Web Security - Servers (October 10, 2012)

Scribe: Wei Wu

October 16, 2012

Today, we covered ways to defend against two types of common web vulnerabilities, SQL injections, and XSS, on the server side.

1 SQL Injection

1.1 Query objects

- Represent each query as an object, perhaps as a query parse tree
- Don't allow input strings to be considered as part of the query object

1.2 ORM

- Some web frameworks provide a specific syntactic library for interacting with the database
- Abstracts database interactions, details of specific database-level protocols
- Represent data model as objects, i.e. Student(sid, name, year)

It's important to note that query objects and ORM are not the same thing. They are separate concepts, but are oftentimes tied together in database APIs in application.

1.3 Embed SQL as DSL

fill me in

1.4 Parameterized/prepared statements

- Precompile a query with parameters to fill in in one step, and insert user-provided inputs as strings in the second step.
- Most common ways to protect against SQL injection
- Compared to other techniques, prepared statements has a lower learning curve in that it only requires knowledge of SQL. Those who know SQL can implement prepared statements.

2 XSS (Cross-site Scripting)

Cross-site scripting can be mitigated by input sanitation, which sanitizes untrusted data that's included in HTML.

2.1 HTML class DOM model

- Expose methods to the developer like `setInnerText("foo")` or `setInnerHtml("foo<i>bar</i>")`.
- `setInnerHtml()` interprets tags inside the param and creates new DOM elements fro the tags. It is probably not safe to expose.
- This technique is not very usable from developers' perspective. It is annoying for developers to learn, and to do in 5 lines what one can accomplish in just 1 line.

2.2 Template system

- Similar to the idea of prepared statements, but for HTML
- Useful because you implicitly declare what parts of a template are dangerous, so you can filter/escape all of the variables that are inserted into the template.
- For use cases like a wiki, where you want users to be able to provide formatted input, you can parse the input and clean it against a whitelist of allowed tags before inserting it into the HTML

Sample template syntax:

```
<html><body><p>Hi ${firstname}!  Your balance is ${balance}.</p>
```

2.3 Parsing app output

Another approach would be to parse all app output before rendering to clean it of unsafe text. There are two major problems with this approach:

- Performance degradation: parsing all output from the application each time before rendering will slow down rendering significantly
- Shadow parser problem: While the application's parser may attempt to clean it up, it may not interpret the output in the same way that the user's browser will. Thus, a cleaned output by the app may still be rendered unsafely by the browser's parser.

2.4 Taint-tracking

Taint-tracking is a technique that can be used to trace untrusted data through the web application pipeline. It can identify which parts of text didn't come from the app, but from user-generated content, so we can filter/sanitize the latter. See the example below. The underlined parts denote user-generated content that would be considered tainted.

```
<p>Hi {username}< /p >
<p>Hi Dave,</p>
<p>Hi <script>alert()</script>,</p>
```

For the above techniques, taint-tracking is only necessary when we parse all app output and sanitize it. We need taint-tracking because we only want to normalize the user-provided input, and need to distinguish parts of the HTML that were generated from the app vs. from the user.