in the set of user terms Jt. The base case is straightforward. For the inductive cases, using the definition of Jb^r, we can show that conditions (1), (2) and (3) in the definition of term goodness hold. Condition 4 is trivially true for all inductive cases except e1[IDX(e2)]. In this case we use Lemma 4 to argue that term goodness holds for e1[IDX(e2)].
□

Combining Lemmas 2, 3, 5, and 6 we can prove Theorem 1.

Restatement of Theorem 1 *For all well-formed states* S_0 *in* $Initial(Jb^r(\mathcal{B}))$*,* $Pt(S_0)$ *holds.*

Proof. From Lemma 2 we obtain, $Pb^{strong}(S_0) \Rightarrow Pb(S_0)$. Therefore proving that $Pb^{strong}(S_0)$ holds is sufficient for proving this theorem. From Lemma 5, goodness property for a state implies that no reduction rule from the set $\mathcal{R}^{bad}$ applies to it. Thus showing that for all states $S \in \tau(S_0)$, $Good_{Jb^r}(S)$ holds is sufficient to prove the theorem. From Lemma 6, $Good_{Jb^r}(S_0)$ is true and from Lemma 3, state goodness is preserved during reduction.

Therefore goodness holds for all states in the trace $\tau(S_0)$. □

A.4 Proof of Claim 2

In this Subsection we prove Claim 2, which states that if P is a Firefox-JavaScript program in Js^s that does not contain \$, the program \$=true;Q where Q is obtained by rewriting every instance of this in P to NOSCOPE(this), behaves exactly like P if P does not access a this bound to the scope object. We assume that the Firefox-JavaScript semantics is the ECMA-262 semantics for javascript with the difference in the rule for 'this value assignment for function calls'. In particular, the corresponding operational semantics has the rule E−CallRefAct modified to the following:

$$\frac{\begin{array}{l}\text{Type(ln}*\text{m) = Reference}\\ \text{isActivation(H,ln) OR isCatch(H,ln)}\end{array}}{\text{H,l,ln}*\text{m([va}\tilde{\ }\text{])} \longrightarrow \text{H,l,@Fun(Global,ln}*\text{m[,va}\tilde{\ }\text{])}}\ [\text{E-CallRefAct-mod}]$$

where the predicate isCatch(H,ln) is true iff H(ln) is the "catch-scope" object, the definition for which is elaborated in section 3.2. If P evaluates this to the scope object then Q evaluates NOSCOPE(this) to null.

We now formalize this claim in terms of the operational semantics. The this property is accessed only by the rule E−This.

$$\frac{\text{Scope(H,l,@this)=l1}\quad \text{H,l1.@Get(@this)= l2}}{\text{H,l,this} \longrightarrow \text{H,l,l2}}\ [\text{E-This}]$$

For the sake of argument, we replace this rule by the rules below, that return null if @this points to a scope object, and the effective value of @this otherwise.

$$\frac{\begin{array}{l}\text{Scope(H,l,@this)=l1}\\ \text{H,l1.@Get(@this)=l2}\quad \text{@scope in H(l2)}\end{array}}{\text{H,l,this} \longrightarrow \text{H,l,null}}\ [\text{E-This-KO}]$$

$$\frac{\begin{array}{l}\text{Scope(H,l,@this)=l1}\\ \text{H,l1.@Get(@this)=l2}\quad \text{@scope notin H(l2)}\end{array}}{\text{H,l,this} \longrightarrow \text{H,l,va}}\ [\text{E-This-OK}]$$

The property Ps which implies isolation of all scope objects can be formalized as follows :

Definition 13 *(Ps) Given a state S, let $S' = Final(S)$. $Ps(S)$ holds iff* @Scope *is not in $\mathcal{H}(S')(\mathcal{V}(S'))$.*

We denote the Firefox-JavaScript semantics along with the modification for the semantics of this as the 'modified Firefox-JavaScript semantics'.

Theorem 2 *For all well-formed states S_0 in $Initial(Js^s)$, $Ps(S_0)$ holds for execution with respect to the modified Firefox-JavaScript semantics*

In order to prove this theorem we need some supporting lemmas and definitions. As in the proof of the earlier claim, we define a goodness property on the states and show that it is inductive, and then show that the state goodness property implies the property Ps.

Definition 14 *(State goodness for Js^s) We say that a state S is* good, *denoted by $Good_{Js^s}(S)$, iff it has the following properties*

(1) *Structure of $\mathcal{T}(S)$ does not contain any of* eval, Function, hasOwnProperty, propertyIsEnumerable, constructor, valueOf, sort, concat, reverse. *Also $\mathcal{T}(S)$ does not contain any identifiers or property names beginning with* \$.

(2) *Structure of $\mathcal{T}(S)$ does not contain any contexts of the form* eforin(), pforin(), cEval() FunParse() *or [ ] contexts and any constructs of the form* e in e, for (e in e) s *and* e[e].

(3) *Structure of $\mathcal{T}(S)$ does not contain any of the heap addresses $l_{Function}, l_{eval}, l_{hOP}, l_{pIE}, l_{valueOf}, l_{sort}, l_{concat}$ and $l_{reverse}$.*

(4) *If a heap address l is present in $\mathcal{T}(S)$ such that* @Scope *$\in \mathcal{H}(S)(l)$ is true, then l must appear inside one of the following contexts only :* Function(fun([x˜]){P},−); −.@Put(mp,va); l.@call(−,[va˜]); Fun(−,e[,va˜]); @ExeFPA(l,−,va); @FunExe(−,P); @with(−ln1,ln2,s); −∗mp.

Heap goodness

Let H denote $\mathcal{H}(S)$.

$$\begin{array}{lcl}
\forall l, l', p : H(l).p = l' \wedge & \Rightarrow & p = \text{@Scope}\\
\quad \text{@Scope} \in H(l') & & \vee\, p = \text{this}\\
 & & \vee\, p = \text{@FScope}\\
\forall l, p : H(l).p = l_{Function} & \Rightarrow & p = \text{constructor} \vee\\
 & & p = Function\\
\forall l, p : H(l).p = l_{eval} & \Rightarrow & p = \text{eval}\\
\forall l, p : H(l).p = l_{hOP} & \Rightarrow & p = \text{hasOwnProperty}\\
\forall l, p : H(l).p = l_{pIE} & \Rightarrow & p = \text{propertyIsEnumerable}\\
\forall l, p : H(l).p = l_{valueOf} & \Rightarrow & p = \text{valueOf}\\
\forall l, p : H(l).p = l_{concat} & \Rightarrow & p = \text{concat}\\
\forall l, p : H(l).p = l_{sort} & \Rightarrow & p = \text{sort}\\
\forall l, p : H(l).p = l_{reverse} & \Rightarrow & p = \text{reverse}\\
\forall l, p : p \in H(l)\, \wedge & \Rightarrow & l = l_G\\
\quad isPrefix(\$, p) & & \\
\$ \in H(l_G) & & \\
H(l_G).\$ = true & &
\end{array}$$

$isPrefix(\$, p)$ is true iff "\$" is a prefix of the property name p. Observe that if $Good_{Js^s}(S)$ holds then there is no l such that $\mathcal{V}(S) = l$ and @Scope $\in \mathcal{H}(S)(l)$).

Lemma 7 *For all well-formed states S_1 and S_2 in the subset Js^{s*}, $S_1 \to S_2 \wedge Good_{Js^s}(S_1) \Rightarrow Good_{Js^s}(S_2)$*

Proof. We prove this lemma by an induction over the set of all reduction rules. We consider only those reduction rules that apply to good states. All context rules which have a reduction in their premise form the inductive cases and the transition axioms form the base cases. The proof is on same lines as that of lemma 4 in [14]. □

Lemma 8 *For all well-formed states S_0 in $Initial(Js^s)$, $Good_{Js^s}(S_0)$ is true.*

Proof. Similar to the proof for lemma 5 in [14]. Using the definition of Js^s, we show that $Good_{Js^s}(\mathcal{T}(S_0))$ is true. Using the semantics and the definition of heap goodness, we show that $Good_{Js^s}(\mathcal{H}(S_0))$ holds for the initial heap. □

Combining lemmas 7 and 8 we have the following proof for theorem 2.

Proof of Theorem 2 : From lemma 7, the state goodness property implies that the corresponding term can never be the address of a scope object. From lemma 8, state goodness holds for all initial states and from lemma 7, state goodness is preserved under reduction. Combining these facts, we get that all states present in $\tau(S_0)$ are good and therefore the term part for none of them would be an address of a scope object. □

We will now state and prove Theorem 3, which will show that rewriting `this` to `NOSCOPE(this)` correctly implements the rules `E−This−KO` and `E−This−OK` in terms of the original semantics with the rule `E−This`. These results together prove our claim.

In order for a rewritten program Q to behave exactly like the original program but with the modified firefox-JavaScript semantics we need to define `$= true` in the beginning. In order to formalize this we define the subset $Initial^r(Js^s)$ as the set of states $Initial(Js^s)$ but with the heap having an additional property `$` in the global object, which is set to `true`.

Since*Js* is a subset of *Jt*, by Theorem 1 of [14], if `$` is not present as an identifier in a program then the property `$` can never get accessed. Using this it is easy to show that lemmas 7 and 8 and hence theorem 2 are true for the modified set of initial states- $Initial^r(Js^s)$.

Theorem 3 *For all states $S_1 = (H_1, l_1, t_1)$ such that $Good_{Js^s}(H_1) \bigwedge \mathcal{T}(S_1) = NOSCOPE(this)$, there exists a state $S_2 = (H_1, l_1, va)$ such that*

- *$S_1 \to^* S_2$ in the unmodified Firefox-JavaScript semantics*
- *$S_1' = (H_1, l_1, this) \to S_2$ in the modified Firefox-JavaScript semantics with rules `E−This−KO` and `E−This−OK`*

Proof. We prove this theorem by a symbolic execution of the semantic rules. Due to space considerations we will sketch out only the main steps of the symbolic execution. Recall that `NOSCOPE(this)` is the expression :

```
(this.$=false,$?(delete this.$,this):(delete this.$,$=true,null))
```

We consider the following three cases

- Case 1 : `this` returns the address of a scope object, say l_{scp} which is along the current scope chain. The modified Firefox-JavaScript semantics would therefore reduce S_1' to $(H_1, l_1, null)$ by rule `E−This−KO`.

  In the unmodified semantics,

  (1) Executing `this.$= false` would set the property `$` of the object at l_{scp} to false. Therefore the heap after this statement would be $H_1^1 = H_1[l_{scp}.\$ = false]$.

  (2) The conditional `$?` in the next step would resolve to the else branch because the identifier `$` would resolve to $l_{scp} * \$$ since l_{scp} shadows the global object. The heap after this statement would be $H_1^2 = H_1^1$.

  (3) Within the else branch, `delete this.$` will delete property `$` from object at l_{scp}. The next expression `$=true` will resolve to $l_G * \$ = true$ and therefore this statement will amount to setting the property `$` of the global object back to true. The heap after this statement would be same as the original one, that is, $H_1^3 = H_1$.

  (4) Finally, the value `null` is returned and so the final state would be $(H_1, l_1, null)$.

  The final state obtained after the reduction of S_1' under the modified Firefox-JavaScript semantics is also $(H_1, l_1, null)$. Thus the theorem is true in this case.

- Case 2 : `this` returns the address of a non scope object, say l_o. The modified Firefox-JavaScript semantics would therefore reduce S_1' to (H_1, l_1, l_o) by rule `E−This−OK`.

  In the unmodified semantics,

  (1) Executing `this.$= false` would set the property `$` of the object at l_o to false. Therefore the heap at this state would be $H_1^1 = H_1[l_o.\$ = false]$.

  (2) The conditional `$?` in the next step would resolve to the if branch because the identifier `$` would resolve to $l_G * \$$. This is because $Good_{Js^s}(H_1)$ is

true and hence for heap $H_1^1 = H_1[l_o.\$ = false]$, the only object in the current scope chain which has the property $ would be the global object. The heap after this statement would be $H_1^2 = H_1^1$.

(3) Within the else branch, delete this.$ will delete property $ from object at l_o. The next expression $=true will resolve to $l_G * \$ = true$ and therefore this statement will amount to setting the property $ of the global object back to true. The heap after this statement would be back to the original one, that is, $H_1^3 = H_1$.

(4) Finally, the value l_o is returned and so the final state would be (H_1, l_1, l_o).

The final state obtained after the reduction of S_1' under the modified Firefox-JavaScript semantics is also (H_1, l_1, l_o) . Thus the theorem is true in this case.

- Case 3 : this returns the address of a scope object, say l_{scp} which is NOT along the current scope chain.

  According to the original JavaScript semantics the only case in which this can happen is when the @this property of the current activation object points to a "catch scope" object. However as explained earlier in section 3, in the Firefox-JavaScript semantics this cannot happen because of the rule E−CallRefAct−mod. Hence this case does not apply.

□

A.5 Proof of Claim 3

In this Subsection we prove Claim 3, which states that if P is a JavaScript program in Jg that does not contain $, the program $=this;Q where Q is obtained by rewriting every instance of this in P to NOGLOBAL(this), behaves exactly like P if P does not access a this bound to the global object. If P evaluates this to the global object then Q evaluates NOGLOBAL(this) to null.

First of all, we need to formalize this claim in terms of the JavaScript operational semantics. The this property is accessed only by the rule E−This.

$$\frac{\text{Scope(H,l,@this)=l1} \quad \text{H,l1.@Get(@this)=\#Global}}{\text{H,l,this} \longrightarrow \text{H,l,null}} \quad \text{[E-This]}$$

As in the previous section, for the sake of argument, we replace this rule by the rules below, that return null if @this points to the global object, and the effective value of @this otherwise.

$$\frac{\text{Scope(H,l,@this)=l1} \quad \text{H,l1.@Get(@this)=\#Global}}{\text{H,l,this} \longrightarrow \text{H,l,null}} \quad \text{[E-This-KO]}$$

$$\frac{\begin{array}{c}\text{Scope(H,l,@this)=l1} \\ \text{H,l1.@Get(@this)=va} \quad \text{va!=\#Global}\end{array}}{\text{H,l,this} \longrightarrow \text{H,l,va}} \quad \text{[E-This-OK]}$$

The property Pg which implies isolation of the global object can be formalized as follows :

Definition 15 *(Pg) Given a state S, let $S' = Final(S)$. $Pg(S)$ holds iff $\mathcal{V}(Final(S)) \neq l_G$.*

We denote the ECMA-262compliant JavaScript semantics with the modification for the semantics of this as the 'modified JavaScript semantics'. By Theorem 3 of [14], we have that in modified JavaScript semantics, no program P ever evaluates to the global object.

Restatement of Theorem 3 of [14] *For all well-formed states S_0 in Initial(Jg), $Pg(S_0)$ holds, under the modified JavaScript semantics.*

We will now state and prove theorem 4, which will show that rewriting this to NOGLOBAL(this) correctly implements the rules E−This−KO and E−This−OK in terms of the original semantics with the rule E−This. These results together prove our claim.

In order for a rewritten program Q to behave exactly like the original program but with the modified JavaScript semantics we need to define $= l_global in the beginning. In order to formalize this we define the subset $Initial^r(Js^{sr})$ as the set of states $Initial(Js^{sr})$ but with the heap having an additional property $ in the global object which is set to the address of the global object.

As in the previous section, we define a goodness property on the states and show that the during the execution of NOGLOBAL(this), goodness of the initial heap implies goodness of the final state. We consider the definition of states goodness $Good_{Jg}(S)$ as mentioned in definition 17 in [14]. We refine this definition by conjuncting it with two conditions :

(1) $\forall l : \$ \notin H(l)$

(2) $\mathcal{T}(S)$ does not contain any identifier or property name beginning with "$".

Since Jg is a subset of Jt, by Theorem 1 of [14], if $ is not present as an identifier in any program then the property $ can never get accessed. Using this it is easy to show that lemmas 8,9 and hence theorem 3 in [14] are true even with the refined definition of state goodness and the modified set of initial states $Initial^r(Jg)$.

Theorem 4 *For all states $S_1 = (H_1, l_1, t_1)$ such that $Good_{Jg}(H_1)$ holds and $\mathcal{T}(S_1) = NOGLOBAL(this)$, there exists a state $S_2 = (H_1, l_1, va)$ such that*

- *$S_1 \to^* S_2$ in the unmodified JavaScript semantics*

- *$S_1' = (H_1, l_1, this) \rightarrow S_2$ in the modified JavaScript semantics with rules E−This−KO and E−This−OK*

Proof. We prove this theorem by a symbolic execution of the semantic rules. Due to space considerations we will only sketch out the main steps of the symbolic execution. Recall that NOGLOBAL(this) is the expression : (this==$?null;this). $.
We consider the following two cases :

- Case 1 : this returns the address of the global object (l_G). The modified JavaScriptsemantics would therefore reduce S_1' to the state $(H_1, l_1, null)$ by rule E−This−KO.

  In the unmodified semantics

  (1) The conditional this==$? would resolve to $this == l_G * \$$. This is because the initial heap H_1 is good and therefore only the global object will have the $ property set to the address of the global object. Since this resolves to l_G, the conditional would resolve to the if branch. The heap obtained after this statement would be $H_1^1 = H_1$.

  (2) Within the if branch, the value returned would be null and therefore the final state obtained would be $(H_1, l_1, null)$.

  The final state obtained after the reduction of S_1' under the modified JavaScript semantics if also $(H_1, l_1, null)$. Thus the theorem is true in this case.

- Case 2 : this returns the address of an object, say l_o, which is different from the global object. The modified JavaScriptsemantics would therefore reduce S_1' to the state (H_1, l_1, l_o) by rule E−This−OK.

  In the unmodified semantics,

  (1) As in the previous case, the conditional this==$? would resolve to $this == l_G * \$$. This is because the initial heap H_1 is good and therefore only the global object will have the $ property set to the address of the global object. Since this resolves to l_o, the conditional would resolve to the else branch. The heap obtained after this statement would be $H_1^1 = H_1$.

  (2) Within the else branch, the value returned would be l_o and therefore the final state obtained would be (H_1, l_1, l_o).

  The final state obtained after the reduction of S_1' under the modified JavaScript semantics if also (H_1, l_1, l_o). Thus the theorem is true in this case as well.

□

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 9

Why is it important to prevent access to scope objects?

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

Top // **6.858 home** // *Last updated Saturday, 29-Sep-2012 10:55:47 EDT*

10/3

Paper Question 9

Michael Plasmeier

Restricting access to scope objects is important, because otherwise the application code could use them to access global variables on the page. Facebook only wanted an applications' JavaScript to interact with itself inside an application box and through approved APIs – not having access to the entire page.

6.858

10/3

Sandboxing

Javascript

~~Sandboxing Javascript~~

Last time Run out of time for qs → static analysis

Q: Do people use?

- Not really what we looked at

But for C, Java

Not prelevant in Python, PHP

L hard to build static analysis for

People care much more about false ⊕

Then false ⊖

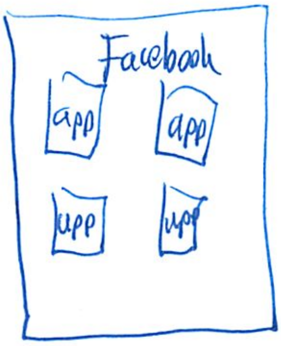
Devs don't want to wade through

errors that aren't

2

Today: How do we isolate JavaScipt?

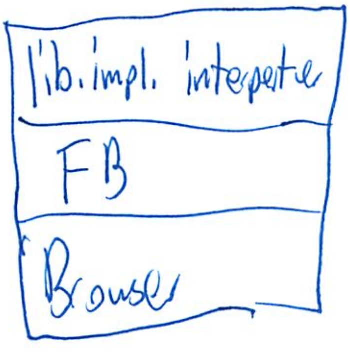
One page



- Old FB Profile
- iGoogle
- Advertising -> banner ad on NYT.com

Ad, app, widget Should not have access to facebook.com Origin

Approach 1 Run an interperter



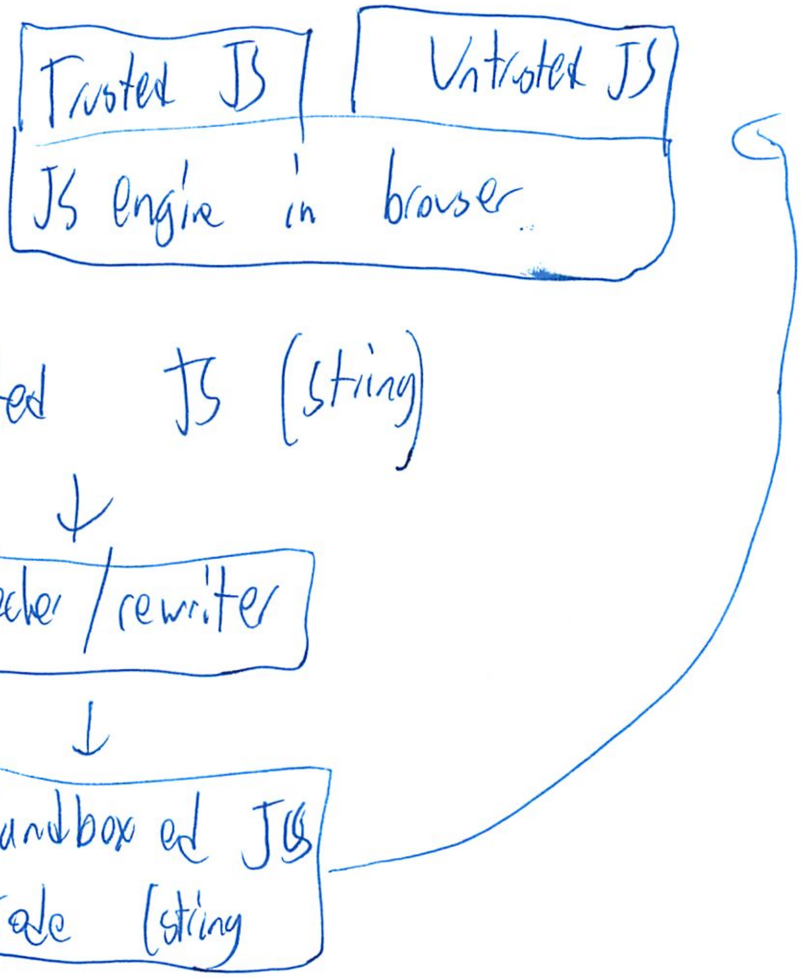
3

Narcissus - interperter of JS written in JS

Slow

like X86 virtilization → QMU

Approach 2 Lang level Isolation (this paper)



$\text{alert}(\underbrace{\text{document}}.\text{cookie});$

↑

```
var x = {cookie : 5}
x.cookie
```

accessing document is bad

So no accesses

JS is kinda like capability

↳ need to have ben given ref to object

1) Prohibit sensitive names

- document
- window
- etc
- eval(string)
- but hard to run multiple diff namespaces
- does not create diff protection domains

2) Renames variables

```
var x → var a12345_x;
```

So rewriter issues

x.cookie ⇒ a_12345_x.cookie

Then we would have

alert(a_12345.document);
Which is not sensitive

But one app should not be able to guess another's prefix –
└ is taken care of

alert(a2_x)

↳ rewritten
alert(a1_a2_x)

Can have custom variables in

- eval(s)
- ~~set_time out(s, t)~~
  setTimeout(s, t)

So prohibit these

So prefix does not need to be secret

But must be unique

or else share a sandbox

(like a uid in Linux)

~~First make sure~~

Special object → "this" ← implicit

like Python's self. ← explicit

but we'll have al_this

Won't work!

```
o = { p: 42,
      f: function() {
           return this.p;
      }
};
```

Something more clever is needed

1. Remove this all togehter

   but this breaks a lot of JS libraries

if in object

↳ this gives ref to obj

but otherwise

↳ this gives global object "window"

```
var x=5;
var y = this;

y.x → 5;

y.document → doc
y.document → cookie
```

How do we decide if this is the case?

(missed)

Runtime instrumentation

this → $FBJS.ref(this)

↑ know is safe if wrapped in ref object

```
fun ref(x) {
    if (x == window) { return null;}
    return x
}
```

Note $ is not special

but convention is to use it for auto generated code

⑨

```
with (al_o) {
    x = 5              → o.x → 5
                         al_o.x = 5
    al_x = 5
  }     ↓
 o.x    al_o.al_x = 5
 ↓
 al.o.x
```

If let JS use attributes
What do you rewrite?
∴ with is a special, built in fn ???
Use obj effectively as a stack frame

So everything prefixed?

```
with (al_o) (
    al_x = 5
al_o.al_x;
```

rewrite everything?

But now everyone has their own fields?

Everyone would need lib?

JS built-in methods assume some things

Paper tries to keep scope and non-scope obj seperate!

