Web Security 2

Databases

- Most real web sites are based on databases
- Often, what's of most value is those databases—how can they be protected?
- It takes careful design

Danger Scenarios

- Web server is penetrated via the web interface

☞ A realistic threat, and difficult to defend against

- Web server is penetrated some other way from the outside

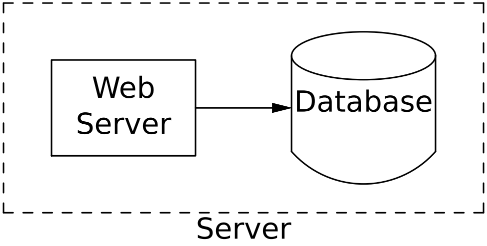
☞ Firewall other ports

- Web server is penetrated from the inside of the company

☞ Internal firewalls and more

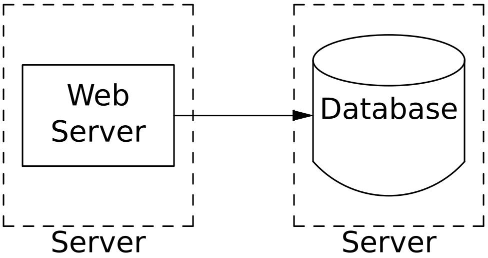
Solution 1: The Same Machine

- Put the database on the same computer
- Communicate via local RPC



Solution 2: Separate Machines

- Put the database on a separate computer
- Communicate over the network



Analysis

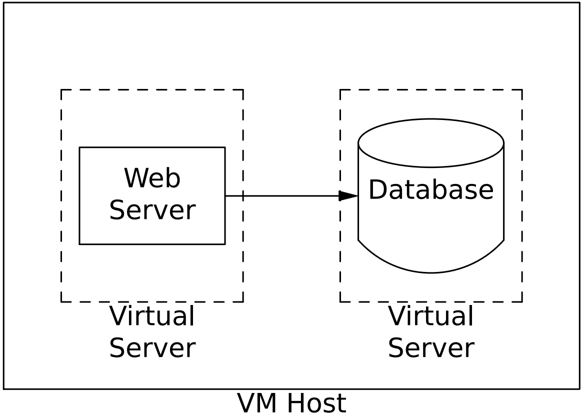
- No difference for inside or outside attacks
- Firewalling still helps
- But what about web server penetration?
- As noted, it's hard to defend against

Web Server on the Same Machine

- If a web script is buggy, the web server can fall
- Get to a shell and steal the entire database that way!
- Partial defense: use separate userIDs for the web server and database, plus set restrictive file permissions
- But: what about local privilege escalation attacks?
- But: the web server *must* have a password to the database

Separate Machines

- Local shells don't help the attacker
- Much better protection
- But: the web server *still* must have a password to the database
- ☞ Separate machines help (especially if you do good logging and intrusion detection), but it's a dangerous situation
- Query: must those be separate physical machines, or would two VMs suffice?



More Generally. . .

- Use separate computers and/or separate userIDs for different functions
- Example: the Apache server is owned by one userID, the content it serves is owned by another user, and the TLS private key is only readable by `root`
- Why?

Separation of Privileges

- Suppose the server is compromised
- The attacker cannot overwrite the executable
- The attacker cannot steal the secret key
- You can protect read-only data by making it not writable by the execution userID

Shedding Privileges

- Apache starts as `root`
- Note: it must be invoked by `root`
- It opens the socket and some log files, then forks and sheds privileges
- Serving web pages is done as non-privileged user “`www`”

File Permissions

- If the web server isn't `root`, it can't open protected files
- All pages served must be readable by the web server
- *Don't* make them owned by `www`; that way, a compromised web server can't overwrite them
- In other words, the web server itself has as few privileges as possible

Design Philosophy

- Use the OS to protect the system against the web server
- Assume the web server can enforce its own access control mechanisms

Can We Lock Things Away?

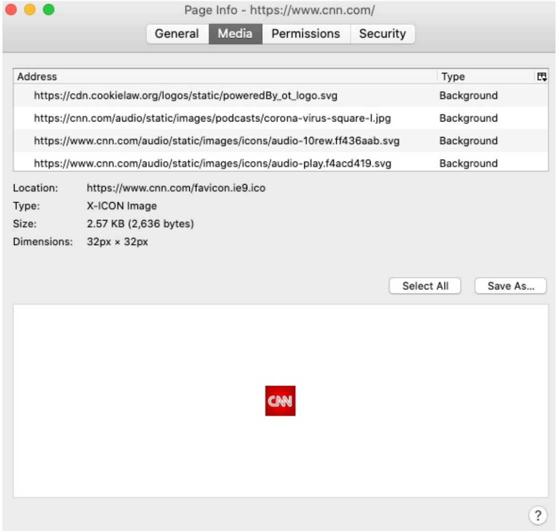
- There is a certain set of files that we want the browser to be able to read
- Most of the files on the system are not in that set
- Can we configure things to prevent www from reading or writing them? Alas, that isn't easy.

