

XSS Everywhere!

What is it?

Why should I care?

How can I avoid it?

Nick Blundell — [AppCheck NG](#)

About Us

- We provide a vulnerability scanning service via software forged through day-to-day pen testing experience. <http://appcheck-ng.com>
- We specialise in Web App security
- We live and breathe vulnerability research, and we continue to push our scanning technology to the limits... and beyond

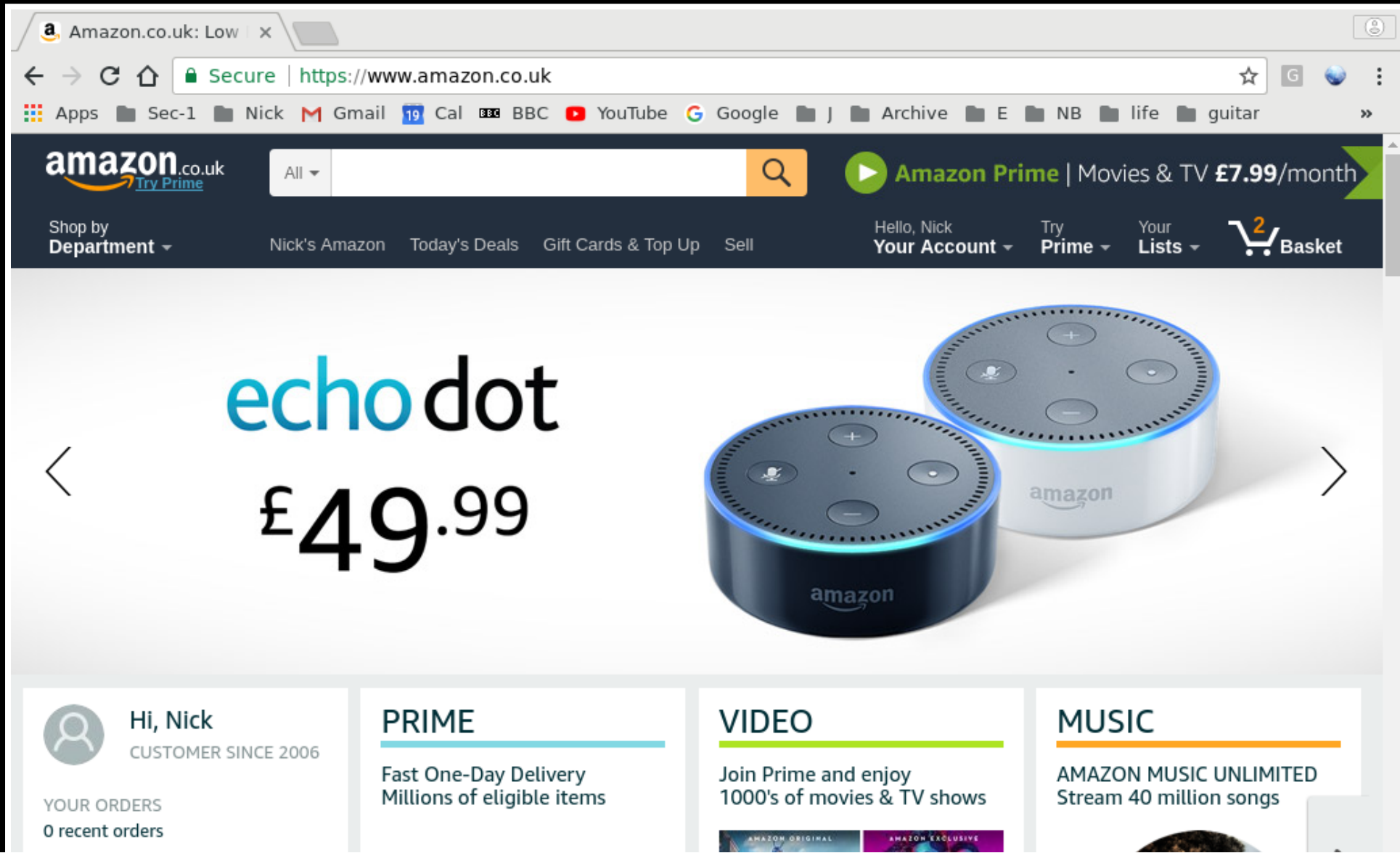
Overview

- What is a Web Application?
- What is XSS?
- How can we avoid it?
- How can it be mitigated?

What is a Web Application?

