Putting up with input

A03:2021 – Injection

Rose Pezzuti Dyer
August 30, 2023

We will talk about…

- What an injection attack is
- 4 ways to prevent an injection attack
- Demo of SQL Injection and Reflected XSS
- Lessons learned trying to create successful attacks

An ***injection attack*** happens when an application accepts input, which has executable code, and then the app executes the code.

It can affect any app or system that dynamically creates executable code using user input, in many different languages: SQL, javascript (cross-site scripting or JSON injection), shell scripting, and now even AI prompts.

OWASP considered Injection as the number 1 risk until very recently. Now it's number 3. It’s not as prevalent now.

What is the potential impact?

- Bypassing authentication and authorization
    - Access to data that the user should not have access to (exfiltration)
    - Access to system commands and app functions
- Data loss

How can you prevent injection attacks?

Input sanitizing:

1. Use a framework that enforces input sanitizing by default.
2. Use functions that sanitize by default.
3. Examine how dynamic code statements are constructed.

4. Apply principle of least privilege where possible.

Any other ways to prevent injection attacks?

➡️

1. Use a framework that enforces input sanitizing by default.

Example:

Twig templating language sanitizes variables by default before writing them to the DOM. This prevents Reflected XSS attacks.

User input is ‘<script>alert(document.cookie)</script>’.

❌ `<?php echo $var ?>` writes this to the DOM: `<script>alert(document.cookie)</script>`

✅ `{{ var }}` writes this to the DOM: `<script>alert(document.cookie)</script>`

2. Use functions that sanitize by default.

This is a good option if you don’t have a framework to rely on.

Example:

Prepared statements are SQL query templates. User input is handled as bound parameters. The parameters are interpreted as scalars/primitives and will not be interpreted as SQL code.

Question:

Can anyone explain how parameter binding works?

With posted variables username **`1' or 1='1`** and password **`password`**

❌
```
$query = "select * from Users where username = '".$_POST['username']."'
    and password = '".$_POST['password']."';";
$conn->query($query);
```

This will execute:

```
select * from Users where username = '1' or 1='1' and password = 'password';
```

✅
```
$query = "select * from Users where username = ? and password = ?";
$params = [ $_POST['username'], $second = $_POST['password'] ];
$result = $mysqli->execute_query($query, $params);
```

3. Examine how dynamic code statements are constructed.

When you can't rely on a framework or parameterized queries, you need to review code with your own eyeballs. A static analysis tool will help too.

If any statements are built with variables, every variable should be sanitized with a function specific for that language.

Example:

In the absence of a templating language, be sure to html-encode variables before writing to the DOM.

User input is `<script>alert(document.cookie)</script>`.

```
<?php echo htmlspecialchars($var, ENT_QUOTES, ‘UTF-8') ?>
```

Writes out: `&lt;script&gt;alert(document.cookie)&lt;/script&gt;`

```
<?php echo htmlentities($_GET['term'], ENT_HTML5) ?>
```

Writes out: `&lt;script&gt;alert&lpar;document&period;cookie&rpar;&lt;&sol;script&gt;`

Example:

In application code, check all variables used to build SQL statements.

With posted variables username **`1' or 1='1`** and password **`password`**

❌ An unsafe query:

```
$query = "select * from Users where username = '".$_POST['username']."' and password = '".$_POST['password']."';";
 $conn->query($query);
```

This will execute:

```
    select * from Users where username = '1' or 1='1' and password = 'password';
```

✅ Escaped variables:

```
$query = "select * from Users where
  username = '".$mysqli->real_escape_string($_POST['username'])."' and
  password = '".$mysqli->real_escape_string($_POST['password'])."';";
$conn->query($query);
```

This will execute:

```
    select * from Users where username = '1\' or 1=\'1' and password = 'password';
```

Example:

In database code, check all stored procedures for dynamic SQL.

```
CREATE PROCEDURE GetCustomerInfo (@LastName varchar(200))
AS
```

❌ `EXEC ('select * from Customers where LastName = '''+@LastName+''');`

✅ `select * from Customers where LastName = @LastName;`

4. Apply principle of least privilege where possible.

Do not let your application act as an admin user or a root user.

Example: Use a db user with reduced permissions in your application. Don’t allow this db user to create or drop objects.

Do not allow every app user to save strings that could be used as executable code.

Example: In the comments section of a blog, allow only privileged users to save javascript in their submissions.

MAMP Attacks

Here is a little demo to demonstrate:

- SQL injection and mitigation in PHP
- Reflected XSS and mitigation
  - BONUS: Phishing email
  - BONUS: Session hijacking

I learned some things preparing the MAMP Attacks demo…

It's easy to avoid injection attacks for most coding.

Good frameworks and libraries remove so much decision-making and enforce best practices so effectively, that most of your code will probably be safe.

Attacks need to be pretty creative to be successful.

It was difficult to come up with attacks against an app that takes no precautions against injection.

I wasn’t able to succeed without knowing the app code and database.

And I had to ignore all the warnings about unsafe functions in coding manuals.

Some attacks required several steps and were dependent on some other knowledge that would have come from previous attacks.

Question: Can anyone guide us through a way to sniff out vulnerabilities?

Layering app security is a good strategy.

Taking any precautions at all will positively impact the security of your app.

What if security in one or more layers in your app's stack is deficient?

Be sure to use safe coding practices in as many layers as possible.