1 Overview

In the last lecture we discussed setuid programs and how to manage privileges within them.

In this lecture we will be looking at security practices for the web, in particular those related to Cross-Site Scripting (XSS).

2 Web Security Model

We begin by describing the problem of web security. With vast numbers of pages handling traffic from a large number of simultaneous users, it is important to enforce multiple security policies to ensure adequate protection.

2.1 Domain Separation

A single user may connect to multiple pages simultaneously. To prevent a malicious page from interfering with a critical page, we wish to ensure that pages cannot attack one another.

The question we wish to answer is this: How can we protect pages from one another?

Earlier, when we studied security vulnerabilities on at the system level, we used protected memory to prevent processes from interfering with one another. One idea that was mentioned was to run scripts from different pages in different processes to make use of this memory protection. But separate processes alone cannot fully protect pages from other malicious pages on the web.

Another idea that was suggested was to limit outgoing requests from pages to that of the origin of the page. This is the core of the Same Origin Policy.

2.2 Same Origin Policy

**Same Origin Policy:** Documents/scripts that are loaded from one origin cannot get or set the properties of documents from different origins.
2.2.1 Origins

We might consider using the IP address of the page as the origin. However, IP addresses might change unexpectedly for perfectly valid reasons, such as load balancing. We wish to use something more abstract for our origin.

The domain name of the page provides an appropriate level of abstraction. We will define our origin in the following way:

Origin: protocol:domain:port
where protocol is either HTTP or HTTPS.

2.2.2 Same Origin Policy

As stated above, the Same Origin Policy ensures that documents that are loaded from one domain cannot access or modify the properties of documents from other domains.

It is important to note that this applies only to the active elements of the page, such as scripts. Static content, such as images and links, are not restricted by this policy.

2.2.3 Remaining Vulnerabilities

Though the Same Origin Policy prevents pages from directly attacking one another, there are alternate methods of attack that it does not prevent. For example:

- Drive-by-download -- A malicious object may downloaded if the user visits a malicious site. This attack can be mitigated by prompting the user for all downloads.
- Browser vulnerability exploits -- A malicious element may exploit the browser to gain access to the larger system.
- Phishing -- A malicious site may deceive the user into exposing valuable information.
- Cross-Site Scripting -- We will discuss this in more detail below.

3 Cross-Site Scripting (XSS)

3.1 Reflective Cross-Site Scripting

Suppose we were to build our own search service. Our service may be written as follows:

<html>
Query: <?php echo $_GET(q) ?>
Results:
...
</html>

In this case, our search query is passed within the URL, for example:

http://foo.com/q="pepper spray"

As we can see, our results page will display not only the results of the search, but the original search query text as well.

If our search query contains valid HTML or Javascript code, then that code will be included in the page returned by foo.com. Moreover, there will be no way for the user’s browser to know that the newly-added code was not intended by foo.com.

3.2 Stored Cross-Site Scripting

Unlike the code in the Reflective XSS attack, which is non-persistent, a Stored XSS attack involves code that is kept by the server for use beyond the immediate page.

For example, Facebook stores the names of its users and displays them not only across multiple pages, but for multiple users as well. If an attacker was able to create a name composed of valid code, then that code would persist within Facebook’s database. The code would be executed whenever a user loaded a page that contained the malicious name.

4 Defenses to Cross-Site Scripting

4.1 Server-side Solutions

4.1.1 Untrusted Tag

One suggestion for preventing a Reflective XSS attack is to modify our search engine code as follows:

<html>
Query: <untrusted> $q </untrusted>
Results:
....
</html>
By enclosing any untrusted input within a special tag, we could alert the browser that some specific elements of the page should be considered unsafe. However, the attacker can easily counter this defense using 'node splitting' to escape the untrusted tag:

```
http://foo.com/q="</untrusted> malicious_code <untrusted>"
```

4.1.2 InfILTERING

Another idea to prevent both Reflective and Stored XSS is to prevent unsafe strings from being uploaded at all. When the input is received, the server should check for any unsafe tags or elements within the string. This is considered the fail-safe default in comparison to outfiltering.

4.1.3 Outfiltering

Outfiltering also checks any input for unsafe tags or elements, but does so when pages are being served, and not when the input initially arrives. There are several proposed benefits to outfiltering:

- **Compromised server** -- If the server is compromised and malicious code is injected directly into the database, then outfiltering prevents it from being sent.
- **Logging/Analysis** -- Outfiltering allows for the storage of the original raw input, which may be useful later on.
- **Stored only** -- If the received input is never sent back out, then there is no need for infiltering.

However, as mentioned, the fail-safe default is infiltering. Outfiltering requires a check for every instance where the data is sent back out, and is more prone to errors because of this.

Outfiltering requires taint tracking.

Tainted Variable/Data: Any value that contains untrusted (i.e. user-supplied) data.

Tainted data obviously includes variables that directly hold user input, but also includes variables that may have been influenced by other tainted variables during operations such as concatenation.

Taint-Tracking Tracking untrusted data.

Taint-tracking is included with some languages, such as Perl, and by way of libraries. It does not work on Stored XSS, since a typical database does not have a means of tracking tainted and untainted data stored together. However, there are those seeking to add taint-tracking to databases.
4.1.4 Inadequacy of Filtering

Most programming languages provide a strict syntax with little or no leeway given to errors. However, because it is most profitable to make sure browsers are as forgiving as possible, web programming often does not share the same strictness.

4.2 Validation

The following scripts may all be considered valid:

- `<script src="foo.js"></script>
- `<script /src="foo.js>
- `<script src="foo.js"></script
- `<p><script>alert();
- `</script>
- `[script src="foo.js"></script>

Because of the many forms allowed, it is very difficult to properly filter user input for the purposes of XSS prevention.

4.3 Client-Server Collaboration to Defend against XSS

A server-only solution is insufficient, since it fails to take into account how the browser will interpret the input. A browser-only solution is insufficient, because the browser can't possibly know which part of the page is to be trusted.

The solution is a collaborative one, in which the server annotates the page to mark untrustworthy elements and the browser enforces the annotations.

How do we implement such a solution?

One proposal is to provide a nonce as part of an `<untrusted>` tag. Of course, the browser must know the nonce to be able to verify it, and so it must be added to the HTTP header. This requires modifying the browser to properly read and enforce the rule.

An alternate solution is to take advantage of the properties of XML, a stricter version of HTML. Using the namespace property of XML and defining our tags, we can annotate the page in such a way that the browser requires no modifications to enforce it. We simply use:

```
<nonce:untrusted> </nonce:untrusted>
```

Without knowledge of the nonce, an attacker is unable to use the node splitting technique to escape the untrusted tag.