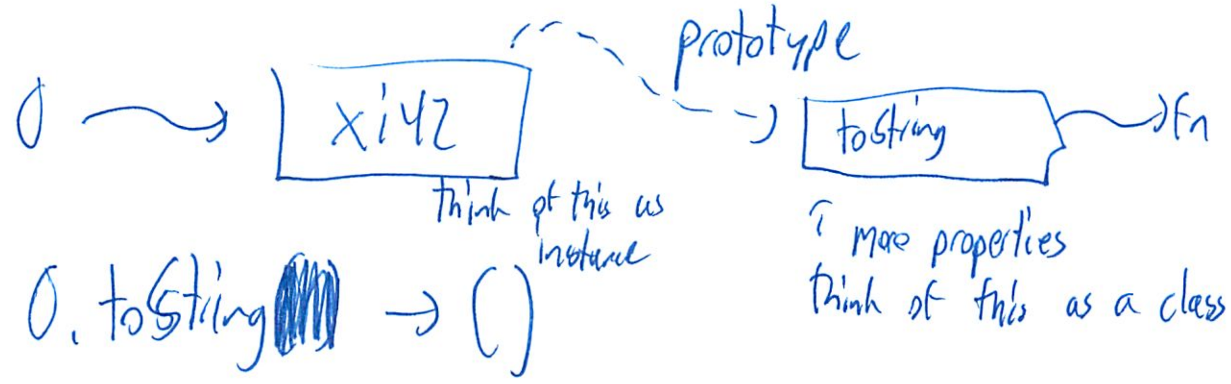
So prohibit with() statement

So only scope obj are rewritten

Non scope objects can only be accessed w/ $\overset{\text{dot}}{.}$ & $\overset{\text{sq brackets}}{[\ ]}$

But also implicitly provided shared state

How JS objects work

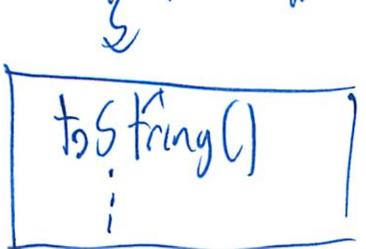


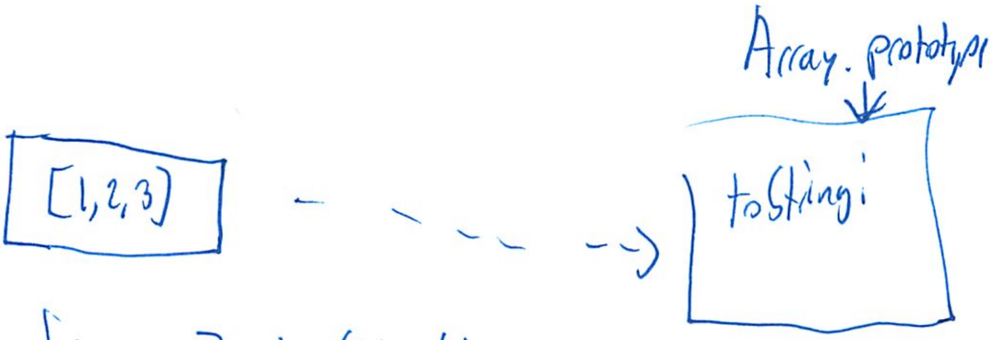
So if don't find it in instance, follow the prototype link

It will contiue following prototype chain
└ how JS does inheritance

Why is this problematic?
Pretty much all obj are writable

Array.prototype





[1,2,3].toString()

Array.prototype.toString()
    function() { return "x";}

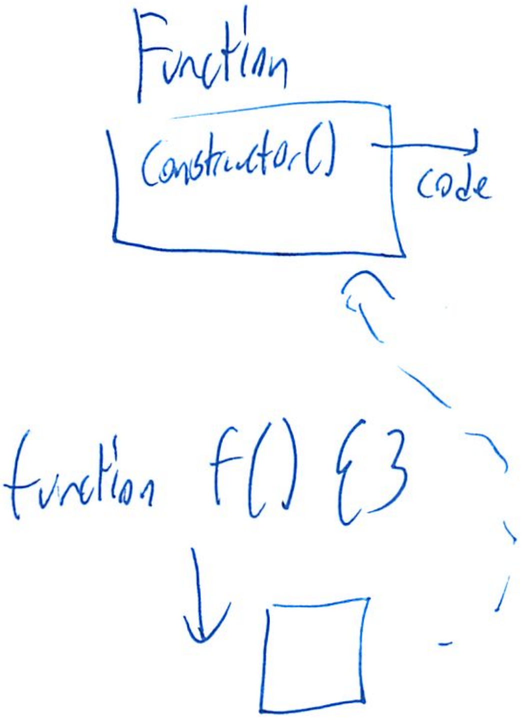
How do you prevent?

al Array.prototype.toString =
    function() { return "x";}

But many browsers allow
    [1,2].__proto__
That gives you prototype
not really in JS standard

So these systems look at certain methods + blacklist them

Functions are instances of Function class



f = Function("2+3")

f(); → 5

f. constructor ≊ eval

↑ must blacklist many of these attributes

What other supprising things show up when we block access to attributes

```
f['constructor'].x
f[x] → f[$FBJS.idx(x)]
```

```
function  idx(x)  {
   if  (x == "constructor")
       {return "bad";}

   return x;
}
```

I want to prevent functions from being used as an attribute

```
c=1
s={toString: function() {
             if (c==0){return "constructor";}
             else { c--; return "nice";}
   }    }
```

So can see how idx executes
it compares them using toString method

How do we fix?

x = x.toString()
? its a regular fn
So it can return anything we want

JS has === ~ Same ref to same exact object

(missed)

String(x) → string.

So use x = String(x)
not x = x.toString()

We know String is safe b/c it has not been prefixed
Code prefixes other stuff

One downside

- every trusted support routine must be bulletproof
- Check your arguments

But to do useful stuff
need to give it ~~and~~ access to some dom elements

But

i) e.parent

ii) e.innerHTML = "<script> ... </script>"

hard to statically detect since you can encode

e.addElement(e2)

Passing obj b/w trusted + untrusted code is dangerous and error prone

Each JS interpeter has its own quirks

details matter a lot

~~Portable code~~

Secure, Portable code is more important

So FB is depreciating this

Now doing sep origins

Running JS code in that origin

~~Cle~~ Then use message passing passMessage()

Less convienent, but more secure

Student: Reminds her of OkWS proxy server

limits # of interactions you can have

Perfomance has gotten much better

10/3

Sandboxing Javascript

Leftover question from last time: is static analysis actually used?
  Quite popular for Java, C: sparse/smatch, Coverity, etc.
  Less popular for Python, PHP in practice.
  Works well for bugs that have well-defined mistake patterns.
    NULL pointer checks, some buffer overflows, ..
  Interesting lesson: false positives often bigger problem than false negs.

What's the goal of this paper?
  Execute untrusted Javascript code in isolation.
  More specifically, origin A wants to run some Javascript code,
    without giving it all of the privileges of origin A.

Why would anyone want to execute untrusted Javascript code?
  Ads.
  Mashup applications.
  Third-party apps in an integrator site.
  Third-party library (e.g., spell-checker, text editor, etc).

How should we sandbox Javascript?
  There's a few ways to do it.
  The paper is one route, taken by Facebook's FBJS and Yahoo's AdSafe.
  Google's Caja project is similar in some ways, although it tends to give
    acecss to virtualized objects instead of giving access to real objects.
  Lab 6 will involve Javascript sandboxing similar to FBJS/AdSafe.
  This approach has some advantages, but is also hard to get right.
  Will look at other approaches too.

Approach 1: use an interpreter to safely run untrusted code.
  The fact that this untrusted code happens to be JS is almost irrelevant.
    E.g., http://en.wikipedia.org/wiki/Narcissus_(JavaScript_engine)
  Our interpreter could easily provide special objects -- virtual DOM, etc.
  +: Conceptually clean.
  -: Poor performance.

Approach 2: language-level isolation (this paper's plan).
  Almost think of this as an optimization on above plan (especially Caja).
    Don't write our own Javascript interpreter.
    Instead, carefully run untrusted code in the existing interpreter.
    Need to be sure we can control what the code can do.

  What does it take to reuse the same interpreter but still get isolation?
    Need to define a more precise goal.
    Starting point: run Javascript code, as if it runs in separate interpreter.
      Typically want to ensure code can't access arbitrary DOM variables.
    Next step: allow sandboxed Javascript code to interact with certain objects.

  Overall workflow:
    Something checks/rewrites untrusted Javascript code.
      In some ways, similar to the static analysis tool we looked at on Monday.
      Usually done on the server, but could be implemented on the client too.
      Just a transformation algorithm on strings (containing JS code).
      As a baseline, rely on Javascript's capability-like guarantees:
        code cannot make up arbitrary references to objects.
    Then, this checked/rewritten code runs in the browser.
    In some cases, rewriter adds calls to runtime support routines.
      Expects those routines to be present in the browser when code runs.
      Or can just include the code for those routines in the rewritten blob.

  Let's consider various ways in which Javascript code could escape sandbox.

```
Global variable names: document.cookie.
  Solution 1: prohibit accesses to sensitive names [Jt, Js].
    Need to prohibit some sensitive identifiers:
      eval (run arbitrary code that wasn't inspected at analysis time).
      Function (function constructor, does eval).
      ...
    Workable, but non-sensitive names aren't protected.
    Multiple pieces of untrusted Javascript code not isolated from each other.
  Solution 2: rename all variable names, adding a unique prefix.
    Each unique prefix becomes a separate sandbox / protection domain.
    Also would rewrite things like eval, Function to be meaningless / safe.
  E.g., origin code was:
    alert(document.cookie);
  New code will be:
    alert(a12345_document.cookie);
  Also helps prevent access to variable names in an enclosing scope.
  What if attacker guesses the a12345_ prefix?
    Shouldn't matter: just need it to be unique, not secret.
  What if attacker has variables named a12345_foo?
    Everything will get an extra "a12345_" prefix, so double-prefixed.

Adding a prefix to all variables breaks "this".
  Javascript's semi-equivalent of Python's "self".
  What if we didn't add a prefix to "this"?
  If running code not bound to an object, "this" is the global scope object.
    Also known as "window".
    Can access variables from global scope as attributes of scope object.
    E.g., "this.document"/"window.document" is the "document" global object.
    As a result, adversary can use "this" to get access to entire DOM.
  How to prevent?
    Problem: unknown statically if function will run as bound to an object.
  Solution 1: prohibit "this" altogether.
    Might be reasonable for newly-written code.
    Less practical if we want to sandbox existing Javascript libraries.
  Solution 2: runtime instrumentation.
    E.g., FBJS replaces "this" with "$FBJS.ref(this)".
  What's going on:
    $ is a perfectly legitimate character in Javascript variable names.
    Not really special, but generally reserved for synthesized code.
    $FBJS is a global object created by the FBJS library.
    $FBJS.ref() is a function that the static analysis tool knows about.
    $FBJS.ref() is roughly:
      function ref(x) {
        var globalscope = this;
        // Here, "this" will be the global scope because ref()
        // will not be bound to any object.
        if (x==globalscope) { return null; } else { return x; }
      }
    [ See NOGLOBAL() in section 4.3 for more precision. ]

Why are scope objects problematic?
  Static rewriting adds prefix to variable names, but not attribute names.
    Some attribute names are special and shouldn't be modified.
  But scope object allows accessing variable as an attribute.
    See also the "this" = "window" = global scope problem above.
  So code might break (variable has seemingly two different names).
    Attributes in a scope object get renamed: they're variables.
    Attributes in a non-scoped object don't get renamed.
  More likely: can access non-rewritten variables, which might
    belong to a different sandbox (if using multiple sandboxes).
  Worse yet, may leverage this inconsistency to escape sandbox.
    Depends on assumptions being made about scope objects.
```

```
How else could an adversary mix scope objects and regular objects?
  The "with" statement uses a given object as a scope (almost).
  E.g.:
    a = { x: 12, y: 23 };
    with (a) { x = 13; };    // now a.x=13
  Typically prohibit "with" to keep scope & non-scope objects separate.

Other ways of getting reference to a scope object?
  Section 3.2 suggests a few, but they don't work in Firefox/Chrome anymore.
  So problem disappeared since they wrote the paper.

Why did they have to prohibit sort, concat, etc?
  Array's sort/concat/... methods operates on "this" -- the object that
    the method was bound to, and returns it.
  Javascript allows taking a method (e.g., Array's sort) and binding
    to another object.
  Or it's possible to invoke a method without any binding at all,
    in which case "this" refers to the global scope.
  But recent Javascript interpreters don't seem to do this anymore
    (Firefox, Chrome, ..).

Shared state.
  Javascript uses prototypes to implement its version of objects.
  Prototypes are just another object, and are mutable.
  Even built-in objects like String have mutable prototypes.
  E.g.:
    x=[3,5,2]
    ""+x -> "3,5,2"
    Array.prototype.toString = function(){return "zz"}
    ""+x -> "zz"
  If malicious code gets access to the prototype object, can
    change behavior of objects used by trusted code in rest of page.
  Important to avoid giving access to shared mutable state.
  Thus, prohibit certain attributes: "prototype", etc.

Some more attributes of built-in objects can also be dangerous.
  E.g., all objects have a constructor attribute.
  For function objects, the function constructor turns strings to code (eval).
    new Function("return 2+3")() -> 5
  Can also access constructor without invoking the name "Function":
    var f = function() { return 3; };
    f() -> 3
    f.constructor("return 2+3;")() -> 5
  Attributes can be accessed in two ways: either dot-separated or brackets.
    f.attr
    f['attr']
  Common plan: filter out dangerous attributes: constructor, __parent__, etc.
    Paper has a detailed list.

What if we can't decide array indexing statically?  E.g., a[b].
  Recall, the PHP static analysis tool would just call it a[bottom].
  Can't do this here, because we need to be sure we catch everything.
  Solution 1: statically insert a call to a special function: a[$FBJS.idx(b)].
  At runtime, the $FBJS.idx() function will check if b's value is allowed.
    If it's allowed, return b.
    Otherwise, throw an exception.
  Static analyzer knows about $FBJS.idx() and assumes it works correctly.
  Slightly problematic: evaluation order [see 4.1].

How to implement trusted runtime functions?
  E.g., $FBJS.idx(), or some other trusted function accessible to sandbox?
  Consider the following implementation:
    function idx(x) {
          if (x == 'constructor') { return '__unknown__'; }
          return x;
        }
        a[idx(b)];

      Problem: there are implicit calls to toString(), effectively:
        if (x.toString() == 'constructor') { ... }
        a[idx(b).toString()];

      Can circumvent with:
        b = { toString: function() { if (count==0) { return 'constructor'; }
                                     count--; return 'nice'; } };
        c = 1;
        Then a[idx(b)] gives us a['constructor']

      What if we call toString() ourselves inside idx()?
        Not good enough: toString() could return another dynamic object.

      Correct fix: use the built-in String constructor.
        Need to save a reference to the built-in String constructor for idx().
        See section 4.1 for more details.

    Is it reasonable to give untrusted code access to a DOM element object?
      Could assign e.innerHTML = "<script>...</script>".
      Could traverse up the DOM tree using e.parentNode.
      Less error-prone to mediate access using a specialized API.
      Watch out for implicit operations when dealing with JS objects (IDX).

    Is it reasonable to give an object to an untrusted library?
      E.g., what if I want to import an untrusted sum() function.
      Can I give it an arbitrary object to sum up?
      Risk: untrusted code could modify any attributes on the object.
      Might not be able to modify prototype (if we prohibit that attribute).
      But could easily assign, say, the toString attribute on object itself.

    How robust is this plan?
      Turns out to be pretty fragile due to inconsistent implementations.
      Javascript interpreters don't always correctly implement ECMAscript.
      E.g., unable to get the examples from section 3.2 to work in
        either Firefox or Chrome anymore.
      Sandboxing depends intimately on understanding Javascript engine.
      Hard to do this reliably when the JS engine changes underneath.
      Proofs aren't very meaningful under non-compliant ECMAscript engines.
      Interesting final project: write something similar for Python?

    +: Fine-grained isolation.
    +: Potential compatibility with existing Javascript code.
    -: Fragile.
    -: Crossing between trusted & untrusted code requires careful analysis.
       E.g., exposing DOM objects, calling functions in either direction, etc.
       (Some examples came up with idx(), but not fully analyzed in this paper.)

  Approach 3: run the Javascript code in another origin, using an <iframe>.
    How to generate a separate origin?
      Could create random subdomains (e.g., randomstring.google.com).
      But that origin isn't quite isolated: can write to google.com cookies.
      New HTML5 feature: <iframe sandbox="allow-scripts">.
      Creates a new "synthetic" origin for the iframe.
      Potentially useful use case: <iframe sandbox=""> does not allow JS at all.
      Might be good for displaying untrusted documents that may have Javascript.
      Possible risk: if attacker guesses iframe page URL, can load w/o sandbox
        and run JS code on that page with access to your standard origin!
    +: Enforced by the browser's same-origin policy: perhaps less fragile.
  -: Does not allow shared state / interactions between origins.
  -: Limits display use to iframe's rectangle.

Approach 4: interaction between frames using server communication.
  Play tricks like <SCRIPT SRC="http://server.com/msg?params">.
  Use cross-origin resource sharing (CORS) to allow cross-origin XHR.
  -: Requires round-trip to the server, higher latency.

Approach 5: allow some interaction between frames on the client.
  Javascript provides a postMessage() API.
  Requires sender to have a reference on the recipient frame.
    Typically done by having one frame use an <iframe> to load another frame.
    Parent gets a handle on the child frame.
    In the child frame, 'parent' refers to the containing frame.
  Given a frame/window w, send a message using:
    w.postMessage(m, origin);
  Sends message m as long as w is in the specified origin (string).
    Why worry about the recipient's origin?
    Potential problem: adversary might navigate frame/window w!
  HTML5 allows many kinds of data structures to be passed as a message.
    Structured clone algorithm.
  Receiving frame must register to receive messages:
    window.addEventListener('message', processMsg, false);
    function processMsg(event) {
      // check event.origin for source of message
      ...
    }
  +: Strict isolation, only need to inspect messages being sent over.
  -: Requires RPC wrappers for everything.
  -: Hard to share objects (including DOM objects).
  -: Still limits display to iframe's rectangle, big deal for ads.
```

6.858 Fall 2012 Lab 2: Privilege separation

All parts due: Friday, October 5, 2012 (5:00pm)

Reprint of just part 2

Part 2: The `zoobar` Web Site

In the rest of this lab assignment, you will further secure the `zoobar` web site using privilege separation.

Well found = opps forgot

In the previous exercise, we fixed the bugs in the transfer code; now we would like to make sure we can deal with any future such bugs that come up. To do so, we want to make sure that we have a reliable log of all zoobar transfers that happened in the system. The current design stores the transfer log in the `transfer` SQL table, stored in `zoobar/db/transfer/transfer.db`. This table is accessible to all python code in the `zoobar` site, which means that an attacker might be able to change the history so that we will never find out about his or her attack.

We will try to make the transfer log more reliable by performing the logging operations in a separate process, running as a different user from the rest of the `zoobar` code. This user ID will only run logging code, which will insert new log entries into the `transfer` table. By setting permissions on the `zoobar/db/transfer` directory accordingly, we will ensure that only the logging code (which will hopefully be trustworthy) will be able to modify log entries, but any other python code will be able to read the log.

To break off some python code into a separate process, running as a separate user ID, we have provided you with some helper tools. The `zooksvc` service creates a Unix domain socket, and when someone connects to this socket, it will launch an arbitrary program. We have created a simple echo service using this tool. Look at how `zook.conf` spawns this echo service, the source code for the echo service tool in `zoobar/svc-echo.py`, and the sample client of this service in `zoobar/demo-client.py`. The client uses a simple library for connecting to Unix domain socket services, in `zoobar/unixclient.py`. Note that the client is meant to be invoked from the command line, rather than being executed as a CGI script via HTTP.

You may find `zoobar/demo-client.py` helpful in debugging any new Unix domain socket services you create.

Exercise 4. Create a new service to perform transfer logging as a separate user ID. You will need to create a new service along the lines of `svc-echo.py`; modify `zook.conf` to start it appropriately (under a different UID); modify the permissions on the `transfer` database directory such that only this new service can modify it; and modify the `transfer.py` code to invoke this service to log transactions, instead of logging transactions directly.

Make all of your changes in the `lab` directory rather than in `/jail`. In particular, if you need to set certain permissions on files or directories, or install additional files or directories in `/jail`, do so in the `chroot-setup.sh` script.

Note: be careful when picking a format for messages in your service. What if someone tries to passes spaces or a newline as an argument? (Hint: use some existing encoding like JSON. But don't use Python's pickle module.)

Run `sudo make check` to verify that your privilege-separated transfer service passes our tests.

Now, you will break up the `zoobar` code into two additional protection domains. First, we want to make sure that only the transfer code is actually able to modify the zoobar balances of different users, so that a vulnerability in the rest of the python code will not be able to directly modify the number of zoobars that some user has.

One complication in doing this rests in the fact that the zoobar balance information is stored in the same database table, `person`, that stores profile and login information that the rest of the code must be able to modify. To protect zoobar balances from being corrupted by the rest of the python code, you will need to create a new database table holding just the zoobar balances for each user, and remove the balance information from the `person` table.

Exercise 5. In preparation for privilege-separating the transfer code, split the `zoobars` field from the `person` table into a new `zoobars` table stored in the database file `zoobar/db/zoobars/zoobars.db`, and remove the `zoobars` column from the `person` table. Change the rest of the python code to access the correct table when fetching zoobar balances. Don't forget to handle the case of account creation, when the new user needs to get an initial 10 zoobars.

Exercise 6. Create a new service to transfer zoobars from one user to another. Change the `transfer.py` code to invoke this service instead of modifying the zoobar balances directly. Set the permissions on the new balances table such that only the transfer code can modify it, and the rest of the python code can only read it. Don't forget to handle account creation, which needs to involve your new transfer service.

Finally, make sure that only the transfer code is able to invoke the logging service -- after all, no other python code should be able to generate log entries! You should be able to do this by using groups and group permissions on the directory containing the logging service socket. As before, make sure all of your changes are reflected in the `chroot-setup.sh` script, and not only in the `/jail` directory in your VM.

Now our web application should be more secure, because compromises of most of the python code will not allow the attacker to modify zoobar balances. Unfortunately, the attacker can still subvert the web site by modifying user passwords or HTTP cookies in the `person` database table. For the final part of this assignment, you will move the authentication and cookie-verification code into a separate service that runs under a distinct user ID, to prevent such attacks.

Exercise 7. Split the authentication information (`password`, `salt`, and `token` fields) into an `auth` table that is separate from the original `person` table. Store this table in the database file zoobar/db/auth/auth.db. After you do this, the only remaining fields in the person table should be the username and the profile.

Create a new service that implements user login and cookie verification using this table. This service should implement three functions, which correspond to existing functions implemented in auth.py that you will need to replace. First, check the username and password for login, returning an HTTP cookie token if the password is correct. Second, verify whether a token is correct, returning true or false. Third, register a new user, again returning true or false depending on success.

Make sure that the auth table storing passwords and tokens is only readable by your new authentication service.

Now that the attacker cannot obtain anyone's passwords or HTTP cookies from the database, there is one last problem to fix.

Exercise 8. In the current design, the attacker can still invoke the transfer service and ask for credits to be transferred between an arbitrary pair of users. Modify the transfer service protocol to require a valid token from the sender of credits, and modify the transfer service implementation to validate this token with the authentication service before approving the transfer.

Although `make check` does not include an explicit test for this exercise, you should be able to check whether this feature is working or not by manually connecting to your transfer service using `zoobar/demo-client.py`, and verifying that it is not possible to perform a transfer without supplying a valid token.

Deliverables

Again, explain in answers.txt any non-obvious changes you made to zookws and zoobar for each exercise. Feel free to include any comments about your solutions in the answers.txt file.

You are done! Run `make submit` to submit your answers to the the submission web site.

Acknowledgments

Thanks to Stanford's CS155 course staff for the initial zoobar web application code, which we extended in this lab assignment.

```
#!/usr/bin/python

from unixclient import call

resp = call("/jail/echosvc/sock", "hello")
print "Response = ", resp


```

So when run this (demo-client.py) on cmd. line

L which uses call() to send commands

Our svc-echo.py service which is running
will return the text.

```python
#!/usr/bin/python

import socket

def call(pn, req):
    sock = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
    sock.connect(pn)
    sock.send(req)
    sock.shutdown(socket.SHUT_WR)
    data = ""
    while True:
        buf = sock.recv(1024)
        if not buf:
            break
        data += buf
    sock.close()
    return data

```

```
#!/usr/bin/python

import sys

req = sys.stdin.read()
print "You said:", req

```

Unix domain socket

From Wikipedia, the free encyclopedia

A **Unix domain socket** or **IPC socket** (inter-process communication socket) is a data communications endpoint for exchanging data between processes executing within the same host operating system. While similar in functionality to named pipes, Unix domain sockets may be created as byte streams or as datagram sequences[*clarify*], while pipes are byte streams only. Processes using Unix domain sockets do not need to share a common ancestry. The API for Unix domain sockets is similar to that of an Internet socket, but it does not use an underlying network protocol for communication. The Unix domain socket facility is a standard component of POSIX operating systems.