It is an application that involves your *Browser* and a
Web Server

You put a URL in your browser and it loads a (HTML) page



The image is a screenshot of a web browser displaying the Amazon.co.uk website. The browser's address bar shows the URL <https://www.amazon.co.uk>. The page features the Amazon logo, a search bar, and navigation links for Amazon Prime, account, lists, and basket. The main content area displays an advertisement for the Echo Dot smart speaker, priced at £49.99. Below the main content, there are four personalized recommendation tiles: 'Hi, Nick' (Customer since 2006), 'PRIME' (Fast One-Day Delivery), 'VIDEO' (Join Prime and enjoy 1000's of movies & TV shows), and 'MUSIC' (Amazon Music Unlimited).

Amazon.co.uk: Low x

Secure | <https://www.amazon.co.uk>

Apps Sec-1 Nick Gmail 19 Cal BBC BBC YouTube Google J Archive E NB life guitar

amazon.co.uk Try Prime

All

Amazon Prime | Movies & TV £7.99/month

Shop by Department Nick's Amazon Today's Deals Gift Cards & Top Up Sell Hello, Nick Your Account Try Prime Your Lists Basket

echo dot

£49.99

amazon

amazon

Hi, Nick
CUSTOMER SINCE 2006

YOUR ORDERS
0 recent orders

PRIME
Fast One-Day Delivery
Millions of eligible items

VIDEO
Join Prime and enjoy
1000's of movies & TV shows

MUSIC
AMAZON MUSIC UNLIMITED
Stream 40 million songs

Your *browser* then renders the page from the response's *HTML*, which looks like this:

```
<!DOCTYPE html>
<html>
<body>

<h1>My First Heading</h1>

<p>My first paragraph</p>

<a href="/some_other_page">Visit other page</a>

</body>
</html>
```

- The HTML usually also contains *scripts*, typically *JavaScript* which the browser executes alongside rendering the page

```
<!DOCTYPE html>
<html>
<body>

<h1>My First Heading</h1>

<script>window.alert("Hello, World")</script>

</body>
</html>
```

- This script *execution* is what enables the *dynamic* nature of day-to-day web applications

You Have Mail

Congratulations: You have WON an iPhone X!

Well done, Ben. As a long-term and valued member of our *online banking service*, you were *shortlisted* and selected as *the lucky winner of an iPhone X*




Claim your iPhone X

That's Fantastic!

Hmmm... but wait...?

Erm, why is my bank balance now at *zero*?

Social engagement	
Year-end sale	
Total Funds	£0.00



Probably just a temporary issue...

... and nice about the iPhone

Time for a little scrutiny

That link in my email: [Claim your iPhone X](#)
points to this (yikes!):

[http://secure-bank.com:8686/mini_pres_app/vanilla?](http://secure-bank.com:8686/mini_pres_app/vanilla?report_id=A6FE25423%3Cscript%3Epayload%20%3D%20%22%24(%27%23firstNa)

```
report_id=A6FE25423%3Cscript%3Epayload%20%3D%20%22%24(%27%23firstName%27).val(%27Evil%27)%3B%24(%27%23lastName%27).val(%27Hacker%27)%3B%24(%27%23account_num%27).val(1236532)%3B%24(%27%23sort_code%27).val(547345)%3B%24(%27%23amount%27).val(12354434)%3B%24(%27%23do_transfer%27).click()%22%0dsetTimeout(function(){w=$(%22iframe%22)[0].contentWindow%0dw.eval(payload)},1000)%3B%24(%22body%22).prepend(%27%3Cimg%20src%3D%22%2Fcongrats.jpg%22%20style%3D%22margin-left%3A300px%22%3E%27)%3C/script%3E%3Ciframe%20name=%22theFrame%22width=%22100%%22%20height=%22300%%22src=%22transfer_money%22%3E
```


Time for a little scrutiny

- But still... just looks like some long URL, and I'm used to seeing obscure stuff like this
- And it *is* pointing to the secure banking application that I'm familiar with: secure-bank.com

Let's make that clearer

- If we *URL decode* that (i.e. change patterns like %3C to the characters they represent, like <), it looks like this:

```
http://secure-bank.com:8686/mini_pres_app/vanilla?
```

```
report_id=A6FE25423<script>payload =
```

```
"$('#firstName').val('Evil');$('#lastName').val('Hacker');$('#account_num').val(1236532);$('#sort_code').val(547345);$('#amount').val(12354434);$('#do_transfer').click()"
```

```
setTimeout(function(){w=$("#iframe")
```

```
[0].contentWindoww.eval(payload)},1000);$("#body").prepend('')</script><iframe
```

```
name="theFrame"width="100%" height="300%"src="transfer_money">
```

This doesn't look right to me...

```
$('#firstName').val('Evil');$('#lastName').val('Hacker')  
...  
$('#do_transfer').click()
```

Looks like the intention is hidden here

- They are *masking* (i.e. hiding) something with this image:

```
$("#body").prepend('')
```

Looks like the intention is hidden here

- Let's take away the mask, and see...

