Secure Programming Techniques

Margus Freudenthal, Mart Oruaas

Lecture 6, 2022-03-25
Injection
The Origin of Injection
Von Neumann Architecture

Input Device

Central Processing Unit
  Control Unit
  Arithmetic/Logic Unit

Memory Unit

Output Device
Example: SQL

```sql
UPDATE PERSON
SET NAME = 'John Doe', AGE = 42
WHERE PERSON_ID = 1000;
```
Example: HTML

<button onclick="doStuff()">
  Push me!
</button>
Example: PHP

```html
<html>
  <head>
    <title>PHP Test</title>
  </head>
  <body>
    <?php echo '<p>Hello World</p>'; ?>
  </body>
</html>
```
Example: LDAP

$ ldapsearch -x -b "dc=company,dc=com" \
  -H ldap://192.168.178.29 \
  -D "cn=admin,dc=company,dc=com" \
  -W
Example: `printf`

```
printf("%-15.10s: \n", "Hello, world!");

printf("Fahrenheit: %3d, Celsius: %06.3f \n",
    fahrenheit,
    (5.0 / 9.0) * (fahrenheit - 32));
```
Example: XPath

//button[text() = 'Click me!']

/bookstore/book[price > 35]/title
More Examples

- Any template language, ever
- Most of the query languages
- Any call to `eval()`
- Format strings
- Any call to external tools
  - For example, invoking external commands with shell
All this Can Be Infinitely Nested

```html
<script>
    function getElementByXpath(path) {
        return document.evaluate(
            path, document, null,
            XPathResult.FIRST_ORDERED_NODE_TYPE, null).singleNodeValue;
    }

    console.log(
        getElementByXpath(
            "//html[1]/body[1]/div[1]"));
</script></html>
```
Injection

In the very basic level, injection attack occurs when a data (specified by user/attacker) finds its way to affect program execution flow in ways programmer did not intend.
The Problem

- In the most basic case, injection vulnerabilities result from the confusion of the control plane and the data plane
- Many (most?) modern data representation/query languages have in-band data representation
- In general, the injection vulnerability is created when user input becomes part of a program
  - Many of the more powerful configuration/data presentation languages are quite powerful, if not Turing complete
A Concrete example: SQL Injection

"SELECT * FROM users WHERE name = '' "
+ userName
+ "'"

userName = "' OR '1'='1' -- ";

The result:

SELECT * FROM users
  WHERE name = '' OR '1'='1' -- ;
What to Do 1/3

- Everything starts with proper input validation
- Always use safe APIs in a safe manner
  - Prepared statements, parameterized queries, etc.
  - Use the parameterized query facilities for ORM, NoSQL, etc.
  - `print("%s", foo)` instead of `print(foo)`
  - For shell commands, use the option with explicit argument array
  - Also, for shell commands, use library calls that achieve the same purpose
- Always try to use "structured" data, enforce separation between the structure and the data
What to Do 2/3

- In many cases, parameterized queries are not an option
  - For example, LDAP, a protocol that is often used for user authentication
  - Also, SQL when writing a reporting app
- In these cases, use whitelist validation and escape/encode all the input
  - Always use the (correct) standard library function, do not try to roll your own escape regexp
  - Pay attention to double-encoding issues
What to Do 3/3

- Be veeery careful when doing any string processing or when using templates
- Be very explicit about using and/or mixing different languages
- Never, ever use `eval()` or similar constructs
- OWASP cheat sheets are a useful resource
Cross-Site Scripting (XSS)
How It Works

- Data enters a web application through an untrusted source, most frequently a web request.
- The data is included in dynamic content that is sent to a web user without being validated for malicious content.
  - Typically, the dynamic content is JavaScript, but any other languages can be used as well.
- Potential outcomes include:
  - Stealing user’s cookies (＝hijacking user’s session)
  - Stealing other data from user’s computer
  - Redirecting the user to some other page
  - Modify presentation of page content
Types of XSS

- Stored XSS – injected script is permanently stored on the target server. Typical for comments, reviews, names, items for sale, etc.
  - Blind XSS – injected script is displayed to the user in a backoffice application
- Reflected XSS – the injected script comes from user’s request and is reflected back to the user (for example, in error message, search query, search result, etc.)
  - The attack involves making the user to click on a link containing the necessary parameters
- DOM Based XSS – attack payload is executed by modifying the DOM environment to cause errors in the client JavaScript code. The attack occurs on the client side
Example: Stored XSS

Pre-filled input box:

```html
<input class="inputbox" type="text" name="email" size="40" value="aaa@aa.com" />
```

Possible input:

`aaa@aa.com"&gt;&lt;script&gt;
alert(document.cookie)&lt;/script&gt;`

Resulting HTML from server:

```html
<input class="inputbox" type="text" name="email" size="40" value="aaa@aa.com"><script>
alert(document.cookie)</script>
```
Example: Reflected XSS

Same input box, but the e-mail is taken from the request parameters.

```html
<input class="inputbox" type="text" name="email" size="40" value="aaa@aa.com" />
```

Possible values to use in the email URL parameter:

```
" onfocus="alert(document.cookie)
"/><script>alert(document.cookie)</script>
"/><ScRiPt>alert(document.cookie)</ScRiPt>
"%3cscript%3ealert(document.cookie)%3c/script%3e
```
Example: DOM-Based XSS

Code in vulnerable page:

```html
<script>
document.write("Site is at: "
+ document.location.href + ".");
</script>
```

Attack URL example:

```html
https://example.com/page.html

#<script>alert('xss')</script>
```
XSS: Solution

- Most important: use modern web framework securely
  - Avoid operations marked as insecure
- Validate and escape all the variables
  - If possible, use the automatic encoding protection offered by the framework
- Rich text editing by user is problematic
  - You can try to sanitize user-supplied HTML, but it may be easier to use a simple markup language
- Use Content Security Policy [OWASP, 2021]
- Use cookie attributes (HttpOnly) to restrict their usage
Questions?
References I

Content security policy cheat sheet.