Unix domain sockets use the file system as address name space. They are referenced by processes as inodes in the file system. This allows two processes to open the same socket in order to communicate. However, communication occurs entirely within the operating system kernel.

In addition to sending data, processes may send file descriptors across a Unix domain socket connection using the `sendmsg()` and `recvmsg()` system calls.

See also

- Raw socket
- Datagram socket
- Stream socket
- Network socket
- Berkeley sockets
- Pipeline (Unix)

External links

- `socketpair(2) (http://www.kernel.org/doc/man-pages/online/pages/man2/socketpair.2.html)` – Linux Programmer's Manual – System Calls
- `sendmsg(2) (http://www.kernel.org/doc/man-pages/online/pages/man2/sendmsg.2.html)` – Linux Programmer's Manual – System Calls
- `recvmsg(2) (http://www.kernel.org/doc/man-pages/online/pages/man2/recvmsg.2.html)` – Linux Programmer's Manual – System Calls
- `cmsg(3) (http://www.kernel.org/doc/man-pages/online/pages/man3/cmsg.3.html)` – Linux Programmer's Manual – Library Functions
- ucspi-unix (http://untroubled.org/ucspi-unix/) , UNIX-domain socket client-server command-line tools
- unix domain sockets guide (http://beej.us/guide/bgipc/output/html/multipage/unixsock.html)
- another unix domain sockets guide (http://www.thomasstover.com/uds.html)
- Unix sockets vs Internet sockets (http://lists.freebsd.org/pipermail/freebsd-performance/2005-February/001143.html)
- Unix Domain Sockets for Java (http://code.google.com/p/junixsocket/)

Retrieved from "http://en.wikipedia.org/w/index.php?title=Unix_domain_socket&oldid=514437531"
Categories: Network socket | Unix | Computer science stubs

6.858 Lab 2 Part 2

9/29

Opps forgot to read closly...

transfer.~~py~~ svr accessible by all files

- yeah all dynamic files
  how do you fix?

Want logging to be seperate user
so only it could write
but everyone can read

fix!

?how do atomicity?
locking?

Unless we were
supposed to find
other bugs

~~A~~ must make setup
after every change!

Oh did a locking lib

And change after
log entry - good
enough

Some helper tools

Zook svc creates Unix domain socket
when connect will launch an
arbitray program

? that does not sound good

Zook.conf spawns an echo service

- Which just reads + returns sys.stdin

Hopefully not a big interprocess comm mess here

②

demo-client.py   svc-echo.py

Unix-client.py

execute via command line

Exercise 4 Create a new service to perform transfer logging

Need to create new service like svc-echo.py

Modify zook.conf to launch the service

I don't see where this is going...

Only new service shall be able to touch the transfer db

Transfer.py should call the service instead

Ah I see

So only through these approved APIs

Do what extent do people do these things in practice?

③

Watch newline characters

- Do JSON, not pickle
  - they don't say why

Understand svcs [illegible] stuff

- see other sheet

Now start to implement

Gets copied to seperate db

So move actual adding to svc-transfer.py

And then have some text communication

SQL?

So leave read access

this is for _write_

Web: Appears execute _not_ needed

(4)

(x) Transfer Setup can't run
so not readable

Oh groups?
- yeah

Needs permission ?
- so setup db there?

Somehow must create directory/db w/o write permissions

So writable 1st
- init db
- then unwritable?

Where are dbs made?
Should be during make
But does not sem to be...

Are made in makeFile
But then in _init__ its setup again?

Removed → fixed!

Forgot the lock code
L just cut it

Balance wrong on transfer pg
Not on home pg though...

Oh always blank ? — well JS

Likly from before?
Yes

Note: only if logged in for some reason

6

Oh back to fixing trasfer

Call not defined

No [illegible] file error

What is conf?

Must copy file over

└ No how does it create that sock?

Oh in fork.conf

Jwang is extending his previous sock for echo svc

But how do you set up?

Oh [illegible] didn't put in title!

Still no

Posted to piazza@108

— Sock file made an hr ago

2 ¿Must it be a high user #?

Uses comma ✓

Oh echo is 755

this is ~~[illegible]~~ 644

So confused

Trying nc -U /jail/echosvc/sock

↳ just returns blank

Can't even enter anything

-U = unix domain sockets

Now error starting echo svc...

Permisson denied

18:49 - it updated

Oh it must be running...

Worked! — the demo

try changing the demo...

Oh still erroring

→ what is not true?

run actually + debug

Now not opening db — why?

Seems to be file permissions...

do 777 for now + fix

Still got connection refused....

when test w/ demo

— when switch to orig echo works

So keep debugging my file

Oh also need to copy zoodb?
Why not important?

It seemed to work now...

↳ at least doing something productive...

~~Wh~~ How does JSON loads work?
What does it return?

So what does db retrn on sucessfl add?

Ok demo working w/ transfer
Using hacked echo

Why did registration just fail?
Oh user exists

Why NoneType has no attribute zoohors error?

It's not printing as well

10

My echo works → even w/ db

But not the echo svc...

↳ ¿permissions?

¿Or we are at jail? as root?

Oh I think it worked!

But returned wrong pg...

Run sudo make check for 1st time

files seemed to have been moved!

11

Oh it did work

just got in headers

└ oh text errors!

✓ It works

- though never subtracts old ones
  woops..

and perms prob wrong

⊗ Not testing - fix later

works for me....

6.858 Lab More

10/2
10P

Last time it was kinda working

My Piazza ans said make sure py file executable

And run make before make check

802 So lets try moving it over 1st

it has the same permission as echo

and make didn't do anything

(I forgot everything w/ this lab)

Now permission denied on Zook svc 5

└ Hmm still working

Well w/ the echo service

②

And look up user info broken

[illegible]

just fix make check

Lgrading seems only broken there

So our get param

No it seems to work

In my ~~[illegible]~~ thing

Then why failing otherwise?

- ~~its just~~

Need the other steps

Or its the transfer db flow?

(Took like 30 min back up to speed)

✓ Fixed

12 13

Though no data on table

So balance + history seperate

Sender does not ↓

Oh sender ≠ recipient

Oh never add transfer
& commit

✓ Done
— works

Make check

✗ no. Zooksvc w/ access to transfer.db
That was our error earlier
but it works . . .

(4)

Converted other service to 61010

Now more than 1 svc error

61011 ?

Oh zooksvc needed ~~transfer~~ rely 1004 perm

just use 61011 now

⊗ Multiple can write to transfer.db

750 breaks it

/ \

61011 1002

¿Only see 1 user

But why can't write

⊗ Non owners can write

Not allowed: Group write
Other write

so 755

⑤ Then user is wrong

but shouldn't it be transfer svc?

Oh 6010 10 since echo

✓ Pass ex 4

still use echo

Oh well

Though we have a .sock file now

So let me convert

⊗ Connection reset by peer

Oh svc echo gets special ex permission in jail

Normal py files don't have

Don't know where converted

✓ Works

Ex 5

2 additional protection domains

transfer code can only modify zoobar
bal of diff users
so take it out of the persons.db

that is seperate exercise

So make new ~~folder~~ db
remove old col
update all code

This one in plural
despite others

---

do this 1st before services
actually they are tested together
& do both

exG → new service
for transfer code
also support inital setup

¿how to make sure only transfer code
able to invoke?

Get to that late...

Oh same transfer code!
log entry
and ± values!

¿How to do create?
Sep services?

svc-register.py

(this will prob be messiest)

⊗

Ah nice debugging!

⊗ No such file or directory register src/sed

It's there - but no such

What was the issue again?

L Grrr forgot...

My initial Piazza post

Forgot which worked

Oh must add to list

Still no

Permission denied

how give permission for multiple

Change grp

Got it

---

⑧

⊗ Now connection reset by peer

So debug that file

debugged

⊗ still connection reset by peer

~~Still~~ don't get it

it works w/ sample data

same file us transfer...

how did we fix earlier

Py permissions

Same...

⊗ permission denied

keep socks 777

⊗ Connection reset by peer...

(why can't I remember what happened!)

¿load SQL?

¿Why not printing?

Kathryn

¿is there a way to get it to print?

Demo-register.py!

(need to do other before I forget!

Sure connection reset error...

echo works

Ahh and gives db setup error

Oh I moved person over to that...

Move person back to normal

✓ works to register

(11)

Transfer still works

But read - fix
on user pg

Wait -> do a ~~m~~ make check

(this is echo svc again)

Do need to fix user pg

Hmm lookup not working

Find ~~function lang~~ template lang

¿How to query new db?

Why is it returning none?

Jinja: Not committing new user

Must add and commit **both**

(does the login look different?)

└ Think I am imagining

(12)

¿how test db added?

Ah -> read only

Did 777 -> works for now

index.py

This is such a mess!

✓ update

balance.py

✓ updated

¿table does not get reset on reload?

User.py

Same

Why does ~~zoo~~ zoobars.db not clear?

✓ All zoobars seem right

(I didn't see what half done went earlier

Now check

⊗ non owners can write to zoobars.db

This is group or owner

So same permission for both reg + transfer

Are we still using echo svc?

⊗ Not registering...

Moving to register service not working

revert

(14)

¿Why are Toolbars not updated

Permission error

need save for transfer + req

which mean need to fix socket

Fix in OH

TA Advice

10/3
10A

Connection reset by peer

↳ usually threw an exception

OH

10/3
5p

Ok where was I?

Emails

remove local zoobars/db before setting up

add the rm to the sh file

Sock creation is prob because other file threw an exception

So try running it

* Must be running ./zookld for demo!

⊗ Username not unique
bt changed make to delete!

Ok think fixed

Yeah → gets deleted
(try to understand!)

⊗ Socket Connection refused !!!

②

Sock is 61011

svc-register.py is 0:0 775

register svc is 61011

db is 755 61011:1002

NC ~~doesn't~~ produces an error
on other screen!.

Oh when it trys to execute can't
get file or directory
look in zook.conf

its the same!

Files match other services!

I don't get it!

TAs solution used same svc
is that issue
Should not be...

③

TA: prob can't have multiple services w/ same user id

I don't see how that makes sense...
~~[illegible]~~ unless user ids matter for ~~[illegible]~~ domain sockets

TA: Could do same grep - but not w/ the check code

So just transfer svc w/ other params

So now ✓ ~~daemon~~ works when direct
✓ demo works

Real world check
Looks good ✓

Check ⊗ More than 1 ~~r~~ running process has access
remove from zook.conf

④

⊗ transfer log service has unresticted access

which is su httpd -c ls -l

TA: Socket should not be 777

~~Normal~~ ~~all~~ Dynamic + Transfer service rwx

Others not

⊗ person table still has zoobars col

Remove it

Still seems to work

~~No trans~~

Display worked

Transfer failed ⊗

transfer.py : 53

Never updated

✓ Fixed

✓ Pass ex 6!

6.858

10/3
10P

Finish Lab 2

Last line -> did we prevent other py issuing log entries

well any dynamic .py code

So not much isolation - well from other scripts

though if logging service is db

only svc-transfer can touch

Oh well it passes

Prob could not isolate anymore

except if diff script
move or

Exercise 7

Attacker can subvert app by editing

~~a~~ person.db table

So split authentication into into sep table

7

New service

1. login
2. Cookie verification
3. register new users

Same template

Hopefully no problems

(But this part of lab is pointless
just more practice / debugging)

Problem

Permission denied on auth svc
when loader

Why was this again?

Make sure no errors lol!

Oh gid must be allowed for zooksvc

13

Now the Connection reset by peer

w/ the No Such file or directory

Same error as when set up register svc!

Which I fixed by raising another service

Darn!

But should check the string 1st...

✓ registers

now make sure returns right value...

✓ Sock shows up

But still Connection reset by peer when run

Oh must change it to read stdin

No same error —grr!

Oh what user run as! → the demo?

That didn't seem to matter

Why exec No such file or directory?

Did I have this before? what did we do?

Added command line args

✓ Register

✓ Login w/ cookie

✓ Return cookie

So it works from command line

Just need to fix commands

Reads don't need changing?

So just stuck now on sockets

⑤

Now ⓧ BTB 32 Broken Pipe

first time I saw that!

But demo still Connection Reset by Peer

no exceptions

it runs directly

unless echoing is bad...

I'm going to OH...

Re-read all Piazza ④108

All about no such file

10/4
1:30A

Jwang person.db read by everyone
had 700 → do 755

Still no

Jwang: Make everything 777

The other services work!
w/ demo script....

Change demo_auth 755 ← 644
x No -same error

OM-ish

10/4
12:40P

if no argument is provided → errors

So cut that line

Or if len (argv) check that

No proper program

print 1st arg of execl.

Wrong file encoding

set-buffer-file-encoding : unix

in my demo file

Some windows error

OH-ish

10/8
12P

Where was I?

Just fixed the auth svc call

~~Was the~~ Demo works

Now try it for real

⊗ Registrain failed - actual error

it sent message

but ? didn't return ?

no cookie

no response

Oh never did

Simple to fix → but stupid mistakes!

Oh it retrned a cookie

but why not set?

Oh 2 responses - sloppy ...

②

Now registers -> but does not log in ...

Cookie appears set -> but no!

? or set twice

TA: ~~it~~ its there 2x

User already exists -> return cookie?

Super bad pratice!

✓ registe works

but does not redirect -> never did?

Does ~~303~~ found
302

? but browser redirects back?

TA: Failed to registe cookies

never did check cookie?

never did

still need person db

(3)

why did I delete?

Username + profile

Ok added checkcookie method

Why no debugging that its working?

it does not read a cookie...

is quote in front?

TA: lots of things unsuccessfully tried

10/5
1:40P

Though when I pulled them from FF it worked

Though diff error (stupid)

?Is it a browser thing

? Or unprintable char?

(at least debugging Py)

9

¿Oh never save token?

✓ saves token now

¿ not same tokens ...

same issue

TA: Call .stip() to remove extra characters!

✓ it logs in now!

So what was the original issue?

¿make check ever run now?

~~Oh~~ Oh the quote -> now fixed

✗ non owners can write to auth.db

Ok fix that

Jwang was wrong here

⑤

⊗ auth table directory has grp or world permission

↳ so no one should read at all

IRWXG

✓ Pass ex 7!

Ex 8

Anyone can invoke the transfer service

Oh I pointed this out earlier

Add token to call transfer

Have transfer verify

No test

↳ but it still must work

So token browser has

(6)

¿How can svc _transfer check and call the sock?

check for sender

TA: So a process can be in multiple groups

So set up a common group

⊗ transfer exception line 55

to view sqlite
"hd"

So demo didn't have the righ param

and didn't import the call

⊗ permission denied

that extra -gid was not there

✓ tokens match

but pg output still screwed up

(7)

Normally

HTTP/1.1 200 Ok

Content-Type:

Content-Length

✓ Worked

transfer service can't print anything even if pg never prints

✓ That should work | done ex 8

won't test don't match - think it works

---

answers.txt

any non obvious ans + comments

none → so submit

I got better at coding now...

8

One last more check

☺ Done

Submit

Lab 2 Grade

```
PASS App functionality
PASS Exercise 1
PASS Exercise 2
FAIL Exercise 3: transfer succeeded with eval
PASS Exercise 4
PASS Exercise 6
PASS Exercise 7

Functionality: 10/10

Exercise 1: 10/10

Exercise 2: 10/10

Exercise 3: 7/10
- eval(...) allows arbitrary code execution. Use int(...) instead.
- (There shouldn't be a race condition with the zoobars db. Database
  transactions will give you the needed consistency. Cross-database
  consistency is not guaranteed though.)
```

? what happened here?

```
Exercise 4: 10/10

Exercise 6: 14/20
- The log service from exercise 4 should be separate from
  transfer_svc. Then a compromised transfer service can't change
  existing entries (though it can make bogus entries.)
- Not all transfer logic moved to transfer service. Compromised
  service can still improperly change zoobars with negative amounts.

Exercise 7: 10/20
- Adding a 'token' RPC to the auth service defeats the purpose of
  authentication checks. If you can just query the auth service for
  anyone's token, there's no point in, e.g., requiring a token for
  transfers. Instead, add a 'check cookie' RPC that just returns yes/no.
- The 'cookie' RPC is similarly problematic.

Exercise 8: 10/10

Total: 81/100

# Graded tarball SHA-1: cc345bcd8316a1a2842440bc55a4f6362fc7a1f5
```

Read 10/10

ForceHTTPS: Protecting High-Security Web Sites from Network Attacks

why no date?

Collin Jackson
Stanford University
collinj@cs.stanford.edu

Adam Barth
Stanford University
abarth@cs.stanford.edu

ABSTRACT

As wireless networks proliferate, web browsers operate in an increasingly hostile network environment. The HTTPS protocol has the potential to protect web users from network attackers, but real-world deployments must cope with misconfigured servers, causing imperfect web sites and users to compromise browsing sessions inadvertently. ForceHTTPS is a simple browser security mechanism that web sites or users can use to opt in to stricter error processing, improving the security of HTTPS by preventing network attacks that leverage the browser's lax error processing. By augmenting the browser with a database of custom URL rewrite rules, ForceHTTPS allows sophisticated users to transparently retrofit security onto some insecure sites that support HTTPS. We provide a prototype implementation of ForceHTTPS as a Firefox browser extension.

So a list of sites that require SSL?

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*Unauthorized Access*; K.4.4 [**Computers and Society**]: Electronic Commerce—*Security*

General Terms

Design, Security, Human Factors

Keywords

HTTPS, eavesdropping, pharming, same-origin policy

Sounds good

1. INTRODUCTION

HTTPS is designed to be secure against both eavesdroppers and active network attackers. In practice, however, all modern web browsers are willing to compromise the security of sites that use HTTPS in order to be compatible with sites that deploy HTTPS incorrectly. For example, if an active attacker presents a self-signed certificate, web browsers permit the user to click through a warning message and access the site despite the error. This behavior compromises the confidentiality of the site's Secure cookies, which often store a second factor of authentication, and allows the attacker to hijack a legitimate user's session, potentially letting the attacker to transfer money out of the user's bank account or perform other misdeeds. Browsers accept broken certificates and allow embedding of insecure scripts for two reasons:

? should be in DNS?

- **Compatibility.** Many web sites have incorrectly configured certificates and embed insecure scripts. A browser that enforces strict error processing is incompatible with these sites and will lose users to a more permissive browser.
- **Unknown Intent.** Some site owners intentionally use self-signed certificates and host portions of their site over HTTP because these mechanisms provide protection from passive attackers and they believe the risk of an active attack is outweighed by the cost of implementing HTTPS fully.

are reasons why people use home rates

Although a security-conscious site owner, such as a bank, might aim to implement a high-security site, he or she currently has no mechanism for communicating this intent to the browser. Other site owners that are less security-conscious, desiring protection only from passive network attackers, implement low-security sites by deploying certificates that are self-signed or have incorrect common names. The browser has no mechanism for differentiating these two kinds of sites and cannot distinguish between a legitimate misconfiguration in a low-security site and an attack on a high-security site. Without guidance, a browser does not have the context to make an useful risk-management decision about whether to trade off security for compatibility on a particular site.

1.1 Our Proposal

We propose ForceHTTPS, a simple mechanism that security-conscious sites can use to opt in to stricter error processing by the browser, essentially giving the browser guidance to be more secure. By setting a ForceHTTPS cookie, a site owner asks the browser to treat HTTPS errors as attacks, not as simple configuration mistakes. Specifically, enabling ForceHTTPS causes the brower to modify its behavior as follows:

1. Non-HTTPS connections to the site are redirected to HTTPS, preventing contact to the site without TLS.
2. All TLS errors, including self-signed certificates and common-name mismatches, terminate the TLS session.
3. Attempts to embed insecure (non-HTTPS) content into the site fail with network errors.

This stricter error handling has several benefits, including protecting the URL parameters, fragments, and Secure cookies from network attackers and users who click through security warnings. ForceHTTPS blocks participating sites from

? Cookie

Or DNS entry?

WWW 2008, April 21–25, 2008, Beijing, China.
ACM 978-1-60558-085-2/08/04.

(bad description of secure

embedding insecure content, such as scripts, cascading style sheets, and SWF movies, in order to secure the user's session with buggy sites that would otherwise allow an active network attacker to steal the user's password and second factor of authentication by silently replacing SWF movie embedded in the login page. By enabling ForceHTTPS, a site protects itself from careless mistakes by its own web developers. ForceHTTPS also offers a "developer mode" that explains these errors so that the site's web developer can find and fix vulnerabilities.

Used in concert with a phishing defense, such as Bank of America's SiteKey [1], ForceHTTPS lets a site protect itself from pharming. Previously proposed anti-pharming defenses [6, 20, 15] are difficult to implement and face major challenges to deployment. By contrast, ForceHTTPS is easy to implement because browsers already detect the errors sites wish to block and easy to deploy because sites need only set a single cookie. To demonstrate the feasibility of our approach, we provide a prototype of ForceHTTPS as a Firefox browser extension [12].

pharming = redirect to other site w/ hosts file

1.2 Power Users

ForceHTTPS also enables "power users" to upgrade the security of sites that implement HTTPS insecurely by setting a ForceHTTPS cookie on the site's behalf. This approach follows a recent trend in which sophisticated users have taken web security into their own hands. The NoScript [18] browser extension enables users to fix cross-site scripting vulnerabilities in sites they visit by disabling or limiting the capabilities of scripts on that site, albeit at the cost of functionality. Other client side tools for mitigating web site vulnerabilities include Noxes [16] and NoMoXSS [28]. The GMailSecure user script (which has had over 25,000 downloads) enables users to force secure connections to Gmail, mitigating eavesdropping attacks without any reduction in functionality.

In fact, this paper arose largely out of a desire by the authors to secure their Gmail sessions while using the wireless networks at security conferences after witnessing an alarmingly effective attack demonstration at Black Hat 2007 [10]. Securing Gmail without Google's cooperation is challenging because Gmail's session identifier is stored in an insecure cookie that is transmitted whenever a user visits any other Google property. By setting the ForceHTTPS cookie, a Gmail user upgrades the session cookie to a Secure cookie that is protected from both eavesdropping and active attackers.

what if you just reset cookie?

1.3 Organization

The rest of this paper is organized as follows. In Section 2 we describe the threats that ForceHTTPS is designed to protect against. In Section 3 we survey existing techniques that attempt to defend against these threats. In Section 4 we provide a specification of our proposal. In Section 5 we discuss design decisions and implementation details. We conclude in Section 6.

2. THREAT MODEL

2.1 Threats Addressed

ForceHTTPS is concerned with three threats: passive network attackers, active network attackers, and imperfect web developers.

- **Passive Network Attackers.** When a user browses the web on a wireless network, a nearby attacker can eavesdrop on unencrypted connections, such as HTTP requests. Such a passive network attacker can steal session identifiers and hijack the user's session. These eavesdropping attacks can be performed easily using wireless sniffing toolkits [29, 10]. Some sites, such as Gmail, permit access over HTTPS, leading a user to believe that accessing such a service over HTTPS protects them from an passive network attacker. Unfortunately, this is often not the case as session identifiers are typically stored in insecure cookies to permit interoperability with HTTP versions of the service. For example, the session identifier for Gmail is usually stored in a non-`Secure` cookie, permitting an attacker to hijack the user's Gmail session if the user makes a single HTTP request to Gmail. Additionally, the subjects and snippets of the one hundred most recent email messages can be retrieved using the user's `.google.com` session cookie, which is sent in the clear during every Google search request.

- **Active Network Attackers.** A more determined attacker can mount an active attack, either by impersonating a user's DNS server or, in a wireless network, by spoofing network frames or offering a similarly-named "evil twin" access point. If the user is behind a wireless home router, the attacker can attempt to reconfigure the router using default passwords and other vulnerabilities [26, 27, 25]. Some sites, such as banks, rely on HTTPS to protect them from these active attackers. Unfortunately, browsers allow their users to opt-out of these protections in order to be compatible with sites that incorrectly deploy HTTPS. These sites wish to be protected from active network attackers even if users do not understand the security warnings provided by their browsers.

- **Honest but Imperfect Web Developers.** Large web sites are constructed by numerous developers, who occasionally make mistakes and are not all security experts. One simple mistake, such as embedding a cascading style sheet or a SWF movie over HTTP, can allow an active attacker to compromise the security of an HTTPS site completely.[1] Even if the site's developers carefully scrutinize their login page for mixed content, a single insecure embedding anywhere on the site compromises the security of their login page because the attacker can script (control) the login page by injecting script into the page with mixed content. Both the site's owner and the site's users could wish the site to be secure despite its developers making mistakes.

2.2 Threats Not Addressed

- **Phishing.** Phishing attacks [7] occur when an attacker solicits authentication credentials from the user by hosting a fake site located on a different domain than the real site, perhaps driving traffic to the fake

[1] Both cascading style sheets and SWF movies can script the embedding page, to the surprise of many web developers. Most browsers do not issue mixed content warnings when insecure SWF files are embedded.