Claim your iPhone X (unmasked attack)

Nooooo... we Just got XSS-ed

- This was a bog standard **XSS** attack...
- with a devastating impact
- A simple slip up can result in stuff like this
- Let's figure out how all of that just happened

**What is a Cross-Site Scripting
(XSS) Vulnerability?**

What is a Cross-Site Scripting (XSS) Vulnerability?

- Recall that the web server responds to the browser with `HTML` pages
- These pages have a *structure* which describes how the page is to be laid out visually
- That structure can also contain `script` elements, which run code on the page, to do fancy stuff, like displaying a new social media event

What is a Cross-Site Scripting (XSS) Vulnerability?

- Usually, an application does not want the user to directly control the `HTML` structure...
- ... perhaps just portions of it, like actual content ...
- ... and definitely, the application does not want users to add arbitrary `script` elements to the pages

What is a Cross-Site Scripting (XSS) Vulnerability?

- *Confidentiality is lost* when a single instance of *XSS* occurs...
- ... where a user's input is *mishandled* to allow it to alter the wider *HTML* structure of a page
- An attacker will usually then exploit this to run a script on a page accessed by another user
- ... so the hacker's script will run *within another user's private/authenticated session*

What is a Cross-Site Scripting (XSS) Vulnerability?

Depending on the context, an attack can be *delivered* either:

- *Directly*: by a crafted *URL* to a victim (via email, other websites, etc.), known as a *Reflected XSS* attack
- *Indirectly*: by the hacker laying a trap in the application itself, for victims to stumble upon, known as *Persistent XSS*

How bad can it be?

- Users may be exploited to the hacker's financial advantage
- *Admin* accounts may be targeted, leading to *full application compromise*
 - With admin control of the application, an attacker will then seek to *compromise the host* and *all data* accessible from it
 - ... usually through some privileged admin feature like for system configuration

Let's start with the basics: a vanilla example

When we are logged into our online banking app, and we view a report on the following URL:

http://secure-bank.com:8686/mini_pres_app/vanilla?report_id=A6FE25423

A vanilla example

- The first thing we notice is that the ID of the report is *reflected* in the page from the web server

Requesting this URL:

http://secure-bank.com:8686/mini_pres_app/vanilla?report_id=A6FE25423

Gives this response, with the value reflected:

```
<h2>Viewing Report</h2>
```

```
Client: Mr Jones
```

```
Report ID: A6FE25423
```

A vanilla example

So the next question on an attacker's mind is:

How does the application *respond* if I try to input the special **HTML** tag brackets, `<` and `>` ?

`http://secure-bank.com:8686/mini_pres_app/vanilla?
report_id=A6FE25423evil`

```
<h2>Viewing Report</h2>  
Client: Mr Jones  
Report ID: A6FE25423<b>evil</b>
```

A vanilla example

A vanilla example

- So the hacker can create an **HTML** tag on the victim's page...
- ... how about running a *script* in the victim's page (i.e. in their authenticated session)

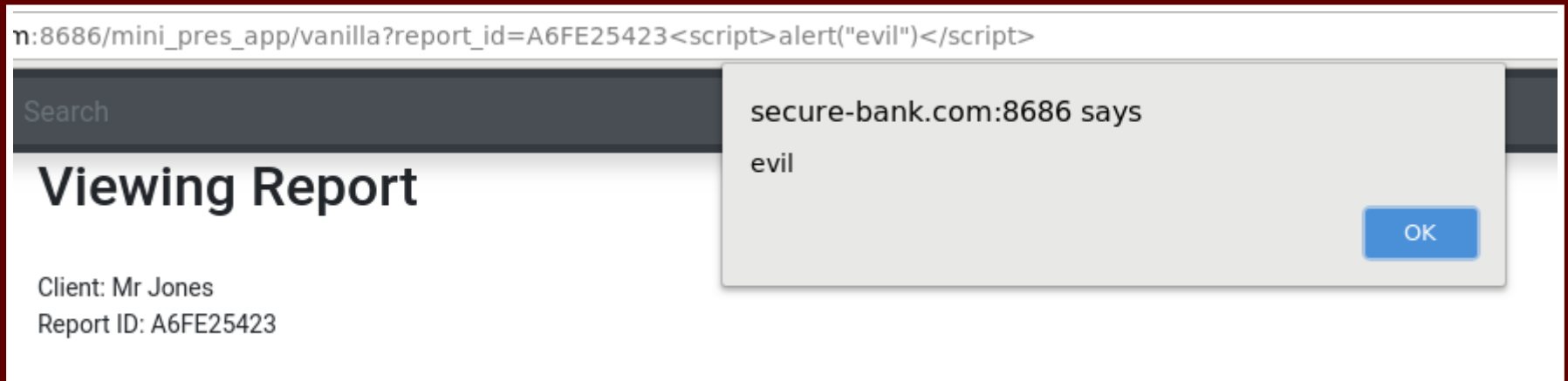
[http://secure-bank.com:8686/mini_pres_app/vanilla?
report_id=A6FE25423](http://secure-bank.com:8686/mini_pres_app/vanilla?report_id=A6FE25423)**<script>alert("evil")</script>**

```
<h2>Viewing Report</h2>  
Client: Mr Jones  
Report ID: A6FE25423<script>alert("evil")</script>
```