What About Browsers?

- Web servers are subject to lots of attacks
- Apart from the ones mentioned, the servers themselves may be buggy
- That problem affects browsers, too—and it some ways, it's worse
- To understand the problem, we need to understand the architecture of today's web

Web Pages

- Web pages are composed of separate elements: the main HTML file, CSS pages (styling information), JavaScript pages, “frames”, and more
- Each element has its own URL
- These URLs can point to different sites
- Translation: you don’t know where parts of the page are actually coming from



Ads and Ad Brokers

- Many commercial sites contain ads
- Where do these ads come from?
- Very few sites host ads themselves
- In fact, they don’t even know what ads are being shown
- They rely on *ad brokers*

Ad Brokers and Frames

- A frame is a separate web page within a page
- Each ad location on a page contains a URL pointing to an ad broker
- (These URLs often contain embedded information to help target the ad, a privacy issue)
- The ad broker effectively auctions the slot (possibly to another ad broker) and issues a `Redirect` response
- The `Redirect` response sends the browser to a new, completely different URL, often hosted by the advertiser
- Neither the user nor the site they want to visit know where the ads will come from—nor who is responsible for their content

Hacking by Advertising

- Suppose some group knows of a hole in, e.g., a JPG library
- They buy an ad from an ad broker pointing to an infected image
- They won't pay much in the auction, so their ad won't be seen by that many people—which helps hide it
- Many people viewing that ad (or the image within it) will be infected
- Bonus: pick sites, targeting information to infect people in a certain group

Cookies and Logins

- Conceptually, the web is “stateless’
- As mentioned, every HTTP transaction is independent—the connection between browser and server is closed after each download
- How do you log in? Where is your shopping cart kept?
- The answer: *cookies*

What is a Cookie?

- A cookie is a small text string sent by a web site to a browser
- When the browser returns to that site, it sends back that string
- That string can contain the login
- Note well: the cookies is returned *every time* you visit that site
- (To see how cookies work, connect to `http://greylock.cs.columbia.edu`—I'll leave it up for a few days after the class)

Usage of “Cookie”

- “Cookie” refers to an opaque value that is sent by one party to another, to be returned to it later
- Cookies have no intrinsic meaning and cannot (safely) be manipulated

NAME

fseek, ftell, rewind – reposition a stream

SYNOPSIS

```
#include <stdio.h>

fseek(stream, offset, ptrname)
FILE *stream;
long offset;

long ftell(stream)
FILE *stream;

rewind(stream)
```

DESCRIPTION

Fseek sets the position of the next input or output operation on the *stream*. The new position is at the signed distance *offset* bytes from the beginning, the current position, or the end of the file, according as *ptrname* has the value 0, 1, or 2.

Fseek undoes any effects of *ungetc*(3).

Ftell returns the current value of the offset relative to the beginning of the file associated with the named *stream*. It is measured in bytes on UNIX; on some other systems it is a magic cookie, and the only foolproof way to obtain an *offset* for *fseek*.

Stealing Cookies: Cross-Site Scripting (XSS)

- Again: any time you visit a site, your cookies for that site are uploaded
- This is how you stay logged in to Google, Facebook, etc.
- (It's also how they track you across the web...)
- Suppose you're logged in to your bank—and an attacker injects a script that goes to your bank and does nasty things
- How?

User-Generated Content

- Many sites allow user-generated content: user profiles, comment fields, chat rooms, etc.
- Some sites allow HTML formatting in such content:

```
I <em>think</em> that...
```

- What if there is JavaScript instead?

```
<script>
    (nasty, evil JavaScript)
</script>
```

- It won't be displayed. It will be executed by the user's browser, and can vist the bank *as that user*
- Defense: web sites must sanitize all user-supplied content—and that isn't easy

Related: Cross-Site Request Forgery (CSRF or XSRF)

- The attacker crafts a single nasty URL
- This is sent to the victim's browser in a way that the user will load it automatically
- One way: as an ad...

Brower-Side Execution

- Web servers can send clients code to execute
- Today, that's JavaScript; in the past, it was also Java and Flash
- Is that safe?
- Not really. . .

Safeguarding Execution

- The Halting Problem tells us that it is impossible to know if an executable is malicious or not
- The only possible approaches are to construct an inherently safe language, one that can't possibly do evil things, or to monitor and block bad operations when they occur
- Java and Flash interpreters have proven to be too buggy
- Besides, other technologies, e.g., more powerful JavaScript and HTML 5, have replaced them

JavaScript

- JavaScript is marketed as a general-purpose programming language
- It is, but it is unclear that it has many advantages over other languages
- Its primary use: much faster, lower-overhead interactivity in web pages
- But: in the past, there have been security holes; today, there are often privacy issues related to browser fingerprinting