?

site by sending a link in an email. Phishing attacks can be very effective because users find it difficult to distinguish the real site from a fake site [5]. ForceHTTPS is not a defense against phishing, but it complements many existing phishing defenses, such as SiteKey [1], the Yahoo! Sign-in Seal [30], and Chase's Activation Code [4], by instructing the browser to protect session integrity and long-lived authentication tokens.

- **Malware and Browser Vulnerabilities.** Because ForceHTTPS is implemented as a browser security mechanism, it relies on the trustworthiness of the user's system to protect the session. Malicious code executing on the user's system can compromise a browser session, regardless of whether ForceHTTPS is used.

3. RELATED WORK

Previously known defenses to the threats described in Section 2 are shown in Table 1 and summarized in this section.

3.1 User-Controlled Defenses

- **User-enforced HTTPS.** Many web sites serve the same content over both HTTP and HTTPS, taking care to use HTTPS on the login or credit card entry page and HTTP elsewhere. This protects the user's long-lived authentication credentials and financial details from being stolen by eavesdroppers while retaining the performance benefits of unencrypted HTTP traffic. Unfortunately, many such sites set an non-`Secure` cookie containing the user's session identifier. This cookie is sent in the clear over HTTP and can be used by an eavesdropper to hijack the user's session.

  Security-conscious users can mitigate this vulnerability by attempting to visit the site using HTTPS, to the exclusion of HTTP. For example, the user can diligently type HTTPS URLs into the address bar and check the status bar before clicking on links. Unfortunately, even a single insecure HTTP request by the web site can lead to a compromise of the session cookies. If the insecure request is the result of a redirect or button click, the user could be unaware of the request until their credentials have already been compromised.

  For example, Gmail serves its content to authenticated users both over HTTPS and HTTP. The login form, however, is served exclusively over HTTPS. Users that want to check sensitive mail using Gmail can access the Gmail site over HTTPS instead of HTTP. In fact, many users install GMailSecure [21] to automatically redirect them to HTTPS pages when using Gmail. Unfortunately, GMailSecure does not actually protect the session cookie on `mail.google.com` because it performs the redirect after the browser has already sent the HTTP request (which contains the cookie) in the clear.

- **Certificate Errors.** Incorrectly configured web servers can cause a number of HTTPS certificate errors:

  - **Common-Name Mismatch.** HTTPS requires that a server present a certificate whose common name matches the server's host name. Many web servers erroneously present certificates with incorrect common names.
  - **Self-Signed.** Many site owners wish to use HTTPS but are unable or unwilling to purchase certificates from certificate authorities. Instead, these owners deploy self-signed certificates that provide security against passive attackers.
  - **Expired.** Certificates are valid only for a limited time period. Many web servers present certificates that have either not yet become valid or whose validity period has expired.

  When it encounters a certificate error, the browser presents the user with a security warning dialog, giving the user the option to continue despite the error. Browsers permit users to override these security errors in order to be compatible with misconfigured servers. Unfortunately, the warnings have become commonplace, with approximately 63% of certificates causing errors [24]. Although the user is in control, many users do not understand these warnings and are trained to ignore them by the multitude of misconfigured sites [23]. ForceHTTPS lets sites force these certificate errors to be treated as fatal.

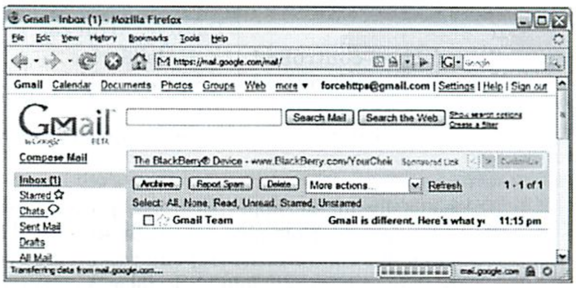

Gmail - Inbox (1) - Mozilla Firefox
File Edit View History Bookmarks Tools Help
https://mail.google.com/mail/
Gmail Calendar Documents Photos Groups Web more ▾ forcehttps@gmail.com | Settings | Help | Sign out
Gmail by Google BETA
Search Mail
Search the Web
Show search options
Create a filter
Compose Mail
The BlackBerry® Device - www.BlackBerry.com/YourChoi Sponsored Link
Inbox (1)
Starred
Chats
Sent Mail
Drafts
All Mail
Archive
Report Spam
Delete
More actions...
Refresh
1 - 1 of 1
Select: All, None, Read, Unread, Starred, Unstarred
Gmail Team
Gmail is different. Here's what y
11:15 pm
Transferring data from mail.google.com...
mail.google.com

Figure 1: This account has only ever been accessed over HTTPS, but the confidentiality of this user's email has already been compromised because Firefox leaked the user's cookie in an automatic request for anti-phishing data from Google.

- **Extended Validation.** Many certificate authorities issue "extended validation" (EV) certificates that require more extensive investigation by the certificate authority before being issued [9]. Like certificate warnings, EV certificates are used to present information about the connection security to the user. For example, Internet Explorer 7 and Firefox 3 highlight the site's identity in green if the site supplies a valid EV certificate. Extended validation certificates have no effect on the browser's defenses against network attackers. A site that uses EV can still be contacted via HTTP and mix insecure content into secure pages. Moreover, the user is still able to accept a broken certificate for the host, putting primary control over enforcement in the hands of the user. ForceHTTPS allows the site to make a security commitment to the browser, rather than to the user.

- **Firefox 3.** Firefox 3 contains a new user interface for dealing with certificate errors. Early versions of this interface required ten clicks to accept certificate errors and asked the user to type the domain name

	Threat Model		
	Passive Attacker	**Active Attacker**	**Imperfect Developer**
User-controlled	GMailSecure	Certificate warnings	Mixed content warnings
Site-controlled	Secure cookies	Locked same-origin policy, HTTPSSR	Content restrictions

Table 1: Current attempts to defend against the threats that ForceHTTPS addresses.

manually in the hopes that this process would discourage users from giving up their security. This proposal was controversial [11] and was eventually scaled back to require only four clicks [3] as a compromise for site owners that use HTTPS with self-signed certificates. ForceHTTPS avoids compromising security for usability by affecting only those sites that are security-conscious.

- **Mixed Content Warnings.** Many sites serve the same content over both HTTP and HTTPS. If the developer expected some of the content to be served over HTTP only, the developer is likely to embed scripts using absolute paths containing the `http` scheme:

```
<script src="http://a.com/foo.js"></script>
```

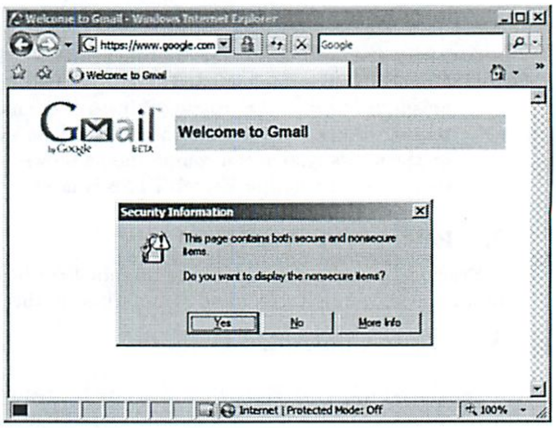
Unfortunately, this compromises the security of HTTPS on the entire site because an active attacker can navigate the user's browser to the broken page over HTTPS, replace the insecure script with his own, and invade the security context of the secure site. These mistakes can easily be corrected by using scheme-relative paths [8]:

```
<script src="//a.com/foo.js"></script>
```

These paths cause the browser to load the script over HTTP when the page is viewed over HTTP and over HTTPS when the page is viewed over HTTPS. Using this technique, a site can benefit from caching and increased performance when the page is viewed over HTTP but retain security when the page is viewed over HTTPS. Unfortunately, many web developers are unaware of scheme-relative paths and often accidentally embed insecure scripts into secure pages. Browsers warn the user about these insecure embeddings in different ways:

  - **Internet Explorer** displays a "mixed content" dialog that asks the user's permission before continuing. Insecure SWF movies and Java applets are loaded automatically without any warnings.
  - **Firefox** automatically accepts the mixed content, but draws a red slash over the browser's lock icon. Insecure images, SWF movies, and Java applets do not trigger the slash.
  - **Opera** automatically accepts the mixed content, but replaces the lock icon with a question mark.
  - **Safari** does not attempt to detect mixed content.

As with certificate warnings, many users do not understand mixed content warnings, and some browsers do not even give users the option of remaining secure. Users have been trained to ignore these warnings because many HTTPS pages, such as the Gmail login page shown in Figure 2, embed mixed content. ForceHTTPS lets security-conscious sites block unwanted mixed content inadvertently introduced by their imperfect developers.



This page contains both secure and nonsecure items.

Do you want to display the nonsecure items?

Figure 2: Users have been trained to click through mixed content warnings at sites such as Gmail.

Ubiquitous error msg

3.2 Site-Controlled Defenses

- **Secure Cookies.** A security-conscious site can mark a cookie as Secure, instructing the browser to refrain from transmitting the cookie over an insecure connection. To use these cookies, the site must ensure that all authenticated web traffic occurs over HTTPS. Many sites, including those that have deployed anti-phishing defenses such as SiteKey, also use a long-lived `Secure` cookie to store a second factor of authentication.

  - **Passive Attackers.** `Secure` cookies defend well against passive eavesdroppers. We recommend that sites use `Secure` cookies as they prevent a passive attacker from learning the confidential information they store.
  - **Active Attackers.** Unfortunately, active attackers can use invalid certificates to steal `Secure` cookies if users click through certificate warning dialog boxes.

ForceHTTPS expands the usefulness of `Secure` cookies to defend against active attackers by recording the web site's intent to use a correct HTTPS certificate. When the attacker presents an invalid certificate for the site, the browser terminates the connection and does not reveal the site's `Secure` cookies.

- **Locked Same-Origin.** Web Server Key Enabled Cookies [20] proposes restricting access to cookies based on the public key of the server. The goal of this policy is to prevent a pharming attacker from accessing HTTPS cookies set by the victim server. Karlof et. al. [15] extend this work to defend against dynamic pharming through the use of two *locked same-origin policies* for browsers. These policies augment the browser's security policy to isolate web pages based on the security of the connection from which they were loaded. Unfortunately, both locked same-origin policies face major deployment challenges.

  - **Weak.** The weak locked same-origin policy isolates pages loaded over broken HTTPS connections from those loaded over unbroken connections. To be secure against an active attacker, a site must not embed any scripts, cascading style sheets, applets, or SWF movies (instead, the site must inline all scripts and style sheets) [15], but this requires virtually all web sites to implement major changes in order to meet this condition.
  - **Strong.** The strong locked same-origin policy segregates two pages if they where loaded over HTTPS connections with different public keys. To enable the strong policy, a site must deploy a `pk.txt` file that specifies the public keys with which it intends to interact. This file is difficult to deploy correctly and must be maintained as servers refresh their keys, likely resulting in a similar misconfiguration rate to that of deploying certificates for HTTPS.

  ForceHTTPS also isolates broken and unbroken pages by allowing security-conscious sites to forbid the browser from loading broken sites, but ForceHTTP is easier for sites to deploy: the site can opt in to ForceHTTPS by simply setting a cookie.

- **Content Restrictions.** Using content restrictions, web servers can transmit metadata to browsers instructing them to impose certain restrictions on the web site's content, such as which scripts are allowed to run. Content restrictions can limit the damage caused by a cross-site scripting attack in which the developer incorrectly sanitizes malicious input. Content restrictions can be communicated in HTTP headers or `<meta>` tags [19]. Other proposals include whitelists written in JavaScript, or using a special `noexecute` property of DOM nodes [13]. ForceHTTPS is another set of content restrictions, but instead of defending against a web developer who inadvertently exposes the session to cross-site scripting attacks, it defends against a web developer who inadvertently exposes the site to network attacks via mixed content.

4. SPECIFICATION

ForceHTTPS can be enabled in two ways:

- **Site.** A security-conscious site can enable ForceHTTPS by setting a cookie with the name `ForceHTTPS` using a `Set-Cookie` header in an error-free HTTPS response. The browser will enable ForceHTTPS for that site as long as the cookie has not expired. The `domain` and `path` attributes of the cookie are ignored.
- **User.** A security-conscious user can enable ForceHTTPS for a host through the browser user interface. The browser gives them the option of configuring custom HTTP-to-HTTPS redirection rules and non-`Secure`-to-`Secure` cookie upgrades for that domain.

ForceHTTPS can be disabled only by an error-free HTTPS response or by the browser's user interface.

When ForceHTTPS is enabled for a host, the browser modifies its behavior as follows:

- Attempts to connect over a non-HTTPS protocol are redirected to HTTPS.
- TLS errors during connections are treated as fatal.
- Attempts to embed insecure content in pages fail.

These rules prevent an active attacker from injecting script into the host's security origin.

5. DISCUSSION

This section contains a discussion of design decisions, error handling scenarios, limitations, and alternate policy advertisement mechanisms.

5.1 Design Decisions

Although the ForceHTTPS mechanism is simple, a number of subtle decisions were made during its design.

- **Redirecting URLs.** When ForceHTTPS is enabled for a host, the browser redirects HTTP requests to that host to HTTPS. For example, if the user types `www.paypal.com` in the location bar, the browser connects to `https://www.paypal.com/` instead, preventing a network attacker from intercepting the HTTP request and redirecting the user to a phishing web site. Additionally, this browser-side redirection transparently corrects a common mixed content scenario in which a site embeds active content from itself over HTTP. To retrofit security onto sites like Google that do not serve all of their content over HTTPS, ForceHTTPS lets power users configure custom rewrite rules.
- **State Exhaustion.** Because the browser has limited state, the browser's cookie eviction policy is critical to the security of ForceHTTPS. An attacker who is able to force the browser to evict the ForceHTTPS cookie is effectively able to "unforce" HTTPS. Moreover, if the browser evicts the ForceHTTPS cookie before other cookies for the same host, the attacker can potentially use the non-evicted cookies (which might store session tokens or second factors of authentication) as part of an attack. To prevent these state exhaustion attacks, the browser should reserve space for ForceHTTPS cookies and limit the rate at which it accepts new ForceHTTPS cookies. If the browser uses an rate-limiting scheme with exponential back-off, the browser can typically prevent an attacker from flooding its ForceHTTPS cookie store in a single session. A concerted attacker, however, can eventually overflow the state limit over many successive sessions. To prevent the other cookies from being stolen, the browser should evict all other cookies for a domain if it evicts the ForceHTTPS cookie.

? what is this browser ex
or just a cookie

- **Denial of Service.** The largest risk in deploying ForceHTTPS is that of denial of service. An attacker who can set a ForceHTTPS cookie for a victim host can prevent users from using that site if the site requires broken HTTPS to function properly. There are two restrictions on when a site can set a ForceHTTPS cookie to mitigate this issue:
  - The server must set the ForceHTTPS cookie during a non-broken HTTPS session. By establishing a non-broken HTTPS session, the host has demonstrated the ability to conduct secure HTTPS. If the browser permitted ForceHTTPS cookies to be set over HTTP, an active attacker could conduct denial of service beyond his ability to control the user's network.
  - The server must set the ForceHTTPS cookie using the `Set-Cookie` header, rather than using script to set the `document.cookie` property. If script were permitted to set ForceHTTPS cookie, a transient cross-site scripting vulnerability could result in a long-lasting denial of service.

  Even with these restrictions, a shared domain ForceHTTPS cookie could still be used for denial of service: A student hosting content on `https://www.stanford.edu/` could set a ForceHTTPS cookie for `.stanford.edu`, denying service to many Stanford web sites. To prevent this scenario, a ForceHTTPS cookie enables ForceHTTPS only for the host that sent the cookie.

- **Policy Expressiveness.** When a site enables ForceHTTPS, the browser makes several modifications to its behavior at once. Instead, the browser could respect finer-grained policies capable of expressing more specific behavior changes, for example allowing a site to require HTTPS without disavowing mixed content or certificate errors. However, exposing a more expressive policy interface increases the burden on site developers to select the appropriate policy and on browser developers to correctly implement each policy permutation. We reserve the value of the ForceHTTPS cookie for future enhancements to the mechanism.

5.2 Error Handling

Although it provides stricter error handling, ForceHTTPS must be prepared to handle misconfigured clients and servers. If ForceHTTPS simply were to provide a click-through error dialog box, the benefits of the mechanism would be lost. Many users consider clicking through security dialog boxes to be a routine task.

- **Wireless HotSpot.** The most common client error occurs when a user first connects their computer to a wireless hotspot. Before allowing access to the Internet, the hotspot typically redirects all network requests to its registration page. If the user attempts to navigate to an HTTPS site, the hotspot will be unable to present a valid certificate and the connection will generate a certificate error. In this situation, the two options offered by current browsers are both poor. The user can either abandon the request (and not join the network) or can accept the broken certificate, sending their secure cookies to the hotspot registration page.

  To better recover from this error condition, the browser could attempt to connect to a known HTTP page on the browser vendor's web site and compare its contents to a known value. If a redirect is encountered or the contents of the page do not match the expected value, the browser could ask the user if they would like to connect to the wireless network registration page (which consists of the redirected content). This technique permits the registration page to successfully redirect the user without compromising the user's cookies and without revealing any sensitive query parameters (as used by PHP sites that set `session.use_trans_sid` to true and `session.use_cookies` to false).

- **Embedded Content.** When ForceHTTPS is enabled for a host, the browser prevents pages on that host from embedding non-HTTPS content. The security of the site can still be compromised, however, if the site embeds content from an HTTPS connection that encountered a certificate error. For this reason, certificate errors are treated as fatal network errors during any dependent load on a ForceHTTPS page. For content that would appear in a frame, the broken content is replaced with a message indicating that the content could not be loaded securely.

- **Opting Out.** If a ForceHTTPS site persists in being misconfigured, the user can remove the ForceHTTPS cookie through the same user interface used to enable ForceHTTPS. This process requires several steps, i.e. not a single mouse click, and both clears the user's cookies and restarts the browser to prevent any existing browser state from being compromised. We expect that the rate of ForceHTTPS hosts misconfiguration will be significantly lower than the general HTTPS misconfiguration rate because the owners of the ForceHTTPS hosts have indicated (by enabling ForceHTTPS) that they take seriously the security of their sites and do not wish to allow users to connect over broken HTTPS connections. In contrast, users will need to become familiar with the browser's mechanism to bypass standard certificate errors in order to access many misconfigured sites.

5.3 Limitations

Although ForceHTTPS has numerous security benefits, it cannot prevent all attacks. In this section, we describe some vulnerabilities that ForceHTTPS does not address.

- **Attacks on Initialization.** If a user is unable to establish a secure connection to a server, then that server cannot set a ForceHTTPS cookie. An attacker who controls the user's network on *every* visit to a target site can prevent the ForceHTTPS cookie at that site from ever being set. Although the user will be exposed to a large number of warnings, ForceHTTPS will not yet be enabled and thus cannot force the user to make the correct security decision. However, if the user does ever connect to the site securely, the browser enforce security until the ForceHTTPS cookie expires.

- **Privacy.** Like any cookie, ForceHTTPS leaves a trace on the user's system for each ForceHTTPS site visited. Users who are concerned about privacy from web sites or from other users who use the same system often reject or frequently clear their cookies. By clearing cookies, these users can remove all evidence of the ForceHTTPS cookie. Although they lose ForceHTTPS protection their next visit, the user's decision to purge all browser state associated with the site will make it unlikely that the browser will have second factor authentication tokens for a future attacker to steal. (Note that the preconfigured ForceHTTPS cookies and rewrite rules are the same for each user and do not reveal the user's browsing behavior other than to identify them as a ForceHTTPS user.)

- **Developer Errors Other Than Mixed Content.** By enabling ForceHTTPS, the web developer opts in to more stringent error processing, but the developer still compromise the security of his or her site by making mistakes. We list a few common mistakes of this sort to remind the reader that ForceHTTPS (and more generally encryption) is not a panacea.
  - **Cross-Site Scripting (XSS).** ForceHTTPS provides no protection if the site contains a cross-site scripting vulnerability. Such a site is completely vulnerable to a web attacker.
  - **Cross-Site Request Forgery (CSRF).** Similarly, ForceHTTPS does not protect a site that contains a cross-site request forgery vulnerability [14]. CSRF vulnerabilities often give attackers the ability to issue commands from the user's browser.
  - **HTTP Response Splitting.** If the server does not properly sanitize carriage returns and other whitespace in input included in HTTP response headers, an attacker can inject headers (and potentially scripts) into HTTP responses. An HTTP response splitting vulnerability can often be used to manipulate ForceHTTPS cookies.
  - **document.domain.** A site that sets its domain to a value must trust all the hosts with that value as a suffix. These hosts can enter the site's security sandbox and script its pages.
- **Plug-ins.** Analysis of browser security features must take plug-ins into account because plug-ins such as Flash Player and Java are widely deployed and can often provide attackers an alternate route to circumventing a security mechanism. ForceHTTPS must ensure that browser network requests on behalf of plug-ins, which carry the user's cookies, enforce the ForceHTTPS restrictions. Furthermore, all cookie management by plug-ins must respect the ForceHTTPS policy. If the plug-in allows the site to make direct network requests using raw sockets, it cannot be forced to use HTTPS without breaking backwards compatibility. We consider it the web site's responsibility to provide appropriate encryption of the raw socket traffic if necessary; ForceHTTPS does not provide protection from the imperfect developer in this case.
- **Complexity of Rewrite Rules.** As we describe in Section 5.5, the rewrite rules required to enable ForceHTTPS at a legacy web site can range from very simple to impossible. A site could become vulnerable if rewrite rules are introduced that redirect sensitive information to an attacker. Rewrite rules can also break functionality at the web site, rendering certain pages inaccessible or issuing unauthorized transactions. If the web site changes significantly, or the site decides to change its support for HTTPS, the rewrite rules might need to be updated. We consider the installation and editing of rewrite rules to be a decision with serious security consequences, similar to installing a browser plug-in. The addition of new rewrite rules is a feature primarily for advanced users.

5.4 Other Policy Advertisement Mechanisms

Other mechanisms that could be used for advertising a ForceHTTPS policy include DNS records and XML files.

- **DNS.** In the HTTP Service Security Requirements (HTTPSSR) proposal [22], a site can indicate its desire for HTTPS by including an HTTPSSR record in DNS. The proposal relies on DNSSEC to prevent a network attacker from manipulating this record. Although the HTTPSSR proposal does not address mixed content, certificate error user interfaces, or cookie security, it could be extended to do so. The DNS policy advertisement mechanism has a number of advantages:

  1. The secure initialization step is not required. The browser can obtain the ForceHTTPS policy on the first visit to the site, even if the network is compromised.
  2. The browser is not required to maintain any persistent state associated for each host, preventing state exhaustion attacks.
  3. HTTP response splitting attacks do not allow an attacker to manipulate ForceHTTPS policies.

  Unfortunately, DNSSEC is not widely deployed. Without DNSSEC, sites can store their ForceHTTPS policies in DNS using the stateful, secure-initialization approach of ForceHTTPS cookies. To support this approach, HTTPSSR records would need to include an "expires" field. The Time-To-Live (TTL) supplied by DNS is not suitable for storing policy expiry because it provides a maximum, rather than a minimum, duration for the validity of the record.

- **XML.** Using the XML paradigm, a site can advertise its ForceHTTPS policy in an XML document hosted over HTTPS at a well-known location. This technique is used by Adobe Flash Player to determine if a server is willing to receive cross-domain URL requests. Adobe's `crossdomain.xml` policy file could be extended to advertise a `ForceHTTPS` policy:

```
<?xml version="1.0" ?>
<cross-domain-policy
    xmlns:f="http://www.forcehttps.com/">
  <allow-access-from
      domain="*.stanford.edu" />
  <f:forcehttps
      expires="Mon, 11 Feb 2009 23:39:27 GMT"/>
</cross-domain-policy>
```

The browser will enable ForceHTTPS for that site for the duration specified by the expires attribute of this element. This element can be included in existing crossdomain.xml files using a unique XML namespace for the element. This approach has the advantage that a site must already control the contents of its crossdomain.xml file in order to be secure against attacks using the Flash plug-in. Additionally, using XML to store policy information makes it possible to extend this policy advertisement mechanism to include future security policies.

5.5 Example Rewrite Rules

In creating our prototype implementation of ForceHTTPS, we developed rewrite rules for seven popular sites to understand the subtleties in deploying ForceHTTPS. To develop the rewrite rules, we installed the ForceHTTPS extension and enabled ForceHTTPS for each site we wanted to support. We then turned on client-side error logging and tried to log in and log out on each site. Using the error messages we identified HTTP content that could be served over HTTPS and used rewrite rules to transform those HTTP requests into HTTPS. The results are summarized below.

- **PayPal.** We did not need specialized rewrite rules for paypal.com, which serves all content on its main site over HTTPS. We also enabled ForceHTTPS for paypalobjects.com, where PayPal's static scripts and stylesheets are hosted. This precaution is necessary for Firefox 2, which prompts users to override certificate errors for embedded content, but is no longer necessary in Firefox 3, which blocks such content automatically.