- We see the script tag was injected *without any interference through validation*

A vanilla example

- And so we see: the hacker's script *executes*
- Note, often *XSS* is demonstrated by popping open an *alert box*...
- ... this is proof the hacker can perform *any action* on the victim's page.



Thwarting the Attacker

"Right, Mr Hacker, you mess with us... we mess with you!"

Thwarting the Attacker (Take 1)

- Approach: If the hacker tries to inject a *script* tag, we are going to strip the keyword out!

Thwarting the Attacker (Take 1)

Hacker does:

[http://secure-bank.com:8686/mini_pres_app/vanilla?
report_id=A6FE25423](http://secure-bank.com:8686/mini_pres_app/vanilla?report_id=A6FE25423)`<script>alert("evil")</script>`

Application responds:

```
<h2>Viewing Report</h2>  
Client: Mr Jones  
Report ID: A6FE25423<>alert("evil")</>
```

We Win: Boof: the *script* keyword was stripped out so no malicious script will execute.

Thwarting the Attacker (Take 1)

Then hacker does:

```
http://secure-bank.com:8686/mini_pres_app/vanilla?  
report_id=A6FE25423<sCriPt>alert("evil")</sCriPt>
```

Application does:

```
<h2>Viewing Report</h2>  
Client: Mr Jones  
Report ID: A6FE25423<sCriPt>alert("evil")</sCriPt>
```

We Lose: we stripped only lowercase *script* but HTML tags are case insensitive... so the hacker wins again!

Thwarting the Attacker (Take 2)

- Approach: Strip out the keyword *script whatever the case* of its characters!

Thwarting the Attacker (Take 2)

Then hacker does:

```
http://secure-bank.com:8686/mini_pres_app/vanilla?  
report_id=A6FE25423<sCrScripTiPt>alert("evil")</sCrScripTiPt>
```

Application does:

```
<h2>Viewing Report</h2>  
Client: Mr Jones  
Report ID: A6FE25423<sCriPt>alert("evil")</sCriPt>
```

We Lose: Hmmmm, this guy knows some tricks: He anticipated me stripping out the keyword and created a *keyword sandwich* which made my efforts look really lame.

Thwarting the Attacker (Take 3)

- Approach: If the hacker is going to sandwich keywords like that...
- ... I will now repeatedly strip out the keyword *script* until there are no more

Thwarting the Attacker (Take 3)

So now when the hacker does:

`http://secure-bank.com:8686/mini_pres_app/vanilla?
report_id=A6FE25423<sCrScriPt>alert("evil")</sCrScriPt>`

Application does:

```
<h2>Viewing Report</h2>  
Client: Mr Jones  
Report ID: A6FE25423<>alert("evil")</>
```

We Win: The keyword *script* is stripped out, *regardless of its case* and if *sandwiched*

Thwarting the Attacker (Take 3)

But... but... but... then the hacker does:

```
http://secure-bank.com:8686/mini_pres_app/vanilla?  
report_id=A6FE25423<img+src=1+onerror=alert("evil")>
```

Application does:

```
<h2>Viewing Report</h2>  
Client: Mr Jones  
Report ID: A6FE25423<img src=1 onerror=alert("evil")>
```

We Lose: Whaaat! So they injected an *img* tag instead of a *script* tag to run a script!!!

Thwarting the Attacker (Take 4)

- Okay, it turns out there are *many ways* to cause a script to run in a browser
- ... so blocking the keywords like *script* is a *complete waste of time*

Thwarting the Attacker (Take 4)

- Right, that's it!, Mr Hacker
- We know what needs to be done:
- Strip out any *tag brackets*, so these: < and >
- Then an attacker can inject *no* tags into the page: no *script*, no *img*... no nothing.

Thwarting the Attacker (Take 4)

So now when the hacker does:

[http://secure-bank.com:8686/mini_pres_app/vanilla?
report_id=A6FE25423](http://secure-bank.com:8686/mini_pres_app/vanilla?report_id=A6FE25423)****

Application does:

```
<h2>Viewing Report</h2>  
Client: Mr Jones  
Report ID: A6FE25423img src=1 onerror=alert("evil") [No tag brackets]
```

We Win: Ha ha: Now you cannot create any HTML tags at all!