Privacy: Web-Tracking

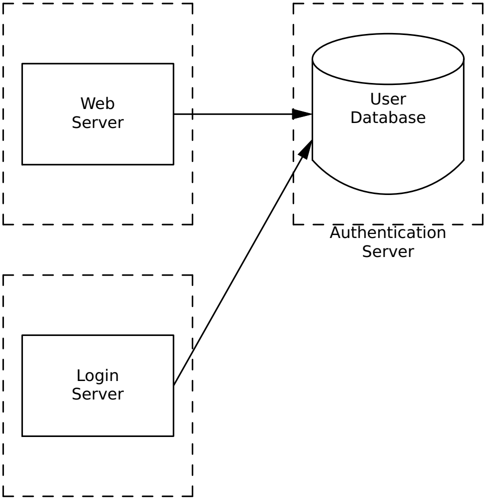
- Suppose you click on a “Share with Twitter” link on a web page
- That request goes to Twitter, along with your Twitter login cookie
- There is also a `Referer:` headre that tells Twitter what page you were coming from
- This implements tracking
- (Google owns Doubleclick, one of the largest ad brokers on the web; they use this to see what ads you click on and what pages you visit. Also: Google Analytics.)

Web Site Logins

- Many sites require user logins
- Ergo, must maintain login and password files
- Passwords must, of course, be salted and hashed for storage
- *Always* put this in a separate database on a separate machine
- Restricted queries to limit risk if the web server is compromised
- *First operation* on all non-login pages: check for a valid logged-in cookie

Storing Credentials

- Legal queries:
  - "Is ⟨user, pw⟩ valid?"
  - "Add ⟨user, pw⟩"
  - "Delete ⟨user, pw⟩"
  - "Generate password reset URL for ⟨user⟩"
  - "Reset credentials to ⟨user, pw⟩"
- All but the first should be done from a separate web server—why?
- How are some of these requests authenticated?



Resetting Passwords

- You *must* have a mechanism for handling lost credentials
- Don’t send password reminders—to do that, you’d need to have a cleartext version of the password
- Secondary authentication questions are weak—and answers are often public or guessable

☞ That’s what happened to Sarah Palin’s email account

Outsourcing Login Security

- Handling logins and passwords is complex
- Many sites outsource this: you log in with your Google, Facebook, or Twitter credentials
- These sites provide that facility because it lets them track users—useful for advertisers...

Browser Security: A General Solution?

- There are many dangers to browsers: executables, buggy code, tracking, etc.
- We were able to provide more server security by using OS features—can we do the same here?
- Yes, but it's not easy

Sandboxing

- Let part of a program run with fewer privileges
- Feature of all modern operating systems
- The hard part: separating the dangerous stuff from stuff that needs full privileges
- (More details later in the term)

Splitting a Browser

- Unsafe: rendering pages, executing JavaScript
- Must support: saving pages, downloading files, `mailto:` URLs, clicking on links in other applications
- Where do cookies live?
- Do we try to sandbox sites from each other, to protect cookies?

Protecting Cookies

- A login cookie is sent from the server to the browser
- The user controls the browser—how does the server know it comes back intact?
- That is: suppose that to user does not treat it as an opaque string, but manipulates it—can trouble occur?
- Sure—so we have to protect it

Protecting Cookies: Method 1

- Generate a key on the server and encrypt the cookie
- Include an integrity check
- Any modification will change the integrity value, so a damaged cookie can be rejected
- You'd think that big companies would understand this, but in 2015 or 2016, "an unauthorised third party accessed [Yahoo's] proprietary code to learn how to forge certain cookies"

Protecting Cookies: Method 2

- Store all data server-side
- Make the cookie a cryptographically strong random number that is only used to find the table entry
- Changing the cookie will simply result in “data not found” and the user will be asked to log in
- “Cryptographically strong”: unpredictable by an attacker, i.e., at least 128 bits and generated by a secure PRNG

Protect All Client-Side Data

- Some software packages store the contents of the user's shopping cart in cookies or other browser-side places
- Does this include prices? Sometimes it does—and people can manipulate these prices
- Or: include a negative quantity, to offset the cost of items you do want

Input Validation

- Many web sites need to validate data, e.g., a certain field must be all numerica
- Often, this is implemented client-side, to provide faster feedback to the user
- Servers can’t trust this!
- The user—that is, the attacker—controls the client

Unsafe URLs

- URLs are a bad place to put sensitive information, e.g., passwords
- URLs are logged, which means that the passwords you properly salt and hash are exposed
- Watch out for guessable userIDs, too
- AT&T once got this wrong—and someone was able to enumerate the space of iPad userIDs

Other Modern Complexities

- Content distribution networks—must ship your content to such sites
- TLS front ends—better for protecting keys, but traffic is then unencrypted after the front end
- Multiple web sites on a single host—how do you protect them from each other?

Web Security

- Web security is *hard*
- I've just given the high-level architecture
- There are many details that matter, e.g., configuration files and file permissions

Questions?

(Barred owl, Central Park, October 11, 2020)