- **American Express.** American Express uses SWF movies to load HTTP files to display advertisements, but the insecure files are served from a different domain (doubleclick.net) and cannot script the main American Express page.

- **Fidelity.** Fidelity uses SWF movies that load HTTP files to display stock quotes, but these requests do not require cookies, so no rewrite rules are necessary. Fidelity hosts a crossdomain.xml file that allows access from *.fidelity.com and *.fmr.com. Thus, to be protected from network attackers, Fidelity needs a ForceHTTPS cookie for both .fidelity.com and .fmr.com.

- **Bank of America.** Bank of America uses both HTTP and HTTPS on its main home page, and certain pages require cookies to be sent over HTTP. However, the login page and online banking are handled on subdomains, such as sitekey.bankofamerica.com. These subdomains use HTTPS exclusively, so we set ForceHTTPS cookies for the online banking subdomains.

- **Gmail.** Google's Gmail web site, mail.google.com, presents a challenge because the site sets a domain-wide .google.com cookie. We enabled ForceHTTPS for the entire Google site and wrote rewrite rules to redirect all Google pages to HTTPS except the search page (which cannot be accessed over HTTPS). Additionally, we rewrote a query parameter for the login page to indicate that we wished Google to mark its session cookies Secure. It is important to redirect all pages (except search) to HTTPS because Google's login page sometimes transmits sensitive authentication information in URL parameters. With ForceHTTPS enabled, search traffic at Google is not protected from eavesdropping, but no cookies are sent with this traffic, keeping the user's session identifier secure.

- **Chase.** Chase refuses to serve its home page over HTTPS. We chose to redirect http://www.chase.com/ to https://chaseonline.chase.com, allowing the user to log in securely, but preventing access to any news or special offers that appear only on the Chase home page. ForceHTTPS also automatically repairs mixed content on Chase's login page by redirecting an insecure SWF movie to HTTPS.

- **Yahoo! Mail.** We were unable to develop rewrite rules for the Yahoo! Mail site because Yahoo! Mail does not support HTTPS. We enabled ForceHTTPS for the Yahoo! login page, with the goal of protecting the user's password (rather than the session) from active attacks. Because the Yahoo! Sign-in Seal [30] is revealed by an insecure cookie, an active attacker could display the sign-in seal on an HTTP page without requiring the user to click through a security warning dialog. With ForceHTTPS installed, the attacker cannot display the Sign-in Seal, upgrading Yahoo!'s phishing defense to a pharming defense as well.

6. CONCLUSIONS AND FUTURE WORK

ForceHTTPS lets users and web sites to opt in to stricter error processing by the browser. For users, ForceHTTPS can fix vulnerabilities in web sites and enable sites that were not designed to be used over hostile networks to be browsed securely over such networks. For web sites, ForceHTTPS protects Secure cookies from active network attackers and remediates accidental embedding of insecure content.

Previous anti-pharming proposals required either overhauling DNS or the deployment of complex, digitally signed policy files encoding the frequently-changing trust relationships between domains. By contrast, ForceHTTPS merely requires setting a cookie, a procedure that many sites already handle with every new session.

ForceHTTPS is a useful mitigation for mixed content, but sites should strive to fix these bugs by removing insecure embeddings. Developers have trouble detecting mixed content because all the major browsers have significant bugs in their mixed content detection mechanisms. In future work, we plan to collaborate with web application vulnerability scanner vendors to build a mixed content scanner that spiders a web site and reports its mixed content vulnerabilities.

ForceHTTPS has already proven itself useful to its authors, who now check their email at security conferences without fear of eavesdropping and other network attacks. We look forward to extending this protection to other users.

Acknowledgements

We thank Michael Barrett, Dan Boneh, John C. Mitchell, Umesh Shankar, and Andy Steingruebl for their helpful suggestions and feedback. This work is supported by grants from the National Science Foundation and the US Department of Homeland Security.

7. REFERENCES

[1] Bank of America SiteKey. http://www.bankofamerica.com/privacy/sitekey/.

[2] A. Barth, C. Jackson, and J. C. Mitchell. Session swapping: Login cross-site request forgery, March 2008. Manuscript.

[3] M. Beltzner et al. Create preference which restores per-page ssl error override option for it professionals. https://bugzilla.mozilla.org/show_bug.cgi?id=399275.

[4] Chase. Increased security. http://www.chase.com/ccpmapp/shared/assets/page/occ_alert.

[5] R. Dhamija, J. D. Tygar, and M. Hearst. Why phishing works. In *Proceedings of the Conference on Human Factors in Computing Systems (CHI)*, 2006.

[6] DNS Security Extensions. http://www.dnssec.net/.

[7] E. W. Felten, D. Balfanz, D. Dean, and D. S. Wallach. Web Spoofing: An Internet Con Game. In *20th National Information Systems Security Conference*, October 1997.

[8] R. Fielding. Relative Uniform Resource Locators. IETF RFC 1808, June 1995.

[9] C. A. B. Forum. Extended validation certificate guidelines. http://cabforum.org/EV_Certificate_Guidelines.pdf.

[10] R. Graham. Sidejacking with Hamster, August 2007. http://erratasec.blogspot.com/2007/08/sidejacking-with-hamster_05.html.

[11] F. Hecker et al. Improve error reporting for invalid-certificate errors. https://bugzilla.mozilla.org/show_bug.cgi?id=327181.

[12] C. Jackson and A. Barth. ForceHTTPS Firefox extension, 2008. https://crypto.stanford.edu/forcehttps.

[13] T. Jim, N. Swamy, and M. Hicks. BEEP: Browser-enforced embedded policies. In *Proceedings of the 14th International World Wide Web Conference (WWW)*, 2007.

[14] N. Jovanovic, E. Kirda, and C. Kruegel. Preventing cross site request forgery attacks. In *Proceedings of the IEEE International Conference on Security and Privacy for Emerging Areas in Communication Networks (Securecomm)*, 2006.

[15] C. Karlof, U. Shankar, J. D. Tygar, and D. Wagner. Dynamic pharming attacks and locked same-origin policies for web browsers. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS 2007)*, November 2007.

[16] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: A client-side solution for mitigating cross site scripting attacks. In *Proceedings of the 21st ACM Symposium on Applied Computing (SAC)*, 2006.

[17] D. Kristol and L. Montulli. HTTP State Management Mechanism. IETF RFC 2109, February 1997.

[18] G. Maone. NoScript. http://noscript.net/.

[19] G. Markham. Content restrictions. http://www.gerv.net/security/content-restrictions/.

[20] C. Masone, K.-H. Baek, and S. Smith. Wske: Web server key enabled cookies. In *Proceedings of Usable Security 2007 (USEC '07).*

[21] M. Pilgrim. GMailSecure, 2005. http://userscripts.org/scripts/review/1404.

[22] S. E. Schechter. Storing HTTP security requirements in the domain name system, April 2007. http://lists.w3.org/Archives/Public/public-wsc-wg/2007Apr/att-0332/http-ssr.txt.

[23] S. E. Schechter, R. Dhamija, A. Ozment, and I. Fischer. The emperor's new security indicators. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy.*

[24] Security Space and E-Soft. Secure server survey, May 2007. http://www.securityspace.com/s_survey/sdata/200704/certca.html.

[25] S. Stamm, Z. Ramzan, and M. Jakobsson. Drive-by pharming. Technical Report 641, Indiana University Computer Science, Decenber 2006.

[26] A. Tsow. Phishing with consumer electronics – malicious home routers. In *Models of Trust for the Web Workshop at the 15th International World Wide Web Conference (WWW)*, 2006.

[27] A. Tsow, M. Jakobsson, L. Yang, and S. Wetzel. Warkitting: the drive-by subversion of wireless home routers. *Journal of Digital Forensic Practice*, 1(2), November 2006.

[28] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2007.

[29] Wireshark: What's on your network? http://www.wireshark.org/.

[30] Yahoo! Inc. What is a sign-in seal? http://security.yahoo.com/article.html?aid=2006102507.

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 10

Suppose that a web application developer wants to avoid the security pitfalls described in the ForceHTTPS paper. The developer uses HTTPS for the application's entire site, and marks all of the application's cookies as "Secure". If the developer makes no mistakes in doing so, are there still reasons to use ForceHTTPS? Explain why not, or provide examples of specific attacks that ForceHTTPS would prevent.

Yes,

- well assume no insecure reqs like w/ Gmail

QA - an attacker site - clone could not
and user still tricked

browsers will bend - like sending self
signed certs

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

Top // **6.858 home** // *Last updated Saturday, 29-Sep-2012 10:55:47 EDT*

10/10

Paper Question 10

Michael Plasmeier

Yes, ForceHTTPS still provides additional protection. Browsers are currently built for maximum compatibility. One of the examples cited in the paper is that some browsers will send the Secure flagged cookie to a site with a separate self-signed certificate after only a warning message. This means an attacker could set up a rouge site with a self-signed certificate that if a user visits (after clicking past the ubiquitous warning) it will send the user's Secure cookie (ie session key) to that rouge site.

6.858

10/10

L10 SSL

(TA [redacted] tth lecturing)

Lab 3 Part 1 Fri

OH 5-7PM 56-191

Thur 3-5PM Stata ~~Lobby~~ 1st floor

Fri 10-12 PM " " "

Security w/ a Network Advesery

before assumed network was safe

nope!

esp on open wifi

attacker can see, modify, ~~*~~ inject, drop packets

So solve w/ Crypto!

- Symmetric - both use same key $c = E_k(m)$ $m = D_k(c)$
- asymmetric - two keys: public + private

- $C = E_{pk}(m)$
- $M = D_{sk}(c)$

but order of magnitude slower

Encryption: Secrecy bt not integrity

Integrity

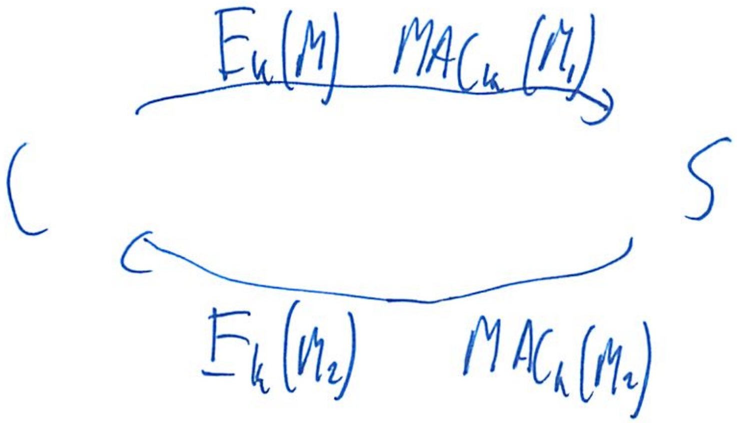
- MAC - message authentication code
    - not like wifi mac address or Apple
    - Uses symetric keys
    - HMAC - like a hash
    - if someone else has key, can veify message

      $MAC_k(m)$
- Public key Crypto - sign a hash of message
    - not directly a hash
    - Sign w/ Sk
    - Veify w/ Pk

Given these - how do we secure communications?

C wants to talk to S

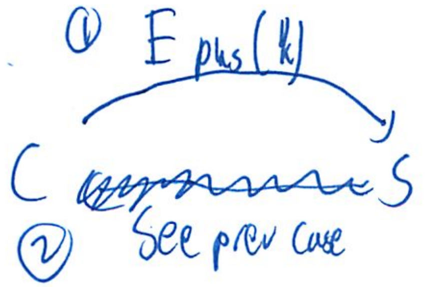
paypal.com



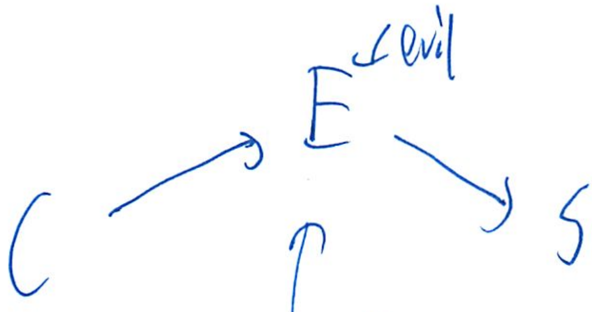
But this assumes you have some sort of pre-shared secret

- Not a good assumption

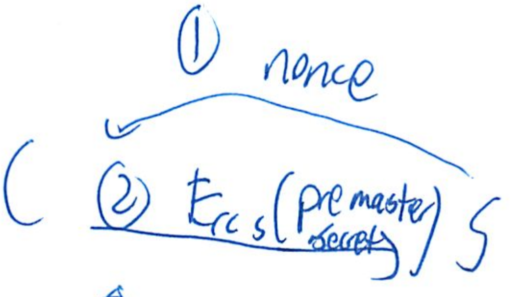
Use public key Crypto to ~~a~~ share initial ~~key~~ symmetric key - which actually encodes the messag



But don't verify that S got proper message



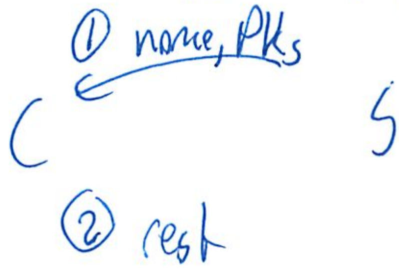
evil can replay msgs
so add a nonce



③ k = f(premaster, nonce)

Premaster Secret - only thing actually secret

But we don't know PayPal's public key



But no clue this Pk actually belongs to S

5

This is often a mess

1. Fall back to trusted authority
   └ certificate signer

T
└ some guy in the sky we all trust
his public key is in our browser

Get server to return certificate signed
(name, public key)

" T belives S public key is Pk_S "

Remember

~~Sign (Sk~~

Sign(Sk, message)
Verify (Pk, signature)

(6)

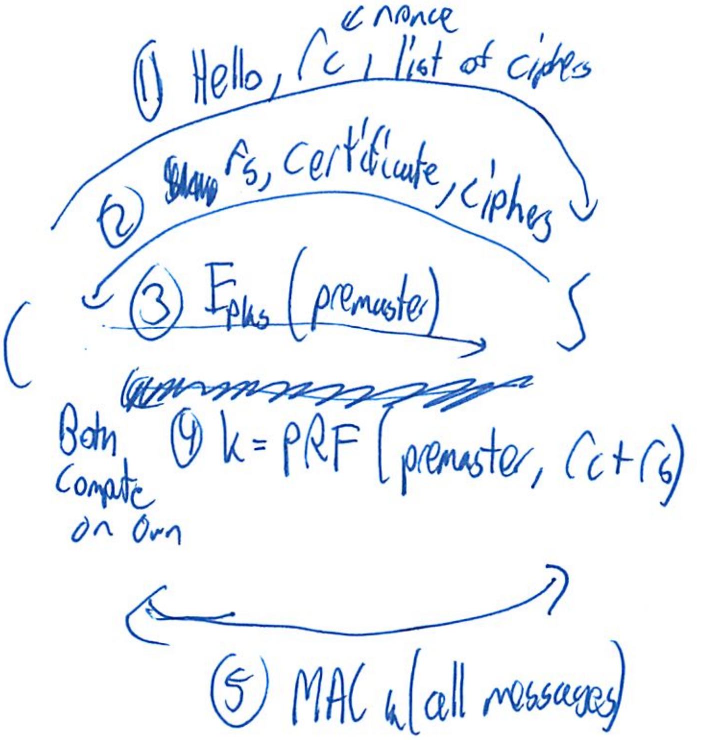
New Protocol: SSL/TLS

https://paypal.com

Everything should be wrapped in SSL

Encrypts + authenticates traffic

Negiotiates ciphers + other features



(1) Hello, r_c (← nonce), list of ciphers

(2) r_s, certificate, ciphers

(3) E_{pk_s}(premaster)

(4) $k = PRF(\text{premaster}, r_c + r_s)$

Both Compute on Own

(5) MAC_k(all messages)

Diffe Hellman is the more complicated version of this

SSL has a speedy session assumption mode
prove knew k
new r_c, r_s to prevent replay

2. Authenticate server

- SSL certificate name matches hostname
  (.paypal.com, PK s)
  or *.paypal.com
- What if network middleman)
  the SK won't be able to decrypt
- But if someone got a false cert
  w/o proving are paypal.com
- 1000s of CAs pre installed

8

Can have a chain

Normally only 2-3 on a chain

Must trust all 100s CAs

Q: Scheme w/ 2 certs?

Could do

Same-Origin

https://paypal.com ≠ http://paypal.com

- We try to follow for HTML ~~else~~ otherwise one could ~~be~~ mess w/ the other

but Cookies don't follow these rules!

Cookies are not scoped to origin

Cookies set by http:// get sent to https://

- "Secure" flag
  - only set by HTTPS
  - Not sent to HTTP

Users

if entering password should know using secure site

🔒 Lock icon

HTTPs in browser

Verify domain name - not paypa1.com
- IE Null Char bug
- Unicode

Problems

1. Crypto
   BEAST
   CRIME
   Servers using weak crypto

2) Authenticating sever

Rely on CAs to only issue valid certs

- Attacker got bad certificate
- As seure as worst CA
  - 100s of them!
  - Comodo
  - DigitiNotar
  - Several a year
- Revocation to remove bad certs
  - have natural ex. dates
  - CAs publish CRL - cert revocation lists
    - too few + not comprehensive
    - too many - huge list
  - OCSP
    - tries to fix
    - online cert stats protocol
    - ask CA online

- but you tell CA every site you visit!
- and Browsers don't like side channels

But also for hotel wifi log in screens

So browser gives warning

If those pages are signed (which should be)

fails open/soft fails

Cert Patrol

SSL Observatory

yells at you if certs changed

but certs change naturally

(12)

Same origin policy issues

pages embedd content

<script>

<style>

<embed>

<script src="http:// ...>

On SSL site

but can intercept req

and change JS

and do anything you want

in the SSL context

Can do funny stuff w/ attributes,

exfiltrate date

Now browsers catch a lot of this

Cookies issues

- Forget seure flag
- No real cookie idenity

- if http cookie, still sent to HTTPS
  - no way for server to tell
- So can still session inject...

Users issues

- Users don't check URL

  paypa1.com

  paypal.com

  Cryilic A

- Users don't check for lock
- Users click through errors

So how do we address these issues?

Forced HTTPs

- flag server sets
- tells browser cert errors are fatal
- no override button
- HTTP redirected to HTTPs
- mixed content blocked

They store this in a cookie

- state exaustion - to prevent attacker from kicking out the HTTPS cookie
- DoS
  if someone else sets cookies for a site that doesn't want
  So only set via header
- Bootstrapping
  Doesn't cover 1st connection

15 My Q's

- DNS method:
  - Need DNSSEC
  - Some ~~browsers~~ networks only return A, not Txt records

It is in most browsers

<u>HSTS</u> - Hypertext strict transport security

Some changes:

Custom Header

STS : max-age = 9999999
↑ Flag

~~Sto~~
Stored in spacial flag - not a cookie
in browser

HTTP → HTTPS redirect

Errors Fatal

but mixed content - some browsers block; not others

Will be an RFC soon

EV

extended verification

CAs put in more work

Blacklist

in browsers

MD5 certs

< 1024 bits

etc

anytime one is found bad

OCSP stapling

- since signed by CA
- so server can present it
- no side channel issue

but most people don't have it in

so hackers opt out

Key Pinning

in Chrome

now hard coded in Browser

list of CAs site supports + uses

so only 3-4 not 100s

but same boot strapping issue

10/10

SSL and HTTPS

```
Administrivia
  Lab 3 parts 1 and 2 due Friday.
  Office hours are weird this week.
   Wed.    5-7pm       56-191
   Thurs.  3-5pm       Stata 1st floor
   Fri.    10am-12pm   Stata 1st floor

Overall problem: security in the presence of a network adversary.
  Web browser communicates with web servers via network.
  Unlike previous lectures, adversary assumed to intercept, modify packets.
  Turns out this is a good model for many situations:
    Nearby adversaries can intercept packets on wired, wireless networks.
    Adversaries can often spoof packets from arbitrary sources.
  How to build secure systems in the presence of such adversaries?

Recall: two kinds of encryption schemes.
  E is encrypt, D is decrypt
  Symmetric key cryptography means same key is used to encrypt & decrypt
    ciphertext = E_k(plaintext)
    plaintext = D_k(ciphertext)
  Asymmetric key (public-key) cryptography: encrypt & decrypt keys differ
    ciphertext = E_PK(plaintext)
    plaintext = D_SK(ciphertext)
    PK and SK are called public and secret (private) key, respectively
  Public-key cryptography is orders of magnitude slower than symmetric

  Encryption provides data secrecy, often also want integrity.
  Message authentication code (MAC) with symmetric keys can provide integrity.
    Look up HMAC if you're interested in more details.
  Can use public-key crypto to sign and verify, almost the opposite:
    Use secret key to generate signature (compute D_SK)
    Use public key to check signature (compute E_PK)

How to secure network communication with cryptography?  (Simple sketch.)
  Suppose two computers already have a shared secret key.
    Use symmetric encryption and MAC to encrypt, authenticate messages.
    Adversary cannot decrypt or tamper with messages.
  What can we do if two computers don't have a shared secret?
  One possibility: two computers know each other's public keys.
    Use public-key encryption (expensive) to exchange symmetric keys.
    Strawman: A picks symmetric key, encrypts with PK_B, sends to B.
    Now fall back to symmetric encryption/MAC case.
  What can go wrong with strawman?
    Adversary can replay all of A's traffic and B would not notice.
    Solution: have the server send a nonce (random value).
      Incorporate the nonce into the final master secret:
        K_master = f(K_pre-master, nonce)
    Adversary can impersonate A, by sending another symmetric key to B.
    Possible solution (one of many; if B cares who A is):
      B also chooses and send a symmetric key to A, encrypted with PK_A.
      Then both A and B use a hash of the two keys combined.
    Adversary can later obtain SK_B, decrypt symmetric key and all messages.
    Solution: use a key exchange protocol like Diffie-Hellman,
      which provides forward secrecy.
  What if neither computer knows each other's public key?
    Common approach: use a trusted third party to generate certificates.
    Certificate is tuple (name, pubkey), signed by certificate authority.
    Meaning: certificate authority claims that name's public key is pubkey.
    B sends A a pubkey along with a certificate.
    If A trusts certificate authority, continue as above.
    The process to establish K_master is called the "handshake"

Plan for securing web browsers: HTTPS
  New protocol: https instead of http (e.g., https://www.paypal.com/).
  1. How to ensure data is not sniffed or tampered with on the network?
    Use SSL (a cryptographic protocol that uses certificates).
    SSL encrypts and authenticates network traffic.
    Negotiate ciphers (and other features: compression, extensions).
    Negotiation is done in clear. Include a MAC of all handshake messages
      to authenticate.
  2. How to ensure that we are talking with the right server?
    SSL certificate name must match hostname in the URL
    In our example, certificate name must be www.paypal.com.
    One level of wildcard is also allowed (*.paypal.com)
    Browsers trust a number of certificate authorities.
    What happens if adversary tampers with DNS records?
      Good news: security doesn't depend on DNS.
      We already assumed adversary can tamper with network packets.
      Wrong server will not know correct private key matching certificate.
  3. How to ensure client-side Javascript cannot be used to subvert security?
    Origin (from the same-origin policy) includes the protocol.
      http://www.paypal.com/ is different from https://www.paypal.com/
      Here, we care about integrity of data (e.g., Javascript code).
      Result: non-HTTPS pages cannot tamper with HTTPS pages.
      Rationale: non-HTTPS pages could have been modified by adversary.
  4. How to ensure user credentials are not sent to wrong server?
    Server certificates help clients differentiate between servers.
    Cookies (common form of user credentials) have a "Secure" flag.
    Secure cookies can only be sent with HTTPS requests.
    Non-Secure cookies can be sent with HTTP and HTTPS requests.
  5. Finally, users can enter credentials directly.  How to secure that?
    Lock icon in the browser tells user they're interacting with HTTPS site.
    Browser should indicate to the user the name in the site's certificate.
    User should verify site name they intend to give credentials to.

How can this plan go wrong?
  As you might expect, every step above can go wrong.
  Not an exhaustive list, but gets at problems that ForceHTTPS wants to solve.

1. Cryptography.
  There have been some attacks on the cryptographic parts of SSL.
  Attack by Rizzo and Duong can allow adversary to learn some plaintext by
    issuing many carefully-chosen requests over a single SSL connection. (BEAST)
  More recent attack by same people using compression, mentioned in iSEC
    lecture. (CRIME)
  Some servers use weak crypto, e.g. certificates signed with MD5.
  But, cryptography is rarely the weakest part of a system.

2. Authenticating the server.
  Adversary may be able to obtain a certificate for someone else's name.
    Used to require a faxed request on company letterhead (but how to check?)
    Now often requires receiving secret token at root@domain.com or similar.
    Security depends on the policy of least secure certificate authority.
    There are 100's of trusted certificate authorities in most browsers.
    Several CA compromises in 2011 (certs for gmail, etc obtained)
    Servers may be compromised and the corresponding private key stolen.
  How to deal with compromised certificate (e.g., invalid cert or stolen key)?
    Certificates have expiration dates.
    Checking certificate status with CA on every request is hard to scale.
    Certificate Revocation List (CRL) published by some CA's, but relatively
      few certificates in them (spot-checking: most have zero revoked certs).
    CRL must be periodically downloaded by client.
```

```
      Could be slow, if many certs are revoked.
      Not a problem if few or zero certs are revoked, but not too useful.
    OCSP: online certificate status protocol.
      Query whether a certificate is valid or not.
    Various heuristics for guessing whether certificate is OK or not.
      CertPatrol, EFF's SSL Observatory, ..
      Not as easy as "did the cert change?". Websites sometimes test new CAs.
    Problem: online revocation checks are soft-fail. An active network attacker
      can just make the checks unavailable. Browsers don't like blocking on a
      side channel. (Performance, single point of failure, captive portals, etc.)
    In practice browsers push updates with blacklist after major breaches.
  SSL implementations have bugs in verifying certificate names.
    Remember important principle from 6.033: "be explicit".
    Certificate contains length (in bytes) followed by that many name bytes.
    Many C implementations store names as standard C strings.
    Some CAs would provide certificates for www.paypal.com\0.attacker.com.
    To non-C code (e.g., Java), looks like a valid attacker.com subdomain.
  Users ignore certificate mismatch errors.
    Despite certificates being easy to obtain, many sites misconfigure them.
    Some don't want to deal with (non-zero) cost of getting certificates.
    Others forget to renew them (certificates have expiration dates).
    End result: browsers allow users to override mismatched certificates.
    About 60% of bypass buttons shown by Chrome are clicked through.

3. Mixing HTTP and HTTPS content.
  Web page origin is determined by the URL of the page itself.
  Page can have many embedded elements:
    Javascript via <SCRIPT> tags
    CSS style sheets via <STYLE> tags
    Flash code via <EMBED> tags
    Images via <IMG> tags
  If adversary can tamper with these elements, could control the page.
    In particular, Javascript and Flash code give control over page.
    CSS gives less control, but still abusable. Particularly with
    complex attribute selectors.
  Probably the developer wouldn't include Javascript from attacker's site.
  But, if the URL is non-HTTPS, adversary can tamper with HTTP response.

4. Protecting cookies.
  Web application developer could make a mistake, forgets the Secure flag.
  User visits http://bank.com/ instead of https://bank.com/, leaks cookie.

  Suppose the user only visits https://bank.com/.  Why is this still a problem?
  1. Adversary can cause another HTTP site to redirect to http://bank.com/.
  2. Even if user never visits any HTTP site, application code might be buggy.
    Some sites serve login forms over HTTPS and serve other content over HTTP.
    Be careful when serving over both HTTP and HTTPS.
      E.g., Google's login service creates new cookies on request.
      Login service has its own (Secure) cookie.
      Can request login to a Google site by loading login's HTTPS URL.
      Used to be able to also login via cookie that wasn't Secure.
      ForceHTTPS solves problem by redirecting HTTP URLs to HTTPS.
      http://blog.icir.org/2008/02/sidejacking-forced-sidejacking-and.html

  Cookie integrity: a non-Secure cookie set on http://bank.com will still be
  sent to https://bank.com. No way to determine who set the cookie.

5. Users directly entering credentials.
  Phishing attacks.
  Users don't check for lock icon.
  Users don't carefully check domain name, don't know what to look for.
    E.g., typo domains (paypa1.com), unicode
  Web developers put login forms on HTTP pages (target login script is HTTPS).
```