Job Done

No more XSS for me :)

News just in...

A HACKER HAS JUST EXPLOITED XSS ON OUR APPLICATION

- But I stripped out *tag brackets* from *all* user input!!!

How did it happen?

There is another part of my application that reflects input like this:

http://secure-bank.com:8686/mini_pres_app/attr_value?report_id=A6FE25423

```
<h2 id='A6FE25423'>Viewing Report</h2>
```

```
Client: Mr Jones
```

```
Report ID: A6FE25423
```

How did it happen?

The hacker found this, then created a yet stranger payload like this:

[http://secure-bank.com:8686/mini_pres_app/attr_value?
report_id=A6FE25423%27+onmouseover=alert\(%27evil%27\)+foo=%27](http://secure-bank.com:8686/mini_pres_app/attr_value?report_id=A6FE25423%27+onmouseover=alert(%27evil%27)+foo=%27)

```
<h2 id='A6FE25423' onmouseover=alert('evil') foo=''>Viewing Report</h2>  
Client: Mr Jones  
Report ID: A6FE25423' onmouseover=alert('evil') foo='
```

We Lose: In this context they *didn't even need to add a new tag* to get execution: they just *added an event attribute* to an *existing tag*.

Thwarting the Attacker

- Okay, Wise Guy, I can see what you are doing here, and I know just what to do:
- I will strip out *quotes*
- ... and *double quotes*, too, so you don't try any more tricks like that

Thwarting the Attacker

[http://secure-bank.com:8686/mini_pres_app/attr_value?
report_id=A6FE25423%27+onmouseover=alert\(%27evil%27\)+foo=%27](http://secure-bank.com:8686/mini_pres_app/attr_value?report_id=A6FE25423%27+onmouseover=alert(%27evil%27)+foo=%27)

```
<h2 id='A6FE25423 onmouseover=alert(evil) foo='>Viewing Report</h2>
```

```
Client: Mr Jones
```

```
Report ID: A6FE25423 onmouseover=alert(evil) foo=
```

We Win: Ha ha: With your quotes stripped out, you *cannot break out* of the quoted attribute value *to add a script executing attribute*

Home Free

Definitely no more XSS for me now :)

News just in...

A HACKER HAS JUST EXPLOITED XSS ON OUR APPLICATION

- But I stripped out *tag brackets* and *quotes!!!*
- There must be some mistake... surely?

How did it happen?

There is another part of my application that reflects input within a *script* tag like this:

http://secure-bank.com:8686/mini_pres_app/script_unquoted?report_id=1236564

```
<h2>Viewing Report</h2>
Client: Mr Jones
Report ID: <span id=id_placeholder>None</span>
<script>
  document.getElementById('id_placeholder').innerText = 1236564
</script>
```

How did it happen?

The hacker found this, then did this:

[http://secure-bank.com:8686/mini_pres_app/script_unquoted?report_id=1236564-alert\(1\)](http://secure-bank.com:8686/mini_pres_app/script_unquoted?report_id=1236564-alert(1))

```
Report ID: <span id=id_placeholder>None</span>
<script>
  document.getElementById('id_placeholder').innerText = 1236564-alert(1)
</script>
```

We Lose: The devils! They found a place in a *script* where I reflected input *without quotes* - so they didn't even need to use a quote (I'd have stripped) to escape.

Thwarting the Attacker

- Okay, to reward your tenacity, Mr Hacker, I'm going to...
- ... trap all of my input *between quotes*

Thwarting the Attacker

Now when the hacker does this:

[http://secure-bank.com:8686/mini_pres_app/script_unquoted?report_id=1236564-alert\(1\)](http://secure-bank.com:8686/mini_pres_app/script_unquoted?report_id=1236564-alert(1))

```
Report ID: <span id=id_placeholder>None</span>
<script>
  document.getElementById('id_placeholder').innerText = '1236564-alert(1)'
</script>
```

We Win: Have that, Mr Hacker! Your attack is now trapped as a simple string and can no longer execute.

Hang up Your Hat, Mr Hacker

Definitely no more XSS for me now :)

News just in...

A HACKER HAS JUST EXPLOITED XSS ON OUR APPLICATION

- But I stripped out *tag brackets* and *quotes!!!*
- ... and I ensured *all* input was *trapped between quotes*
- This is becoming... te... di... ous :(

How did it happen?

There is another part of my application that reflects input like this:

http://secure-bank.com:8686/mini_pres_app/event_attr?report_id=A6FE25423

```
<h2>Viewing Report</h2>
```

```
Client: Mr Jones
```

```
Report link: <a onclick="log('Report opened:' + 'A6FE25423')">link</a>
```

How did it happen?