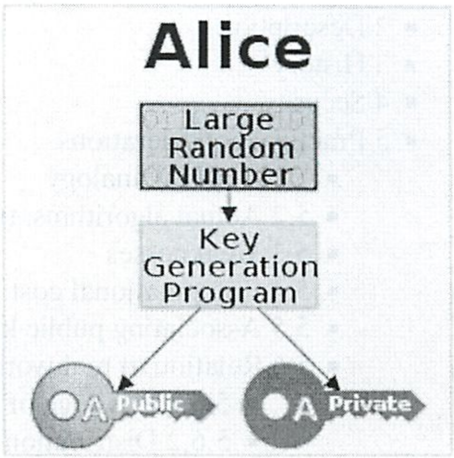
```
      Adversary can modify login form to point to another URL.
      Login form not protected from tampering, user has no way to tell.

  How can we address some of these problems?
    ForceHTTPS (this paper):
      A flag that a server can set for itself.
      - Makes SSL certificate misconfigurations into a fatal error.
      - Redirects HTTP requests to HTTPS.
      - Prohibits non-HTTPS embedding (+ performs ForceHTTPS for them).
      What problems does this solve?  Mostly 2, 3, and to some extent 4.
      Is this really necessary? Can we just only use HTTPS, set Secure
      cookies, etc.?
      - Users can still click-through errors, so it still helps for #2.
      - Not necessary for #3 assuming the web developer never makes a mistake.
      - Still helpful for #4. Marking cookies as Secure gives confidentiality,
        but not integrity. Active attacker can serve fake set at http://bank.com,
        and set cookies for https://bank.com. (https://bank.com cannot distinguish)
      Why not just turn it on for everyone?
      - HTTPS site might not exist
      - If it does, might not be the same site (https://web.mit.edu is
        authenticated, but http://web.mit.edu isn't)
      - HTTPS page may expect users to click through (self-signed certs).

  Implementing ForceHTTPS
    The ForceHTTPS bit is stored in a cookie.
    Interesting issues:
      State exhaustion (the ForceHTTPS cookie getting evicted).
      Denial of service (force entire domain; force via JS; force via HTTP).
      Bootstrapping (how to get ForceHTTPS bit; how to avoid privacy leaks).
        Possible solution 1: DNSSEC.
        Possible solution 2: embed ForceHTTPS bit in URL name (if possible).
        If there's a way to get some authenticated bits from server owner
          (DNSSEC, URL name, etc), should we just get the public key directly?
        Difficulties: users have unreliable networks. Browsers are unwilling
          to block the handshake on a side-channel request.

  Current status of ForceHTTPS:
    Some ideas from ForceHTTPS are being adopted into standards.
    HTTP Strict-Transport-Security header is similar to a ForceHTTPS cookie.
      Uses header instead of magic cookie:
        Strict-Transport-Security: max-age=7884000; includeSubDomains
      Turns HTTP links into HTTPS links.
      Prohibits user from overriding SSL errors (e.g., bad certificate).
      Optionally applies to all subdomains.
        Why is this useful?
        non-Secure and forged cookies can be leaked or set on subdomains.
      Optionally provides an interface for users to manually enable it.
      Implemented in Chrome, Firefox, and Opera.
      Bootstrapping largely unsolved. Chrome has a hard-coded list of preloads.
      Soon to be an RFC.
    IE9 and Chrome block mixed scripting by default. Firefox 18 to follow up.

  Other ways to address problems in SSL
    Better tools / better developers to avoid programming mistakes.
      Mark all sensitive cookies as Secure (#4).
      Avoid any insecure embedding (#3).
      Unfortunately, seems error-prone..
      Does not help end-users (requires developer involvement).
    EV certificates.
      Trying to address problem 5: users don't know what to look for in cert.
      In addition to URL, embed the company name (e.g., "PayPal, Inc.")
      Typically shows up as a green box next to the URL bar.
      Why would this be more secure?
    When would it actually improve security?
    Might indirectly help solve #2, if users come to expect EV certificates.
  Blacklist weak crypto.
    Browsers are starting to reject MD5 signatures on certificates
      (iOS 5, Chrome 18, Firefox 16)
    and RSA keys with < 1024 bits.
      (Chrome 18, OS X 10.7.4, Windows XP+ after a recent update)
  OCSP stapling.
    OCSP responses are signed by CA.
    Server sends OCSP response in handshake instead of querying online (#2).
    Effectively a short-lived certificate.
    Problems:
    - Not widely deployed.
    - Only possible to staple one OCSP response.
  Key pinning.
    Only accept certificates signed by per-site whitelist of CAs.
    Remove reliance on least secure CA (#2).
    Currently a hard-coded list of sites in Chrome.
    Diginotar compromise caught in 2011 because of key pinning.
    Plans to add mechanism for sites to advertise pins (HTTP header, TACK).
    Same bootstrapping difficulty as in ForceHTTPS.
```

References:

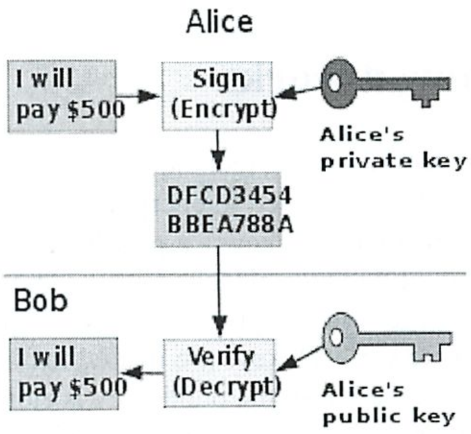
http://www.educatedguesswork.org/2011/09/security_impact_of_the_rizzodu.html
http://en.wikipedia.org/wiki/HTTP_Strict_Transport_Security
http://tools.ietf.org/html/draft-ietf-websec-strict-transport-sec-14
http://blogs.msdn.com/b/ie/archive/2011/06/23/internet-explorer-9-security-part-4-pr
http://blog.chromium.org/2012/08/ending-mixed-scripting-vulnerabilities.html
http://www.imperialviolet.org/2012/07/19/hope9talk.html
http://www.thoughtcrime.org/papers/ocsp-attack.pdf
http://www.imperialviolet.org/2011/03/18/revocation.html
http://www.imperialviolet.org/2012/02/05/crlsets.html
http://tools.ietf.org/html/draft-ietf-websec-key-pinning-02
http://tack.io/
http://dankaminsky.com/2011/08/31/notnotar/

Public-key cryptography

From Wikipedia, the free encyclopedia



In an asymmetric key encryption scheme, anyone can encrypt messages using the public key, but only the holder of the paired private key can decrypt. Security depends on the secrecy of the private key.



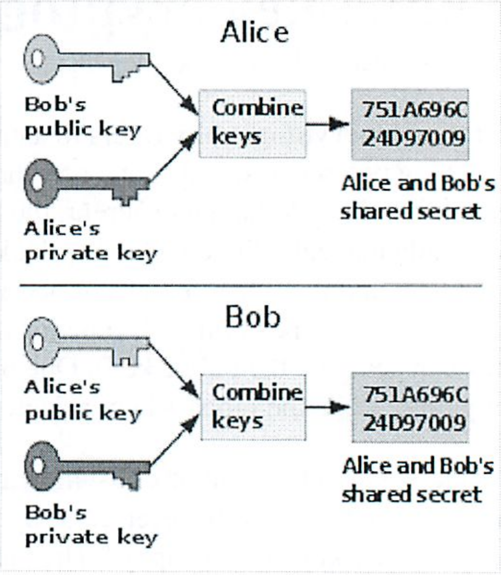
In some related signature schemes, the private key is used to sign a message; anyone can check the signature using the public key. Validity depends on security of the private key.

Public-key cryptography refers to a cryptographic system requiring two separate keys, one of which is secret and one of which is public. Although different, the two parts of the key pair are mathematically linked. One key locks or encrypts the plaintext, and the other unlocks or decrypts the ciphertext. Neither key can perform both functions (however, the private key can generate the public key). One of these keys is published or public, while the other is kept private.

Public-key cryptography uses asymmetric key algorithms (such as RSA), and can also be referred to by the more generic term "asymmetric key cryptography." The algorithms used for public key cryptography are based on mathematical relationships (the most notable ones being the integer factorization and discrete logarithm problems) that presumably have no efficient solution. Although it is computationally easy for the intended recipient to generate the public and private keys, to decrypt the message using the private key, and easy for the sender to encrypt the message using the public key, it is extremely difficult (or effectively impossible) for anyone to derive the private key, based only on their knowledge of the public key. This is why, unlike symmetric key algorithms, a public key algorithm does *not* require a secure initial exchange of one (or more) secret keys between the sender and receiver. The use of these algorithms also allows the authenticity of a message to be checked by creating a digital signature of the message using the private key, which can then be verified by using the public key. In practice, only a hash of the message is typically encrypted for signature verification purposes.

Public-key cryptography is a fundamental, important, and widely used technology. It is an approach used by many cryptographic algorithms and cryptosystems. It underpins such Internet standards as Transport Layer Security (TLS), PGP, and GPG. There are three primary kinds of public key systems: public key distribution systems, digital signature systems, and public key cryptosystems, which can perform both public key distribution and digital signature services. Diffie–Hellman key exchange is the most widely used public key distribution system, while the Digital Signature Algorithm is the most widely used digital signature system.

Contents



In the Diffie–Hellman key exchange scheme, each party generates a public/private key pair and distributes the public key... After obtaining an authentic copy of each other's public keys, Alice and Bob can compute a shared secret offline. The shared secret can be used, for instance, as the key for a symmetric cipher.

How it works

The distinguishing technique used in public-key cryptography is the use of asymmetric key algorithms, where the key used to encrypt a message is not the same as the key used to decrypt it. Each user has a pair of cryptographic keys - a **public encryption key** and a **private decryption key**. The publicly available encrypting-key is widely distributed, while the private decrypting-key is known only to the recipient. Messages are encrypted with the recipient's public key, and can be decrypted *only* with the corresponding private key. The keys are related mathematically, but the parameters are chosen so that determining the private key from the public key is either impossible or prohibitively expensive. The discovery of algorithms that could produce public/private key pairs revolutionized the practice of cryptography, beginning in the mid-1970s.

In contrast, symmetric-key algorithms - variations of which have been used for thousands of years - use a *single* secret key, which must be shared and kept private by both the sender and the receiver, for both encryption and decryption. To use a symmetric encryption scheme, the sender and receiver must securely share a key in advance.

Because symmetric key algorithms are nearly always much less computationally intensive than asymmetric ones, it is common to exchange a key using a key-exchange algorithm, then transmit data using that key and a symmetric key algorithm. PGP and the SSL/TLS family of schemes use this procedures, and are thus called *hybrid cryptosystems*.

Description

The two main uses for public-key cryptography are:

- Public-key encryption: a message encrypted with a recipient's public key cannot be decrypted by anyone except a possessor of the matching private key - it is presumed that this will be the owner of that key and the person associated with the public key used. This is used to attempt to ensure confidentiality.

- Digital signatures: a message signed with a sender's private key can be verified by anyone who has access to the sender's public key, thereby proving that the sender had access to the private key and, therefore, is likely to be the person associated with the public key used. This also ensures that the message has not been tampered with (on the question of authenticity, see also message digest).

An analogy to public-key encryption is that of a locked mail box with a mail slot. The mail slot is exposed and accessible to the public - its location (the street address) is, in essence, the public key. Anyone knowing the street address can go to the door and drop a written message through the slot. However, only the person who possesses the key can open the mailbox and read the message.

An analogy for digital signatures is the sealing of an envelope with a personal wax seal. The message can be opened by anyone, but the presence of the unique seal authenticates the sender.

A central problem with the use of public-key cryptography is confidence (ideally, proof) that a particular public key is correct, and belongs to the person or entity claimed (i.e. is "authentic"), and has not been tampered with, or replaced by, a malicious third party (a "man-in-the-middle"). The usual approach to this problem is to use a public-key infrastructure (PKI), in which one or more third parties - known as certificate authorities - certify ownership of key pairs. PGP, in addition to being a certificate authority structure, has used a scheme generally called the "web of trust", which decentralizes such authentication of public keys by a central mechanism, and substitutes individual endorsements of the link between user and public key. To date, no fully satisfactory solution to this "public key authentication problem" has been found.[*citation needed*]

History

During the early history of cryptography, two parties would rely upon a key using a secure, but non-cryptographic, method. For example, a face-to-face meeting or an exchange, via a trusted courier, could be used. This key, which both parties kept absolutely secret, could then be used to exchange encrypted messages. A number of significant practical difficulties arise with this approach to distributing keys. Public-key cryptography addresses these drawbacks so that users can communicate securely over a public channel without having to agree upon a shared key beforehand.

In 1874, a book by William Stanley Jevons[1] described the relationship of one-way functions to cryptography, and went on to discuss specifically the factorization problem used to create the trapdoor function in the RSA system. In July 1996, one observer[2] commented on the Jevons book in this way:

> In his book *The Principles of Science: A Treatise on Logic and Scientific Method*, written and published in the 1890s,[3] William S. Jevons observed that there are many situations where the "direct" operation is relatively easy, but the "inverse" operation is significantly more difficult. One example mentioned briefly is that enciphering (encryption) is easy while deciphering

> (decryption) is not. In the same section of Chapter 7: Introduction titled "Induction an Inverse Operation", much more attention is devoted to the principle that multiplication of integers is easy, but finding the (prime) factors of the product is much harder. Thus, Jevons anticipated a key feature of the RSA Algorithm for public key cryptography, although he certainly did not invent the concept of public key cryptography.

In 1997, it was publicly disclosed that asymmetric key algorithms were secretly developed by James H. Ellis, Clifford Cocks, and Malcolm Williamson at the Government Communications Headquarters (GCHQ) in the UK in 1973.[4] These researchers independently developed Diffie–Hellman key exchange, and a special case of RSA. The GCHQ cryptographers referred to the technique as "non-secret encryption". This work was named an IEEE Milestone in 2010.[5]

An asymmetric-key cryptosystem was published in 1976 by Whitfield Diffie and Martin Hellman who, influenced by Ralph Merkle's work on public-key distribution, disclosed a method of public-key agreement. This method of key exchange, which uses exponentiation in a finite field, came to be known as Diffie–Hellman key exchange. This was the first published practical method for establishing a shared secret-key over an authenticated (but not private) communications channel without using a prior shared secret. Merkle's "public-key-agreement technique" became known as Merkle's Puzzles, and was invented in 1974 and published in 1978.

A generalization of Cocks's scheme was independently invented in 1977 by Ron Rivest, Adi Shamir and Leonard Adleman, all then at MIT. The latter authors published their work in 1978, and the algorithm appropriately came to be known as RSA. RSA uses exponentiation modulo, a product of two very large primes, to encrypt and decrypt, performing both public key encryption and public key digital signature. Its security is connected to the (presumed) extreme difficulty of factoring large integers, a problem for which there is no known efficient (i.e. practicably fast) general technique. In 1979, Michael O. Rabin published a related cryptosystem that is provably secure, at least as long as the factorization of the public key remains difficult - it remains an assumption that RSA also enjoys this security.

Since the 1970s, a large number and variety of encryption, digital signature, key agreement, and other techniques have been developed in the field of public-key cryptography. The ElGamal cryptosystem, invented by Taher ElGamal. relies on the similar and related high level of difficulty of the discrete logarithm problem, as does the closely related DSA, which was developed at the US National Security Agency (NSA) and published by NIST as a proposed standard. The introduction of elliptic curve cryptography by Neal Koblitz and Victor Miller, independently and simultaneously in the mid-1980s, has yielded new public-key algorithms based on the discrete logarithm problem. Although mathematically more complex, elliptic curves provide smaller key sizes and faster operations for approximately equivalent estimated security.

Security

Some encryption schemes can be proven secure on the basis of the presumed difficulty of a mathematical problem, such as factoring the product of two large primes or computing discrete logarithms. Note that "secure" here has a precise mathematical meaning, and there are multiple different (meaningful) definitions of what it means for an encryption scheme to be "secure". The "right" definition depends on the context in which the scheme will be deployed.

The most obvious application of a public key encryption system is confidentiality - a message that a sender encrypts using the recipient's public key can be decrypted only by the recipient's paired private key. This assumes, of course, that no flaw is discovered in the basic algorithm used.

Another type of application in public-key cryptography is that of digital signature schemes. Digital signature schemes can be used for sender authentication and non-repudiation. In such a scheme, a user who wants to send a message computes a digital signature for this message, and then sends this digital signature (together with the message) to the intended receiver. Digital signature schemes have the property that signatures can be computed only with the knowledge of the correct private key. To verify that a message has been signed by a user and has not been modified, the receiver needs to know only the corresponding public key. In some cases (e.g. RSA), there exist digital signature schemes with many similarities to encryption schemes. In other cases (e.g. DSA), the algorithm does not resemble any encryption scheme.

To achieve both authentication and confidentiality, the sender can first sign the message using his private key and then encrypt both the message and the signature using the recipient's public key.

These characteristics can be used to construct many other (sometimes surprising) cryptographic protocols and applications, such as digital cash, password-authenticated key agreement, multi-party key agreement, time-stamping services, non-repudiation protocols, etc.

Practical considerations

A postal analogy

An analogy that can be used to understand the advantages of an asymmetric system is to imagine two people, Alice and Bob, who are sending a secret message through the public mail. In this example, Alice wants to send a secret message to Bob, and expects a secret reply from Bob.

With a symmetric key system, Alice first puts the secret message in a box, and locks the box using a padlock to which she has a key. She then sends the box to Bob through regular mail. When Bob receives the box, he uses an identical copy of Alice's key (which he has somehow obtained previously, maybe by a face-to-face meeting) to open the box, and reads the message. Bob can then use the same padlock to send his secret reply.

In an asymmetric key system, Bob and Alice have separate padlocks. First, Alice asks Bob to send his open padlock to her through regular mail, keeping his key to himself. When Alice receives it she uses it to lock a box containing her message, and sends the locked box to Bob. Bob can then unlock the box with his key and read the message from Alice. To reply, Bob must similarly get Alice's open padlock to lock the box before sending it back to her.

The critical advantage in an asymmetric key system is that Bob and Alice never need to send a copy of their keys to each other. This prevents a third party - perhaps, in this example, a corrupt postal worker - from copying a key while it is in transit, allowing the third party to spy on all future messages sent between Alice and Bob. So, in the public key scenario, Alice and Bob need not trust the postal service as much. In addition, if Bob were careless and allowed someone else to copy *his* key, Alice's messages to Bob would be compromised, but Alice's messages to other people would remain secret, since the other people would be providing different padlocks for Alice to use.

In another kind of asymmetric key system in which neither party needs to even touch the other party's padlock (or key), Bob and Alice have separate padlocks. First, Alice puts the secret message in a box, and locks the box using a padlock to which only she has a key. She then sends the box to Bob through regular mail. When Bob receives the box, he adds his own padlock to the box, and sends it back to Alice. When Alice receives the box with the two padlocks, she removes her padlock and sends it back to Bob. When Bob receives the box with only his padlock on it, Bob can then unlock the box with his key and read the message from Alice. Note that, in this scheme, the order of decryption is the same as the order of encryption - this is only possible if commutative ciphers are used. A commutative cipher is one in which the order of encryption and decryption is interchangeable, just as the order of multiplication is interchangeable (i.e. A*B*C = A*C*B = C*B*A). A simple XOR with the individual keys is such a commutative cipher. For example, let $E_1()$ and $E_2()$ be two encryption functions, and let "M" be the message so that if Alice encrypts it using $E_1()$ and sends $E_1(M)$ to Bob. Bob then again encrypts the message as $E_2(E_1(M))$ and sends it to Alice. Now, Alice decrypts $E_2(E_1(M))$ using $E_1()$. Alice will now get $E_2(M)$, meaning when she sends this again to Bob, he will be able to decrypt the message using $E_2()$ and get "M". Although none of the keys were ever exchanged, the message "M" may well be a key (e.g. Alice's Public key). This three-pass protocol is typically used during key exchange.

Actual algorithms: two linked keys

Not all asymmetric key algorithms operate in precisely this fashion. The most common ones have the property that Alice and Bob each own *two* keys, one for encryption and one for decryption. In a secure asymmetric key encryption scheme, the private key should not be deducible from the public key. This is known as public-key encryption, since an encryption key can be published without compromising the security of messages encrypted with that key.

In the analogy above, Bob might publish instructions on how to make a lock ("public key"). However, the workings of the lock are such that it is impossible (so far as is known) to deduce from the instructions given just exactly how to make a key that will open that lock (e.g. a "private key"). Those wishing to send messages to Bob must use the public key to encrypt the message, then Bob can use his private key to decrypt it.

Another example has Alice and Bob both choosing a key at random, and then contacting each other to compare the depth of each notch on their keys. Having determined the difference, a locked box is built with a special lock that has each pin inside divided into 2 pins, matching the numbers of their keys. Now the box will be able to be opened with either key, and Alice and Bob can exchange messages inside the box in a secure fashion.

Weaknesses

Of course, there is a possibility that someone could "pick" Bob's or Alice's lock. Among symmetric key encryption algorithms, only the one-time pad can be proven to be secure against any adversary - no matter how much computing power is available. However, there is no public-key scheme with this property, since all public-key schemes are susceptible to a "brute-force key search attack". Such attacks are impractical if the amount of computation needed to succeed - termed the "work factor" by Claude Shannon - is out of reach of all potential attackers. In many cases, the work factor can be increased by simply choosing a longer key. But other algorithms may have much lower work factors, making resistance to a brute-force attack irrelevant. Some special and specific algorithms have been developed to aid in attacking some public key encryption algorithms - both RSA and ElGamal encryption have known attacks that are much faster than the brute-force approach. These factors have changed dramatically in recent decades, both with the decreasing cost of computing power and with new mathematical discoveries.

In practice, these insecurities can be generally avoided by choosing key sizes large enough that the best-known attack algorithm would take so long to have a reasonable chance at successfully "breaking the code" that it is not worth any adversary's time and money to proceed with the attack. For example, if an estimate of how long it takes to break an encryption scheme is one thousand years, and it were used to encrypt details which are obsolete a few weeks after being sent, then this could be deemed a reasonable risk and trade-off.

Aside from the resistance to attack of a particular key pair, the security of the certification hierarchy must be considered when deploying public key systems. Some certificate authority - usually a purpose-built program running on a server computer - vouches for the identities assigned to specific private keys by producing a digital certificate. Public key digital certificates are typically valid for several years at a time, so the associated private keys must be held securely over that time. When a private key used for certificate creation higher in the PKI server hierarchy is compromised, or accidentally disclosed, then a "man-in-the-middle attack" is possible, making any subordinate certificate wholly insecure.

Major weaknesses have been found for several formerly promising asymmetric key algorithms. The 'knapsack packing' algorithm was recently found to be insecure after the development of new attack. Recently, some attacks based on careful measurements of the exact amount of time it takes known hardware to encrypt plain text have been used to simplify the search for likely decryption keys (see "side channel attack"). Thus, mere use of asymmetric key algorithms does not ensure security. A great deal of active research is currently underway to both discover, and to protect against, new attack algorithms.

Another potential security vulnerability in using asymmetric keys is the possibility of a "man-in-the-middle" attack, in which the communication of public keys is intercepted by a third party (the "man in the middle") and then modified to provide different public keys instead. Encrypted messages and responses must also be intercepted, decrypted, and re-encrypted by the attacker using the correct public keys for different communication segments, in all instances, so as to avoid suspicion. This attack may seem to be difficult to implement in practice, but it is not impossible when using insecure media (e.g. public networks, such as the Internet or wireless forms of communications) - for example, a malicious staff member at Alice or Bob's Internet Service Provider (ISP) might find it quite easy to carry out. In the earlier postal analogy, Alice would have to have a way to make sure that the lock on the returned packet really belongs to Bob before she removes her lock and sends the packet back. Otherwise, the lock could have been put on the packet by a corrupt postal worker pretending to be Bob, so as to fool Alice.

One approach to prevent such attacks involves the use of a certificate authority, a trusted third party responsible for verifying the identity of a user of the system. This authority issues a tamper-resistant, non-spoofable digital certificate for the participants. Such certificates are signed data blocks stating that this public key belongs to that person, company, or other entity. This approach also has its weaknesses - for example, the certificate authority issuing the certificate must be trusted to have properly checked the identity of the key-holder, must ensure the correctness of the public key when it issues a certificate, and must have made arrangements with all participants to check all their certificates before protected communications can begin. Web browsers, for instance, are supplied with a long list of "self-signed identity certificates" from PKI providers - these are used to check the *bona fides* of the certificate authority and then, in a second step, the certificates of potential communicators. An attacker who could subvert any single one of those certificate authorities into issuing a certificate for a bogus public key could then mount a "man-in-the-middle" attack as easily as if the certificate scheme were not used at all. Despite its theoretical and potential problems, this approach is widely used. Examples include SSL and its successor, TLS, which are commonly used to provide security for web browsers, for example, so that they might be used to securely send credit card details to an online store.

Computational cost

The public key algorithms known thus far are relatively computationally costly compared with most symmetric key algorithms of apparently equivalent security. The difference factor is the use of typically quite large keys. This has important implications for their practical use. Most are used in hybrid cryptosystems for reasons of efficiency - in such a cryptosystem, a shared secret key ("session key") is generated by one party, and this much briefer session key is then encrypted by each recipient's public key. Each recipient then uses the corresponding private key to decrypt the session key. Once all parties have obtained the session key, they can use a much faster symmetric algorithm to encrypt and decrypt messages. In many of these schemes, the session key is unique to each message exchange, being pseudo-randomly chosen for each message.

Associating public keys with identities

The binding between a public key and its "owner" must be correct, or else the algorithm may function perfectly and yet be entirely insecure in practice. As with most cryptography applications, the protocols used to establish and verify this binding are critically important. Associating a public key with its owner is typically done by protocols implementing a public key infrastructure - these allow the validity of the association to be formally verified by reference to a trusted third party in the form of either a hierarchical certificate authority (e.g., X.509), a local trust model (e.g. SPKI), or a web of trust scheme, like that originally built into PGP and GPG, and still to some extent usable with them. Whatever the cryptographic assurance of the protocols themselves, the association between a public key and its owner is ultimately a matter of subjective judgment on the part of the trusted third party, since the key is a mathematical entity, while the owner - and the connection between owner and key - are not. For this reason, the formalism of a public key infrastructure must provide for explicit statements of the policy followed when making this judgment. For example, the complex and never fully implemented X.509 standard allows a certificate authority to identify its policy by means of an object identifier, which functions as an index into a catalog of registered policies. Policies may exist for many different purposes, ranging from anonymity to military classification.

Relation to real world events

A public key will be known to a large and, in practice, unknown set of users. All events requiring revocation or replacement of a public key can take a long time to take full effect with all who must be informed (i.e. all those users who possess that key). For this reason, systems that must react to events in real time (e.g., safety-critical systems or national security systems) should not use public-key encryption without taking great care. There are four issues of interest:

Privilege of key revocation

A malicious (or erroneous) revocation of some (or all) of the keys in the system is likely, or in the second case, certain, to cause a complete failure of the system. If public keys can be revoked individually, this is a possibility. However, there are design approaches that can reduce the practical chance of this occurring. For example, by means of certificates, we can create what is called a "compound principal" - one such principal could be "Alice and Bob have Revoke Authority". Now, only Alice and Bob (in concert) can revoke a key, and neither Alice nor Bob can revoke keys alone. However, revoking a key now requires both Alice *and* Bob to be available, and this creates a problem of reliability. In concrete terms, from a security point of view, there is now a "single point of failure" in the public key revocation system. A successful Denial of

Service attack against either Alice or Bob (or both) will block a required revocation. In fact, any partition of authority between Alice and Bob will have this effect, regardless of how it comes about.

Because the principle allowing revocation authority for keys is very powerful, the mechanisms used to control it should involve **both** as many participants as possible (to guard against malicious attacks of this type), while at the same time as few as possible (to ensure that a key can be revoked without dangerous delay). Public key certificates that include an expiration date are unsatisfactory in that the expiration date may not correspond with a real-world revocation need - but at least such certificates need not all be tracked down system-wide, nor must all users be in constant contact with the system at all times.

Distribution of a new key

After a key has been revoked, or when a new user is added to a system, a new key must be distributed in some predetermined manner. Assume that Carol's key has been **revoked** (e.g. by exceeding its expiration date, or because of a compromise of Carol's matching private key). Until a new key has been distributed, Carol is effectively "out of contact". No one will be able to send her messages without violating system protocols (i.e. without a valid public key, no one can encrypt messages to her), and messages from her cannot be signed, for the same reason. Or, in other words, the "part of the system" controlled by Carol is, in essence, unavailable. Security requirements have been ranked higher than system availability in such designs.

One could leave the power to create (and certify) keys (as well as to revoke them) in the hands of each user - the original PGP design did so - but this raises problems of user understanding and operation. For security reasons, this approach has considerable difficulties - if nothing else, some users will be forgetful, or inattentive, or confused. On the one hand, a message revoking a public key certificate should be spread as fast as possible, while on the other hand, parts of the system might be rendered inoperable *before* a new key can be installed. The time window can be reduced to zero by always issuing the new key together with the certificate that revokes the old one, but this requires co-location of authority to both revoke keys and generate new keys.

It is most likely a system-wide failure if the (possibly combined) principal that issues new keys fails by issuing keys improperly. This is an instance of a "common mutual exclusion" - a design can make the reliability of a system high, but only at the cost of system availability(and *vice versa*).

Spreading the revocation

Notification of a key certificate revocation must be spread to all those who might potentially hold it, and as rapidly as possible.

There are but two means of spreading information (i.e. a key revocation) in a distributed system: either the information is "pushed" to users from a central point (or points), or else it is "pulled" from a central point(or points) by the end users.

Pushing the information is the simplest solution, in that a message is sent to all participants. However, there is no way of knowing whether all participants will actually *receive* the message. If the number of participants is large, and some of their physical or network distancee are great, then the probability of complete success (which is, in ideal circumstances, required for system security) will be rather low. In a partly updated state, the system is particularly vulnerable to "denial of service" attacks as security has been breached, and a vulnerability window will continue to exist as long as some users have not "gotten the word". Put another way, pushing certificate revocation messages is neither easy to secure, nor very reliable.

The alternative to pushing is pulling. In the extreme, all certificates contain all the keys needed to verify that the public key of interest (i.e. the one belonging to the user to whom one wishes to send a message, or whose signature is to be checked) is still valid. In this case, at least some use of the system will be blocked if a user cannot reach the verification service (i.e. one of the systems that can establish the current validity of another user's key). Again, such a system design can be made as reliable as one wishes, at the cost of lowering security - the more servers to check for the possibility of a key revocation, the longer the window of vulnerability.

Another trade-off is to use a somewhat less reliable, but more secure, verification service, but to include an expiration date for each of the verification sources. How long this "timeout" should be is a decision that requires a trade-off between availability and security that will have to be decided in advance, at the time of system design.

Recovery from a leaked key

Assume that the principal authorized to revoke a key has decided that a certain key must be revoked. In most cases, this happens after the fact - for instance, it becomes known that at some time in the past an event occurred that endangered a private key. Let us denote the time at which it is decided that the compromise occurred as *T*.

Such a compromise has two implications. First, messages encrypted with the matching public key (now or in the past) can no longer be assumed to be secret. One solution to avoid this problem is to use a protocol that has perfect forward secrecy. Second, signatures made with the *no longer trusted to be actually private key* after time *T* can no longer be assumed to be authentic without additional information (i.e. who, where, when, etc.) about the events leading up to the digital signature. These will not always be available, and so all such digital signatures will be less than credible. A solution to reduce the impact of leaking a private key of a signature scheme is to use timestamps.

Loss of secrecy and/or authenticity, even for a single user, has system-wide security implications, and a strategy for recovery must thus be established. Such a strategy will determine who has authority to, and under what conditions one must, revoke a public key certificate. One must also decide how to spread the revocation, and ideally, how to deal with all messages signed with the key since time *T* (which will rarely be known precisely). Messages sent to that user (which require the proper - now compromised - private key to decrypt) must be considered compromised as well, no matter when they were sent.

Examples

Examples of well-regarded asymmetric key techniques for varied purposes include:

- Diffie–Hellman key exchange protocol
- DSS (Digital Signature Standard), which incorporates the Digital Signature Algorithm
- ElGamal
- Various elliptic curve techniques
- Various password-authenticated key agreement techniques
- Paillier cryptosystem
- RSA encryption algorithm (PKCS#1)

Transport Layer Security

From Wikipedia, the free encyclopedia

Transport Layer Security (**TLS**) and its predecessor, **Secure Sockets Layer** (**SSL**), are cryptographic protocols that provide communication security over the Internet.[1] TLS and SSL encrypt the segments of network connections at the Application Layer for the Transport Layer, using asymmetric cryptography for key exchange, symmetric encryption for confidentiality, and message authentication codes for message integrity.

Several versions of the protocols are in widespread use in applications such as web browsing, electronic mail, Internet faxing, instant messaging and voice-over-IP (VoIP).

TLS is an IETF standards track protocol, last updated in RFC 5246, and is based on the earlier SSL specifications developed by Netscape Communications.[2]

Contents

Description

The TLS protocol allows client-server applications to communicate across a network in a way designed to prevent eavesdropping and tampering.

Since most protocols can be used either with or without TLS (or SSL) it is necessary to indicate to the server whether the client is making a TLS connection or not. There are two main ways of achieving this; one option is to use a different port number for TLS connections (for example port 443 for HTTPS). The other is to use the regular port number and have the client request that the server switch the connection to TLS using a protocol specific mechanism (for example STARTTLS for mail and news protocols).

Once the client and server have decided to use TLS they negotiate a stateful connection by using a handshaking procedure.[3] During this handshake, the client and server agree on various parameters used to establish the connection's security.

1. The client sends the server the client's SSL version number, cipher settings, session-specific data, and other information that the server needs to communicate with the client using SSL.
2. The server sends the client the server's SSL version number, cipher settings, session-specific data, and other information that the client needs to communicate with the server over SSL. The server also sends its own certificate, and if the client is requesting a server resource that requires client authentication, the server requests the client's certificate.
3. The client uses the information sent by the server to authenticate the server (see Server Authentication for details). If the server cannot be authenticated, the user is warned of the problem and informed that an encrypted and authenticated connection cannot be established. If the server can be successfully authenticated, the client proceeds to step 4.
4. Using all data generated in the handshake thus far, the client (with the cooperation of the server, depending on the cipher being used) creates the pre-master secret for the session, encrypts it with the server's public key (obtained from the server's certificate, sent in step 2), and then sends the encrypted pre-master secret to the server.
5. If the server has requested client authentication (an optional step in the handshake), the client also signs another piece of data that is unique to this handshake and known by both the client and server. In this case, the client sends both the signed data and the client's own certificate to the server along with the encrypted pre-master secret.
6. If the server has requested client authentication, the server attempts to authenticate the client (see Client Authentication for details). If the client cannot be authenticated, the session ends. If the client can be successfully authenticated, the server uses its private key to decrypt the pre-master secret, and then performs a series of steps (which the client also performs, starting from the same pre-master secret) to generate the master secret.
7. Both the client and the server use the master secret to generate the session keys, which are symmetric keys used to encrypt and decrypt information exchanged during the SSL session and to verify its integrity (that is, to detect any changes in the data between the time it was sent and the time it is received over the SSL connection).
8. The client sends a message to the server informing it that future messages from the client will be encrypted with the session key. It then sends a separate (encrypted) message indicating that the client portion of the handshake is finished.
9. The server sends a message to the client informing it that future messages from the server will be encrypted with the session key. It then sends a separate (encrypted) message indicating that the server portion of the handshake is finished.

The SSL handshake is now complete and the session begins. The client and the server use the session keys to encrypt and decrypt the data they send to each other and to validate its integrity.

This is the normal operation condition of the secure channel. At any time, due to internal or external stimulus (either automation or user intervention), either side may renegotiate the connection, in which case, the process repeats itself.[4]

This concludes the handshake and begins the secured connection, which is encrypted and decrypted with the key material until the connection closes.

If any one of the above steps fails, the TLS handshake fails and the connection is not created.

History and development

Secure Network Programming API

Early research efforts toward transport layer security included the **Secure Network Programming** (SNP) application programming interface (API), which in 1993 explored the approach of having a secure transport layer API closely resembling Berkeley sockets, to facilitate retrofitting preexisting network applications with security measures.[5]

SSL 1.0, 2.0 and 3.0

The SSL protocol was originally developed by Netscape.[6] Version 1.0 was never publicly released; version 2.0 was released in February 1995 but "contained a number of security flaws which ultimately led to the design of SSL version 3.0."[7] SSL version 3.0, released in 1996, was a complete redesign of the protocol produced by Paul Kocher working with Netscape engineers Phil Karlton and Alan Freier. Newer versions of SSL/TLS are based on SSL 3.0. The 1996 draft of SSL 3.0 was published by IETF as a historic document in RFC 6101.

TLS 1.0

TLS 1.0 was first defined in RFC 2246 in January 1999 as an upgrade of SSL Version 3.0. As stated in the RFC, "the differences between this protocol and SSL 3.0 are not dramatic, but they are significant enough that TLS 1.0 and SSL 3.0 do not interoperate. " TLS 1.0 does include a means by which a TLS implementation can downgrade the connection to SSL 3.0, thus weakening security.

On September 23, 2011 researchers Thai Duong and Juliano Rizzo demonstrated a "proof of concept" called **BEAST (Browser Exploit Against SSL/TLS)** using a Java Applet to violate same origin policy constraints, for a long-known Cipher block chaining (CBC) vulnerability in TLS 1.0.[8][9] Practical exploits had not been previously demonstrated for this vulnerability, which was originally discovered by Phillip Rogaway[10] in 2002.

Mozilla updated the development versions of their NSS libraries to mitigate BEAST-like attacks. NSS is used by Mozilla Firefox and Google Chrome to implement SSL. Some web servers that have a broken implementation of the SSL specification may stop working as a result.[11]

Microsoft released Security Bulletin MS12-006 on January 10, 2012, which fixed the BEAST vulnerability by changing the way that the Windows Secure Channel (SChannel) component transmits encrypted network packets.[12]

As a work-around, the BEAST attack can also be prevented by removing all CBC ciphers from one's list of allowed ciphers—leaving only the RC4 cipher, which is still widely supported on most websites.[13][14] Users of Windows 7 and Windows Server 2008 R2 can enable use of TLS 1.1 and 1.2, but this work-around will fail if it is not supported by the other end of the connection and will result in a fall-back to TLS 1.0.

The authors of the BEAST attack are also the creators of the later CRIME attack, which uses data compression as an oracle.

TLS 1.1

TLS 1.1 was defined in RFC 4346 in April 2006.[15] It is an update from TLS version 1.0. Significant differences in this version include:

- Added protection against Cipher block chaining (CBC) attacks.
  - The implicit Initialization Vector (IV) was replaced with an explicit IV.
  - Change in handling of padding errors.
- Support for IANA registration of parameters.

TLS 1.2

TLS 1.2 was defined in RFC 5246 in August 2008. It is based on the earlier TLS 1.1 specification. Major differences include:

- The MD5-SHA-1 combination in the pseudorandom function (PRF) was replaced with SHA-256, with an option to use cipher-suite specified PRFs.
- The MD5-SHA-1 combination in the Finished message hash was replaced with SHA-256, with an option to use cipher-suite specific hash algorithms. However the size of the hash in the finished message is still truncated to 96-bits.
- The MD5-SHA-1 combination in the digitally signed element was replaced with a single hash negotiated during handshake, defaults to SHA-1.
- Enhancement in the client's and server's ability to specify which hash and signature algorithms they will accept.
- Expansion of support for authenticated encryption ciphers, used mainly for Galois/Counter Mode (GCM) and CCM mode of Advanced Encryption Standard encryption.
- TLS Extensions definition and Advanced Encryption Standard CipherSuites were added.

TLS 1.2 was further refined in RFC 6176 in March 2011 redacting its backward compatibility with SSL such that TLS sessions will never negotiate the use of Secure Sockets Layer (SSL) version 2.0.

Applications

In applications design, TLS is usually implemented on top of any of the Transport Layer protocols, encapsulating the application-specific protocols such as HTTP, FTP, SMTP, NNTP and XMPP. Historically it has been used primarily with reliable transport protocols such as the Transmission Control Protocol (TCP).

However, it has also been implemented with datagram-oriented transport protocols, such as the User Datagram Protocol (UDP) and the Datagram Congestion Control Protocol (DCCP), usage which has been standardized independently using the term Datagram Transport Layer Security (DTLS).

A prominent use of TLS is for securing World Wide Web traffic carried by HTTP to form HTTPS. Notable applications are electronic commerce and asset management. Increasingly, the Simple Mail Transfer Protocol (SMTP) is also protected by TLS. These applications use public key certificates to verify the identity of endpoints.

TLS can also be used to tunnel an entire network stack to create a VPN, as is the case with OpenVPN. Many vendors now marry TLS's encryption and authentication capabilities with authorization. There has also been substantial development since the late 1990s in creating client technology outside of the browser to enable support for client/server applications. When compared against traditional IPsec VPN technologies, TLS has some inherent advantages in firewall and NAT traversal that make it easier to administer for large remote-access populations.

TLS is also a standard method to protect Session Initiation Protocol (SIP) application signaling. TLS can be used to provide authentication and encryption of the SIP signaling associated with VoIP and other SIP-based applications.

Security

TLS has a variety of security measures:

- Protection against a downgrade of the protocol to a previous (less secure) version or a weaker cipher suite.
- Numbering subsequent Application records with a sequence number and using this sequence number in the message authentication codes (MACs).
- Using a message digest enhanced with a key (so only a key-holder can check the MAC). The HMAC construction used by most TLS cipher suites is specified in RFC 2104 (SSL 3.0 used a different hash-based MAC).
- The message that ends the handshake ("Finished") sends a hash of all the exchanged handshake messages seen by both parties.
- The pseudorandom function splits the input data in half and processes each one with a different hashing algorithm (MD5 and SHA-1), then XORs them together to create the MAC. This provides protection even if one of these algorithms is found to be vulnerable. *TLS only.*
- SSL 3.0 improved upon SSL 2.0 by adding SHA-1 based ciphers and support for certificate authentication.

From a security standpoint, SSL 3.0 should be considered less desirable than TLS 1.0. The SSL 3.0 cipher suites have a weaker key derivation process; half of the master key that is established is fully dependent on the MD5 hash function, which is not resistant to collisions and is, therefore, not considered secure. Under TLS 1.0, the master key that is established depends on both MD5 and SHA-1 so its derivation process is not currently considered weak. It is for this reason that SSL 3.0 implementations cannot be validated under FIPS 140-2.[16]

A vulnerability of the renegotiation procedure was discovered in August 2009 that can lead to plaintext injection attacks against SSL 3.0 and all current versions of TLS. For example, it allows an attacker who can hijack an https connection to splice their own requests into the beginning of the conversation the client has with the web server. The attacker can't actually decrypt the client-server communication, so it is different from a typical man-in-the-middle attack. A short-term fix is for web servers to stop allowing renegotiation, which typically will not require other changes unless client certificate authentication is used. To fix the vulnerability, a renegotiation indication extension was proposed for TLS. It will require the client and server to include and verify information about previous handshakes in any renegotiation handshakes.[17] This extension has become a proposed standard and has been assigned the number RFC 5746. The RFC has been implemented in recent OpenSSL[18] and other libraries.[19][20]

There are some attacks against the implementation rather than the protocol itself:[21]

- In the earlier implementations, some CAs[22] did not explicitly set basicConstraints CA=FALSE for leaf nodes. As a result, these leaf nodes could sign rogue certificates. In addition, some early software (including IE6 and Konqueror) did not check this field altogether. This can be exploited for man-in-the-middle attack on all potential SSL connections.
- Some implementations (including older versions of Microsoft Cryptographic API, Network Security Services and GnuTLS) stop reading any characters that follow the null character in the name field of the certificate, which can be exploited to fool the client into reading the certificate as if it were one that came from the authentic site, e. g. paypal. com\0.badguy.com would be mistaken as the site of paypal.com rather than badguy.com.
- Browsers implemented SSL/TLS protocol version fallback mechanisms for compatibility reasons. The protection offered by the SSL/TLS protocols against a downgrade to a previous version by an active MITM attack can be rendered useless by such mechanisms.[23]

SSL 2.0 is flawed in a variety of ways:[*citation needed*]

- Identical cryptographic keys are used for message authentication and encryption.
- SSL 2.0 has a weak MAC construction that uses the MD5 hash function with a secret prefix, making it vulnerable to length extension attacks.
- SSL 2.0 does not have any protection for the handshake, meaning a man-in-the-middle downgrade attack can go undetected.
- SSL 2.0 uses the TCP connection close to indicate the end of data. This means that truncation attacks are possible: the attacker simply forges a TCP FIN, leaving the recipient unaware of an illegitimate end of data message (SSL 3.0 fixes this problem by having an explicit closure alert).
- SSL 2.0 assumes a single service and a fixed domain certificate, which clashes with the standard feature of virtual hosting in Web servers. This means that most websites are practically impaired from using SSL.

SSL 2.0 is disabled by default, beginning with Internet Explorer 7,[24] Mozilla Firefox 2,[25] Opera and Safari. After it sends a TLS **ClientHello**, if Mozilla Firefox finds that the server is unable to complete the handshake, it will attempt to *fall back* to using SSL 3.0 with an SSL 3.0 **ClientHello** in SSL 2.0 format to maximize the likelihood of successfully handshaking with older servers.[26] Support for SSL 2.0 (and weak 40-bit and 56-bit ciphers) has been removed completely from Opera as of version 9.5.[27]

Modifications to the original protocols, like **False Start** (adopted and enabled by Google Chrome[28]) or Snap Start, have been reported to introduce limited TLS protocol version rollback attacks[29] or to allow modifications to the cipher suite list sent by the client to the server (an attacker may be able to influence the cipher suite selection in an attempt to downgrade the cipher suite strength, to use either a weaker symmetric encryption algorithm or a weaker key exchange[30]).

TLS handshake in detail

The TLS protocol exchanges *records*, which encapsulate the data to be exchanged. Each record can be compressed, padded, appended with a message authentication code (MAC), or encrypted, all depending on the state of the connection. Each record has a *content type* field that specifies the record, a length field and a TLS version field.