The hacker found this, then did this:

[http://secure-bank.com:8686/mini_pres_app/event_attr?report_id=A6FE25423%26apos%3b-alert\(1\)-%26apos%3b](http://secure-bank.com:8686/mini_pres_app/event_attr?report_id=A6FE25423%26apos%3b-alert(1)-%26apos%3b)

```
<h2>Viewing Report</h2>
Report link: <a onclick="log(
  'Report opened:' + 'A6FE25423&apos;-alert(1)-&apos;'
)">link</a>
```

We Lose: ... but it is *not clear why* from this, since it seems they did not break out of the ' quotes, so how on Earth did it manage to execute???

How did it happen?

We need to understand what *HTML entity encoding* is
to understand how this attack succeeded

So What is HTML Entity Encoding?

- Usually, when we have a *text-based structure* like an `HTML` page...
- ... there is a way of *escaping* special characters so they can be displayed *literally*
- ... and so without being interpreted dangerously by the browser as part of the wider `HTML` structure

HTML Entity Encoding

- So encodings of key characters are encoded like this (with many other possible representations):
 - `<` becomes `<`;
 - `>` becomes `>`;
 - `'` becomes `'`;
 - `"` becomes `"`;
 - `&` becomes `&`;

So how did it happen?

Subtle insight: When the browser parses *tag attributes*,
it *automatically HTML decodes their values*

So how did it happen?

So this:

```
Report link: <a onclick="log(
  'Report opened:' + 'A6FE25423&apos;-alert(1)-&apos;'
)">link</a>
```

... gets interpreted by the browser as this:

```
Report link: <a onclick="log(
  'Report opened:' + 'A6FE25423'-alert(1)-''
)">link</a>
```

So the hacker *bypassed my quote stripping* by *encoding the quote character* in a way I didn't anticipate

Thwarting the Attacker

Two can play this game: let's also use **HTML** encoding — but as a defence — by **HTML** encoding the input

[http://secure-bank.com:8686/mini_pres_app/event_attr?report_id=A6FE25423%26apos%3b-alert\(1\)-%26apos%3b](http://secure-bank.com:8686/mini_pres_app/event_attr?report_id=A6FE25423%26apos%3b-alert(1)-%26apos%3b)

```
Report link: <a onclick="log(
  'Report opened:' + 'A6FE25423&apos;-alert(1)-&apos;'
)">link</a>
```

We Win: In your face, Mr Hacker. When they attempt the bypass, their input ends up *double HTML encoded*, so the attack fails.

I'm Riding on a High, Mr Hacker

I have drunk deeply from the fountain of XSS knowledge and now you must stand aside, My Friend.

News just in...

A HACKER HAS JUST EXPLOITED XSS ON OUR APPLICATION

- But I ensured all input was *trapped between quotes*
- I *stripped out quote characters*
- and then I *HTML entity encoded* all input
- WHAT MORE CAN I DO!!

WHAT MORE CAN I DO!!!!!!

How did it happen?

There is another part of my application that reflects input like this:

http://secure-bank.com:8686/mini_pres_app/href_attr?report_id=A6FE25423

```
<h2>Viewing Report</h2>  
Report ID: A6FE25423  
<a href='A6FE25423/edit'>Edit report</a>
```

How did it happen?

The hacker found this, then did this:

[http://secure-bank.com:8686/mini_pres_app/href_attr?
report_id=javascript:alert\(1\)/](http://secure-bank.com:8686/mini_pres_app/href_attr?report_id=javascript:alert(1)/)

```
<h2>Viewing Report</h2>  
Report ID: javascript:alert(1)/  
<a href='javascript:alert(1)//edit'>Edit report</a>
```

We Lose: What is THAT thing? Their rather elegant payload did not rely on any *quote breakout* and was *unaffected by HTML encoding*, etc.!!

How did it happen?

So it turns out that attributes which take URLs can actually run scripts when prefixed with a special *scheme*

- `javascript:` will run *JavaScript*
- `vbscript:` will run *visual basic* (MS browsers)
- `livescript:` will run a prehistoric relic of JavaScript (old, old, Netscape — Do not worry about this)

Thwarting the Attacker (Take 1)

- Right let's *block anything* that *starts with* one of those *script schemes*
- And we are not going to fall for the exact-case trick again: we will block these *whatever the case*

Thwarting the Attacker (Take 1)

So when the hacker tries this:

[http://secure-bank.com:8686/mini_pres_app/href_attr?
report_id=jaVascRiPt:alert\(1\)/](http://secure-bank.com:8686/mini_pres_app/href_attr?report_id=jaVascRiPt:alert(1)/)

```
<h2>Viewing Report</h2>  
Report ID: BLOCKED  
<a href='BLOCKED/edit'>Edit report</a>
```

We Win: The script prefix is *blocked whatever the case.*

Thwarting the Attacker (Take 1)

Then the hacker does this:

```
http://secure-bank.com:8686/mini_pres_app/href_attr?  
report_id=%20jaVascRiPt:alert(1)/
```

```
<h2>Viewing Report</h2>  
Report ID: javascript:alert(1)/  
<a href='_javascript:alert(1)//edit'>Edit report</a>
```

We Lose: Doh! The swines just *added a space* in front of `javascript:` so my *match-at-the-start-of-the-input* would fail. Browsers are *unhelpfully lenient* in parsing things like this

Thwarting the Attacker (Take 2)

- Right let's *block anything* that simply *contains* one of those *script schemes*
- ... *whatever the case*

`http://secure-bank.com:8686/mini_pres_app/href_attr?
report_id=%20jaVascRiPt:alert(1)/`

```
<h2>Viewing Report</h2>  
Report ID: BLOCKED  
<a href='BLOCKED/edit'>Edit report</a>
```

We Win: Their *space trick* is now also blocked by our stuff.

Thwarting the Attacker (Take 2)

Then the hacker does this:

```
http://secure-bank.com:8686/mini_pres_app/href_attr?  
report_id=java%0ascript:alert(1)/
```

```
<h2>Viewing Report</h2>  
Report ID: java  
script:alert(1)/  
<a href='java <-- sneaky line break  
script:alert(1)//edit'>Edit report</a>
```

We Lose: You've got to be kidding me! They bypassed my check for `javascript` by sticking a line-break character in the middle *which the browser completely ignores and executes regardless!!*

Thwarting the Attacker (Take 3)

- Okay, we have to be really careful when using input with URL attributes like this
- ... especially if they reflect the *start of the URL* like here.

Thwarting the Attacker (Take 3)

- In this case I am going to opt for a strong white listing of only the *alpha-numeric* characters that are required for a report ID

[http://secure-bank.com:8686/mini_pres_app/href_attr?
report_id=%20jaVa%0AscRiPt>alert\(1\)/](http://secure-bank.com:8686/mini_pres_app/href_attr?report_id=%20jaVa%0AscRiPt>alert(1)/)

```
<h2>Viewing Report</h2>  
Report ID: BLOCKED  
<a href='BLOCKED/edit'>Edit report</a>
```

We Win: This blocks any of that weird stuff yet will allow the application to work as intended.

So Long, and thanks for all the Fish, Mr Hacker

The hour is late, and my job here is done.

News just in...

A HACKER HAS JUST EXPLOITED XSS ON OUR APPLICATION

- But I ensured all input was *trapped between quotes*
- I *stripped out quote characters*
- ...

News just in...

A HACKER HAS JUST EXPLOITED XSS ON OUR APPLICATION

- and I *HTML entity encoded* all input
- and I locked down the *harder places* with *whitelists*
- This cannot be happening to me — it must be a *stress-induced nightmare*
- Yes, I'm going to wake up soon... please... *PLEASE!!*

How did it happen?

The hacker did this:

`http://secure-bank.com:8686/mini_pres_app/dom_query_param?report_id=A6FE25423<script>alert('evil')</script>`



We Lose: But how? This is not possible, because we have already *added appropriate protection* for this crude *script kiddie* payload — it was one of the first things we tackled!

How did it happen?

The page just exploited looks like this:

[http://secure-bank.com:8686/mini_pres_app/dom_query_param?
report_id=A6FE25423](http://secure-bank.com:8686/mini_pres_app/dom_query_param?report_id=A6FE25423)

```
<h2>Viewing Report</h2>
Report ID: <span id=report_id></span>
<script>
var report_id = new URLSearchParams(window.location.search).get("report_id")
$('#report_id').html(report_id)
</script>
```

Hmmmm: But I don't see the input value, **A6FE25423**, reflected anywhere in this page??

How did it happen?

Ah, wait a minute: I see what this code is doing... it is a different beast altogether

```
<script>
var report_id = new URLSearchParams(window.location.search).get("report_id")
$('#report_id').html(report_id)
</script>
```

How did it happen?

```
<script>
var report_id = new URLSearchParams(window.location.search).get("report_id")
$('#report_id').html(report_id)
</script>
```

- So my web server *does not return* a page with the value reflected in it
- ... but it *does* return a *script* that runs when the page loads

How did it happen?

```
<script>
var report_id = new URLSearchParams(window.location.search).get("report_id")
$('#report_id').html(report_id)
</script>
```

- ... and this *script* then reads the URL parameter, `URLSearchParams(window.location.search).get("report_id")`
- ... then updates a placeholder element with the value, `$('#report_id').html(report_id)`

How did it happen?