When the connection starts, the record encapsulates another protocol — the handshake messaging protocol — which has *content type* 22.

Simple TLS handshake

A simple connection example follows, illustrating a handshake where the server (but not the client) is authenticated by its certificate:

1. Negotiation phase:
    - A client sends a **ClientHello** message specifying the highest TLS protocol version it supports, a random number, a list of suggested CipherSuites and suggested compression methods. If the client is attempting to perform a resumed handshake, it may send a *session ID*.
    - The server responds with a **ServerHello** message, containing the chosen protocol version, a random number, CipherSuite and compression method from the choices offered by the client. To confirm or allow resumed handshakes the server may send a *session ID*. The chosen protocol version should be the highest that both the client and server support. For example, if the client supports TLS1.1 and the server supports TLS1.2, TLS1.1 should be selected; SSL 3.0 should not be selected.
    - The server sends its **Certificate** message (depending on the selected cipher suite, this may be omitted by the server).[31]
    - The server sends a **ServerHelloDone** message, indicating it is done with handshake negotiation.
    - The client responds with a **ClientKeyExchange** message, which may contain a *PreMasterSecret*, public key, or nothing. (Again, this depends on the selected cipher.) This *PreMasterSecret* is encrypted using the public key of the server certificate.
    - The client and server then use the random numbers and *PreMasterSecret* to compute a common secret, called the "master secret". All other key data for this connection is derived from this master secret (and the client- and server-generated random values), which is passed through a carefully designed pseudorandom function.
2. The client now sends a **ChangeCipherSpec** record, essentially telling the server, "Everything I tell you from now on will be authenticated (and encrypted if encryption parameters were present in the server certificate)." The ChangeCipherSpec is itself a record-level protocol with content type of 20.
    - Finally, the client sends an authenticated and encrypted **Finished** message, containing a hash and MAC over the previous handshake messages.
    - The server will attempt to decrypt the client's *Finished* message and verify the hash and MAC. If the decryption or verification fails, the handshake is considered to have failed and the connection should be torn down.
3. Finally, the server sends a **ChangeCipherSpec**, telling the client, "Everything I tell you from now on will be authenticated (and encrypted, if encryption was negotiated)."
    - The server sends its authenticated and encrypted **Finished** message.
    - The client performs the same decryption and verification.
4. Application phase: at this point, the "handshake" is complete and the application protocol is enabled, with content type of 23. Application messages exchanged between client and server will also be

authenticated and optionally encrypted exactly like in their *Finished* message. Otherwise, the content type will return 25 and the client will not authenticate.

Client-authenticated TLS handshake

The following *full* example shows a client being authenticated (in addition to the server like above) via TLS using certificates exchanged between both peers.

1. Negotiation Phase:
   - A client sends a **ClientHello** message specifying the highest TLS protocol version it supports, a random number, a list of suggested cipher suites and compression methods.
   - The server responds with a **ServerHello** message, containing the chosen protocol version, a random number, cipher suite and compression method from the choices offered by the client. The server may also send a *session id* as part of the message to perform a resumed handshake.
   - The server sends its **Certificate** message (depending on the selected cipher suite, this may be omitted by the server).[31]
   - The server requests a certificate from the client, so that the connection can be mutually authenticated, using a **CertificateRequest** message.
   - The server sends a **ServerHelloDone** message, indicating it is done with handshake negotiation.
   - The client responds with a **Certificate** message, which contains the client's certificate.
   - The client sends a **ClientKeyExchange** message, which may contain a *PreMasterSecret*, public key, or nothing. (Again, this depends on the selected cipher.) This *PreMasterSecret* is encrypted using the public key of the server certificate.
   - The client sends a **CertificateVerify** message, which is a signature over the previous handshake messages using the client's certificate's private key. This signature can be verified by using the client's certificate's public key. This lets the server know that the client has access to the private key of the certificate and thus owns the certificate.
   - The client and server then use the random numbers and *PreMasterSecret* to compute a common secret, called the "master secret". All other key data for this connection is derived from this master secret (and the client- and server-generated random values), which is passed through a carefully designed pseudorandom function.
2. The client now sends a **ChangeCipherSpec** record, essentially telling the server, "Everything I tell you from now on will be authenticated (and encrypted if encryption was negotiated). " The ChangeCipherSpec is itself a record-level protocol and has type 20 and not 22.
   - Finally, the client sends an encrypted **Finished** message, containing a hash and MAC over the previous handshake messages.
   - The server will attempt to decrypt the client's *Finished* message and verify the hash and MAC. If the decryption or verification fails, the handshake is considered to have failed and the connection should be torn down.
3. Finally, the server sends a **ChangeCipherSpec**, telling the client, "Everything I tell you from now on will be authenticated (and encrypted if encryption was negotiated). "
   - The server sends its own encrypted **Finished** message.
   - The client performs the same decryption and verification.
4. Application phase: at this point, the "handshake" is complete and the application protocol is enabled, with content type of 23. Application messages exchanged between client and server will also be encrypted exactly like in their *Finished* message.

Resumed TLS handshake

Public key operations (e. g., RSA) are relatively expensive in terms of computational power. TLS provides a secure shortcut in the handshake mechanism to avoid these operations. In an ordinary *full* handshake, the server sends a *session id* as part of the **ServerHello** message. The client associates this *session id* with the server's IP address and TCP port, so that when the client connects again to that server, it can use the *session id* to shortcut the handshake. In the server, the *session id* maps to the cryptographic parameters previously negotiated, specifically the "master secret". Both sides must have the same "master secret" or the resumed handshake will fail (this prevents an eavesdropper from using a *session id*). The random data in the **ClientHello** and **ServerHello** messages virtually guarantee that the generated connection keys will be different than in the previous connection. In the RFCs, this type of handshake is called an *abbreviated* handshake. It is also described in the literature as a *restart* handshake.

1. Negotiation phase:
   - A client sends a **ClientHello** message specifying the highest TLS protocol version it supports, a random number, a list of suggested cipher suites and compression methods. Included in the message is the *session id* from the previous TLS connection.
   - The server responds with a **ServerHello** message, containing the chosen protocol version, a random number, cipher suite and compression method from the choices offered by the client. If the server recognizes the *session id* sent by the client, it responds with the same *session id*. The client uses this to recognize that a resumed handshake is being performed. If the server does not recognize the *session id* sent by the client, it sends a different value for its *session id*. This tells the client that a resumed handshake will not be performed. At this point, both the client and server have the "master secret" and random data to generate the key data to be used for this connection.
2. The server now sends a **ChangeCipherSpec** record, essentially telling the client, "Everything I tell you from now on will be encrypted. " The ChangeCipherSpec is itself a record-level protocol and has type 20 and not 22.
   - Finally, the server sends an encrypted **Finished** message, containing a hash and MAC over the previous handshake messages.
   - The client will attempt to decrypt the server's *Finished* message and verify the hash and MAC. If the decryption or verification fails, the handshake is considered to have failed and the connection should be torn down.
3. Finally, the client sends a **ChangeCipherSpec**, telling the server, "Everything I tell you from now on will be encrypted. "
   - The client sends its own encrypted **Finished** message.
   - The server performs the same decryption and verification.
4. Application phase: at this point, the "handshake" is complete and the application protocol is enabled, with content type of 23. Application messages exchanged between client and server will also be encrypted exactly like in their *Finished* message.

Apart from the performance benefit, resumed sessions can also be used for single sign-on as it is guaranteed that both the original session as well as any resumed session originate from the same client. This is of particular importance for the FTP over TLS/SSL protocol which would otherwise suffer from a man in the middle attack in which an attacker could intercept the contents of the secondary data connections.[32]

TLS record protocol

This is the general format of all TLS records.

+	**Byte +0**	**Byte +1**	**Byte +2**	**Byte +3**

Byte 0	Content type			
Bytes 1..4	Version		Length	
	(Major)	*(Minor)*	*(bits 15..8)*	*(bits 7..0)*
Bytes 5..(*m*-1)	Protocol message(s)			
Bytes *m*..(p-*1*)	MAC (optional)			
Bytes *p*..(q-*1*)	Padding (block ciphers only)			

Content type

This field identifies the Record Layer Protocol Type contained in this Record.

Content types

Hex	Dec	Type
0x14	20	ChangeCipherSpec
0x15	21	Alert
0x16	22	Handshake
0x17	23	Application

Version

This field identifies the major and minor version of TLS for the contained message. For a ClientHello message, this need not be the *highest* version supported by the client.

Versions

Major Version	Minor Version	Version Type
3	0	SSL 3.0
3	1	TLS 1.0
3	2	TLS 1.1
3	3	TLS 1.2

Length

The length of Protocol message(s), not to exceed 2^{14} bytes (16 KiB).

Protocol message(s)

One or more messages identified by the Protocol field. Note that this field may be encrypted depending on the state of the connection.

MAC and Padding

A message authentication code computed over the Protocol message, with additional key material included. Note that this field may be encrypted, or not included entirely, depending on the state of the connection.

No MAC or Padding can be present at end of TLS records before all cipher algorithms and parameters have been negotiated and handshaked and then confirmed by sending a CipherStateChange record (see below) for signalling that these parameters will take effect in all further records sent by the same peer.

Handshake protocol

Most messages exchanged during the setup of the TLS session are based on this record, unless an error or warning occurs and needs to be signalled by an Alert protocol record (see below), or the encryption mode of the session is modified by another record (see ChangeCipherSpec protocol below).

+	**Byte +0**	**Byte +1**	**Byte +2**	**Byte +3**
Byte 0	22			
Bytes 1..4	Version		Length	
	(Major)	*(Minor)*	*(bits 15..8)*	*(bits 7..0)*
Bytes 5..8	Message type	Handshake message data length		
		(bits 23..16)	*(bits 15..8)*	*(bits 7..0)*
Bytes 9..(*n*-1)	Handshake message data			
Bytes *n*..(*n*+3)	Message type	Handshake message data length		
		(bits 23..16)	*(bits 15..8)*	*(bits 7..0)*
Bytes (*n*+4)..	Handshake message data			

Message type
    This field identifies the Handshake message type.

Message Types	
Code	**Description**
0	HelloRequest
1	ClientHello
2	ServerHello
11	Certificate
12	ServerKeyExchange
13	CertificateRequest

14	ServerHelloDone
15	CertificateVerify
16	ClientKeyExchange
20	Finished

Handshake message data length

This is a 3-byte field indicating the length of the handshake data, not including the header.

Note that multiple Handshake messages may be combined within one record.

Alert protocol

This record should normally not be sent during normal handshaking or application exchanges. However, this message can be sent at any time during the handshake and up to the closure of the session. If this is used to signal a fatal error, the session will be closed immediately after sending this record, so this record is used to give a reason for this closure. If the alert level is flagged as a warning, the remote can decide to close the session if it decides that the session is not reliable enough for its needs (before doing so, the remote may also send its own signal).

+	**Byte +0**	**Byte +1**	**Byte +2**	**Byte +3**
Byte 0	21			
Bytes 1..4	Version		Length	
	(Major)	*(Minor)*	0	2
Bytes 5..6	Level	Description		
Bytes ***7..(p-1)***	MAC (optional)			
Bytes ***p..(q-1)***	Padding (block ciphers only)			

Level

This field identifies the level of alert. If the level is fatal, the sender should close the session immediately. Otherwise, the recipient may decide to terminate the session itself, by sending its own fatal alert and closing the session itself immediately after sending it. The use of Alert records is optional, however if it is missing before the session closure, the session may be resumed automatically (with its handshakes).

Normal closure of a session after termination of the transported application should preferably be alerted with at least the *Close notify* Alert type (with a simple warning level) to prevent such automatic resume of a new session. Signalling explicitly the normal closure of a secure session before effectively closing its transport layer is useful to prevent or detect attacks (like attempts to truncate the securely transported data, if it intrinsically does not have a predetermined length or duration that the recipient of the secured data may expect).

Alert level types

Code	Level type	Connection state
1	**warning**	connection or security may be unstable.
2	**fatal**	connection or security may be compromised, or an unrecoverable error has occurred.

Description

This field identifies which type of alert is being sent.

Alert description types

Code	Description	Level types	Note
0	Close notify	**warning/fatal**	
10	Unexpected message	**fatal**	
20	Bad record MAC	**fatal**	Possibly a bad SSL implementation, or payload has been tampered with e. g. FTP firewall rule on FTPS server.
21	Decryption failed	**fatal**	TLS only, reserved
22	Record overflow	**fatal**	TLS only
30	Decompression failure	**fatal**	
40	Handshake failure	**fatal**	
41	No certificate	**warning/fatal**	SSL 3.0 only, reserved
42	Bad certificate	**warning/fatal**	
43	Unsupported certificate	**warning/fatal**	E. g. certificate has only Server authentication usage enabled and is presented as a client certificate
44	Certificate revoked	**warning/fatal**	
45	Certificate expired	**warning/fatal**	Check server certificate expire also check no certificate in the chain presented has expired
46	Certificate unknown	**warning/fatal**	
47	Illegal parameter	**fatal**	
48	Unknown CA (Certificate authority)	**fatal**	TLS only
49	Access denied	**fatal**	TLS only - E. g. no client certificate has been presented (TLS: Blank certificate message or SSLv3: No Certificate alert), but server is configured to require one.
50	Decode error	**fatal**	TLS only
51	Decrypt error	**warning/fatal**	TLS only
60	Export restriction	**fatal**	TLS only, reserved

70	Protocol version	**fatal**	TLS only
71	Insufficient security	**fatal**	TLS only
80	Internal error	**fatal**	TLS only
90	User cancelled	**fatal**	TLS only
100	No renegotiation	**warning**	TLS only
110	Unsupported extension	**warning**	TLS only
111	Certificate unobtainable	**warning**	TLS only
112	Unrecognized name	**warning**	TLS only; client's Server Name Indicator specified a hostname not supported by the server
113	Bad certificate status response	**fatal**	TLS only
114	Bad certificate hash value	**fatal**	TLS only
115	Unknown PSK identity (used in TLS-PSK and TLS-SRP)	**fatal**	TLS only

ChangeCipherSpec protocol

+	**Byte +0**	**Byte +1**	**Byte +2**	**Byte +3**
Byte 0	20			
Bytes 1..4	Version		Length	
	(Major)	*(Minor)*	0	1
Byte 5	CCS protocol type			

CCS protocol type
: Currently only 1.

Application protocol

+	**Byte +0**	**Byte +1**	**Byte +2**	**Byte +3**
Byte 0	23			
Bytes 1..4	Version		Length	
	(Major)	*(Minor)*	*(bits 15..8)*	*(bits 7..0)*
Bytes 5..(*m*-1)	Application data			

Bytes ***m*..(p-*1*)**	MAC (optional)
Bytes ***p*..(q-*1*)**	Padding (block ciphers only)

Length
Length of Application data (excluding the protocol header and including the MAC and padding trailers)
MAC
20 bytes for the SHA-1-based HMAC, 16 bytes for the MD5-based HMAC.
Padding
Variable length; last byte contains the padding length.

Support for name-based virtual servers

From the application protocol point of view, TLS belongs to a lower layer, although the TCP/IP model is too coarse to show it. This means that the TLS handshake is usually (except in the STARTTLS case) performed before the application protocol can start. The name-based virtual server feature being provided by the application layer, all co-hosted virtual servers share the same certificate because the server has to select and send a certificate immediately after the ClientHello message. This is a big problem in hosting environments because it means either sharing the same certificate among all customers or using a different IP address for each of them.

There are two known workarounds provided by X.509:

- If all virtual servers belong to the same domain, a wildcard certificate can be used. Besides the loose host name selection that might be a problem or not, there is no common agreement about how to match wildcard certificates. Different rules are applied depending on the application protocol or software used.[33]
- Add every virtual host name in the subjectAltName extension. The major problem being that the certificate needs to be reissued whenever a new virtual server is added.

In order to provide the server name, RFC 4366 Transport Layer Security (TLS) Extensions allow clients to include a *Server Name Indication* extension (SNI) in the extended ClientHello message. This extension hints the server immediately which name the client wishes to connect to, so the server can select the appropriate certificate to send to the client.

Implementations

SSL and TLS have been widely implemented in several free and open source software projects. Programmers may use the PolarSSL, CyaSSL, OpenSSL, MatrixSSL, NSS, or GnuTLS libraries for SSL/TLS functionality. Microsoft Windows includes an implementation of SSL and TLS as part of its Secure Channel package. Delphi programmers may use a library called Indy. Comparison of TLS Implementations provides a brief comparison of features of different implementations.

Browser implementations

Further information: Comparison of web browsers

All the current web browsers support TLS:

Browser support for TLS

Browser	Platforms	TLS 1.0	TLS 1.1	TLS 1.2
Chrome 0–22	Linux, Mac OS X, Windows (XP, Vista, 7)[a]	Yes	No	No
Chrome 22–	Linux, Mac OS X, Windows (XP, Vista, 7)[a]	Yes	Yes	No
Firefox 2–	Linux, Mac OS X, Windows (XP, Vista, 7)	Yes[34]	No[35]	No[36]
IE 1–7	Mac OS X, Windows (XP, Vista, 7)[b]	Yes	No	No
IE 8–	Windows 7[b]	Yes	Yes	Yes
Opera 10–	Linux, Mac OS X, Microsoft Windows[c]	Yes	Yes, disabled	Yes, disabled
Safari 5–	Mac OS X, Windows (XP, Vista, 7)[d]	Yes	?	?

Notes:

- a) Google's Chrome browser supports TLS 1.0, and TLS 1.1 from version 22 (after being dropped from version 21). TLS 1.2 is not supported.[37][38]
- b) For versions of IE that support TLS 1.1 and 1.2, both versions are disabled by default. Microsoft's TLS implementation is provided by its Schannel package.[39]
- c) As of Presto 2.2, featured in Opera 10, Opera supports TLS 1.2.[40]
- d) Safari uses OS implementation on Mac OS X, Windows (XP, Vista, 7)[41] with unknown version[42]

Software

- OpenSSL: a free implementation (BSD license with some extensions)
- GnuTLS: a free implementation (LGPL licensed)
- cryptlib: a portable open source cryptography library (includes TLS/SSL implementation)
- JSSE: a Java implementation included in the Java Runtime Environment supports TLS 1.1 and 1.2 from Java 7, although is disabled by default for client, and enabled by default for server[43]
- MatrixSSL: a dual licensed implementation
- Network Security Services (NSS): FIPS 140 validated open source library
- PolarSSL: A tiny SSL/TLS implementation for embedded devices that is designed for ease of use.
- CyaSSL: Embedded SSL/TLS Library with a strong focus on speed and size.

Standards

The current approved version of TLS is version 1.2, which is specified in:

- RFC 5246: “The Transport Layer Security (TLS) Protocol Version 1.2”.

The current standard replaces these former versions, which are now considered obsolete:

- RFC 2246: “The TLS Protocol Version 1.0”.
- RFC 4346: “The Transport Layer Security (TLS) Protocol Version 1.1”.

as well as the never standardized SSL 3.0:

- RFC 6101: “The Secure Sockets Layer (SSL) Protocol Version 3.0”.

Other RFCs subsequently extended TLS.

Extensions to TLS 1.0 include:

- RFC 2595: “Using TLS with IMAP, POP3 and ACAP”. Specifies an extension to the IMAP, POP3 and ACAP services that allow the server and client to use transport-layer security to provide private, authenticated communication over the Internet.
- RFC 2712: “Addition of Kerberos Cipher Suites to Transport Layer Security (TLS)”. The 40-bit cipher suites defined in this memo appear only for the purpose of documenting the fact that those cipher suite codes have already been assigned.
- RFC 2817: “Upgrading to TLS Within HTTP/1.1”, explains how to use the Upgrade mechanism in HTTP/1.1 to initiate Transport Layer Security (TLS) over an existing TCP connection. This allows unsecured and secured HTTP traffic to share the same *well known* port (in this case, http: at 80 rather than https: at 443).
- RFC 2818: “HTTP Over TLS”, distinguishes secured traffic from insecure traffic by the use of a different 'server port'.
- RFC 3207: “SMTP Service Extension for Secure SMTP over Transport Layer Security”. Specifies an extension to the SMTP service that allows an SMTP server and client to use transport-layer security to provide private, authenticated communication over the Internet.
- RFC 3268: “AES Ciphersuites for TLS”. Adds Advanced Encryption Standard (AES) cipher suites to the previously existing symmetric ciphers.
- RFC 3546: “Transport Layer Security (TLS) Extensions”, adds a mechanism for negotiating protocol extensions during session initialisation and defines some extensions. Made obsolete by RFC 4366.
- RFC 3749: “Transport Layer Security Protocol Compression Methods”, specifies the framework for compression methods and the DEFLATE compression method.
- RFC 3943: “Transport Layer Security (TLS) Protocol Compression Using Lempel-Ziv-Stac (LZS)”.
- RFC 4132: “Addition of Camellia Cipher Suites to Transport Layer Security (TLS)”.
- RFC 4162: “Addition of SEED Cipher Suites to Transport Layer Security (TLS)”.
- RFC 4217: “Securing FTP with TLS”.
- RFC 4279: “Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)”, adds three sets of new cipher suites for the TLS protocol to support authentication based on pre-shared keys.

Extensions to TLS 1.1 include:

- RFC 4347: “Datagram Transport Layer Security” specifies a TLS variant that works over datagram protocols (such as UDP).
- RFC 4366: “Transport Layer Security (TLS) Extensions” describes both a set of specific extensions and a generic extension mechanism.
- RFC 4492: “Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)”.
- RFC 4507: “Transport Layer Security (TLS) Session Resumption without Server-Side State”.
- RFC 4680: “TLS Handshake Message for Supplemental Data”.
- RFC 4681: “TLS User Mapping Extension”.
- RFC 4785: “Pre-Shared Key (PSK) Ciphersuites with NULL Encryption for Transport Layer Security (TLS)”.
- RFC 5054: “Using the Secure Remote Password (SRP) Protocol for TLS Authentication”. Defines the TLS-SRP ciphersuites.

- RFC 5081: “Using OpenPGP Keys for Transport Layer Security (TLS) Authentication”, obsoleted by RFC 6091.

Extensions to TLS 1.2 include:

- RFC 5746: “Transport Layer Security (TLS) Renegotiation Indication Extension”.
- RFC 5878: “Transport Layer Security (TLS) Authorization Extensions”.
- RFC 6091: “Using OpenPGP Keys for Transport Layer Security (TLS) Authentication“.
- RFC 6176: “Prohibiting Secure Sockets Layer (SSL) Version 2.0”.
- RFC 6209: “Addition of the ARIA Cipher Suites to Transport Layer Security (TLS)”.

See also

- Multiplexed Transport Layer Security
- Extended Validation Certificate
- SSL acceleration
- Obfuscated TCP
- Server gated cryptography
- tcpcrypt
- Transport Layer Security - Origin Bound Certificates - A proposed protocol extension that improves web browser security via self-signed browser certificates

References and footnotes

1. ^ T. Dierks, E. Rescorla (August 2008). "The Transport Layer Security (TLS) Protocol, Version 1.2" (http://tools.ietf.org/html/rfc5246) . http://tools.ietf.org/html/rfc5246.
2. ^ A. Freier, P. Karlton, P. Kocher (August 2011). "The Secure Sockets Layer (SSL) Protocol Version 3.0" (http://tools.ietf.org/html/rfc6101) . http://tools.ietf.org/html/rfc6101.
3. ^ "SSL/TLS in Detail (http://technet.microsoft.com/en-us/library/cc785811.aspx) ". *Microsoft TechNet*. Updated July 31, 2003.
4. ^ "Description of the Secure Sockets Layer (SSL) Handshake" (http://support.microsoft.com/kb/257591) . Support. microsoft.com. 2008-07-07. http://support.microsoft.com/kb/257591. Retrieved 2012-05-17.
5. ^ Thomas Y. C. Woo, Raghuram Bindignavle, Shaowen Su and Simon S. Lam, *SNP: An interface for secure network programming* Proceedings USENIX Summer Technical Conference, June 1994
6. ^ "THE SSL PROTOCOL" (http://web.archive.org/web/19970614020952/http://home.netscape.com/newsref/std/SSL.html) . Netscape Corporation. 2007. Archived from the original (http://home.netscape.com/newsref/std/SSL.html) on 14 June 1997. http://web.archive.org/web/19970614020952/http://home.netscape.com/newsref/std/SSL.html.
7. ^ Rescorla 2001
8. ^ Dan Goodin (2011-09-19). "Hackers break SSL encryption used by millions of sites" (http://www.theregister.co.uk/2011/09/19/beast_exploits_paypal_ssl/) . http://www.theregister.co.uk/2011/09/19/beast_exploits_paypal_ssl/.
9. ^ "Y Combinator comments on the issue" (http://news.ycombinator.com/item?id=3015498) . 2011-09-20. http://news.ycombinator.com/item?id=3015498.
10. ^ "Security of CBC Ciphersuites in SSL/TLS" (http://www.openssl.org/~bodo/tls-cbc.txt) . 2004-05-20. http://www.openssl.org/~bodo/tls-cbc.txt.
11. ^ Brian Smith (2011-09-30).