- So there *is* a reflection of the user input, but it happens *only on the client-side* (i.e in the browser)
- The web server *never touches* the value, so here all my efforts of *server-side validation* are completely *bypassed*

Introducing: DOM-based XSS

- So just as we have seen lots of examples so far of building unsafe HTML on the server...
- ... it is possible also for a *script* to *dynamically update* the page you are viewing *in an unsafe way*

Introducing: DOM-based XSS

- This vector is known as *DOM-based XSS*, since it occurs *during execution* of a *script* ...
- which directly updates the *Document Object Model* (DOM) (i.e. the parsed HTML page that you view after a page loads)

Thwarting the Attacker

- The main problem here is that we use an *unsafe* method, `html(...)`, to change the *DOM*...
- ... such that raw *HTML* supplied by the hacker will be parsed and rendered as *HTML*

```
var report_id = new URLSearchParams(window.location.search).get("report_id")
$('#report_id').html(report_id)
```

- It is common to see stuff like this when the possibility of malicious input has been *completely overlooked*

Thwarting the Attacker

- Clearly it was not intended for a report ID to ever express a piece of **HTML**
- Let's instead use `text(...)`, which sets the contents of the placeholder tag as text... just *plain text*

```
var report_id = new URLSearchParams(window.location.search).get("report_id")
$('#report_id').text(report_id)
```

Thwarting the Attacker

So now when the hacker does this:

[http://secure-bank.com:8686/mini_pres_app/dom_query_param?
report_id=A6FE25423<script>alert\('evil'\)</script>](http://secure-bank.com:8686/mini_pres_app/dom_query_param?report_id=A6FE25423<script>alert('evil')</script>)

- Their payload is simply (and shamefully) displayed, and does not execute:



We Win

So XSS is Hard, Right?

... and deadly

No wonder it gets *everywhere*

So XSS is Hard, Right?

It is worth *trying* to understand the intricacies of *XSS* as much as possible, if you do not want to fall victim to it

Some general guidelines

- In whatever context, *trap input between quotes* and use context-appropriate escaping/encoding stop a payload breaking out of those quotes.
- Otherwise, employ very tight value/character white listing

Some general guidelines

- Remember: Tag attribute values are **HTML** decoded as the browser parses the **HTML**, so hackers will try to use this fact to bypass validation

Some general guidelines

- Be really careful when using input in *URL contexts*, especially to avoid attacks which use the `javascript:` scheme

Some general guidelines

- Don't neglect *client side* handling of input, to make sure it cannot be written to the *DOM* in an unsafe way

Some general guidelines

- Think twice before mixing user input with raw *JavaScript* evaluation functions such as: `eval(...)`, `setTimeout(...)`, etc.

Mitigation (and its Limitations)

There are three main mitigation approaches for XSS

(Though avoidance of XSS is the best mitigation)

Mitigation (and its Limitations)

HttpOnly Cookies

- These stop a direct hijack of a user's session by *hiding session tokens* (in cookies) from *scripts*
- Though making it harder, an attacker will usually find an *application specific* way to steal the user's account via XSS, such as by changing their email address then triggering a password reset
- It is a good general rule to set any sensitive cookies as `HttpOnly` unless the application requires it to be otherwise

Mitigation (and its Limitations)

Browser XSS Blocking

- Any scripts *observed to be reflected* from the request are *blocked* from executing by the browser
- However, it is an ongoing game of catch-up between the blocking algorithm and new techniques which bypass it
- It is effective only for *Reflected XSS*, so not *Persistent or DOM-based XSS*
- In fact, Microsoft has announced it will soon *retire XSS blocking* in *Edge*, given the little gain it brings

Mitigation (and its Limitations)

Content Security Policy (CSP):

- *Supporting* browsers allow the web application to *lock down sources of JavaScript execution*
- So, a CSP policy might allow scripts to execute *only from a reduced set of source URLs*
- And may *block* the use of *inline javascript*, such as *event handlers* and *script* tags
- In practice, this may require a substantial re-writing of application code, since violations may stop the application from functioning correctly.

Further Reading

(for Techs, or to pass on to your Techs)

- Must read:
[https://www.owasp.org/index.php/XSS_\(Cross_Site_Scripting\)_Prevention_Cheat_Sheet](https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet)
- Great collection of security related browser quirks:
<https://code.google.com/archive/p/browsersec/wikis/Main.wiki>

We Looked at...

- XSS
- XSS
- More XSS
- Then a little more XSS still

Thank You

I hope that was useful

Questions?