Cross-Site Scripting (XSS)

Web Application Security, shown in OWASP Juice Shop

Bernard Nongpoh

Assistant Professor, Department of CSE



ভাৰতীয় প্ৰযুক্তিবিদ্যা প্ৰতিষ্ঠান গুৱাহাটী
भारतीय प्रौद्योगिकी संस्थान गुवाहाटी
INDIAN INSTITUTE OF TECHNOLOGY GUWAHATI

What Is XSS? (1/4)

Cross-Site Scripting: an attacker's script runs in *another user's* browser.



What Is XSS? (2/4)

Cross-Site Scripting: an attacker's script runs in *another user's* browser.



What Is XSS? (3/4)

Cross-Site Scripting: an attacker's script runs in *another user's* browser.



What Is XSS? (4/4)

Cross-Site Scripting: an attacker's script runs in *another user's* browser.



The browser trusts the page's code. The attacker **slipped code into the page**, so it runs with the victim's **cookies and session**.

Why Attackers Need XSS (1/3)

The same origin policy stops `attacker.com` from reading `bank.com`.



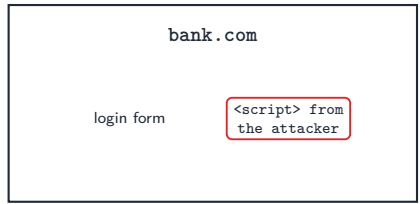
blocked: different origin

Why Attackers Need XSS (2/3)

The same origin policy stops `attacker.com` from reading `bank.com`.



blocked: different origin



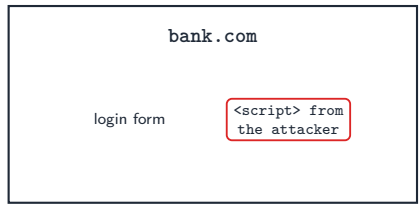
runs *as* bank.com

Why Attackers Need XSS (3/3)

The same origin policy stops `attacker.com` from reading `bank.com`.



blocked: different origin



runs *as* bank.com

XSS = get JavaScript to run **inside the target's own page**, with its origin.

XSS Is a Code Injection Vulnerability (1/4)

Untrusted data unexpectedly becomes code. Sounds familiar?

	SQL injection (S3)
the command	an SQL query
the interpreter	the database
the unexpected code	extra SQL

XSS Is a Code Injection Vulnerability (2/4)

Untrusted data unexpectedly becomes code. Sounds familiar?

	SQL injection (S3)	**cross-site scripting**
the command	an SQL query	an HTML page
the interpreter	the database	the **browser**
the unexpected code	extra SQL	**JavaScript**

XSS Is a Code Injection Vulnerability (3/4)

Untrusted data unexpectedly becomes code. Sounds familiar?

	SQL injection (S3)	**cross-site scripting**
the command	an SQL query	an HTML page
the interpreter	the database	the **browser**
the unexpected code	extra SQL	**JavaScript**
runs with	the database account	the **user's session**

XSS Is a Code Injection Vulnerability (4/4)

Untrusted data unexpectedly becomes code. Sounds familiar?

	SQL injection (S3)	**cross-site scripting**
the command	an SQL query	an HTML page
the interpreter	the database	the **browser**
the unexpected code	extra SQL	**JavaScript**
runs with	the database account	the **user's session**

The attacker can do **anything the user can do** in that site: read cookies, send requests with them.

Session Hijacking with XSS (1/4)

What can the injected script do with the user's session?

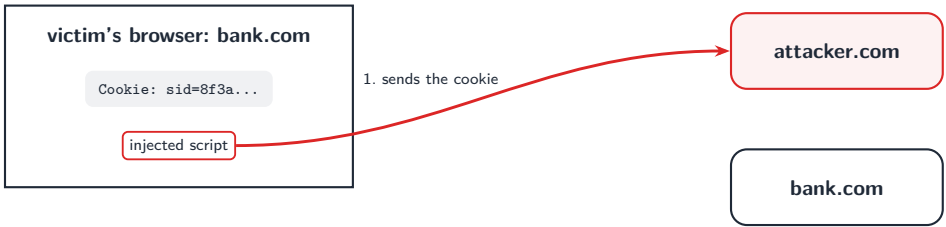




Stanford CS253, L06 Cross Site Scripting; RFC 6265: HTTP State Management Mechanism

Session Hijacking with XSS (2/4)

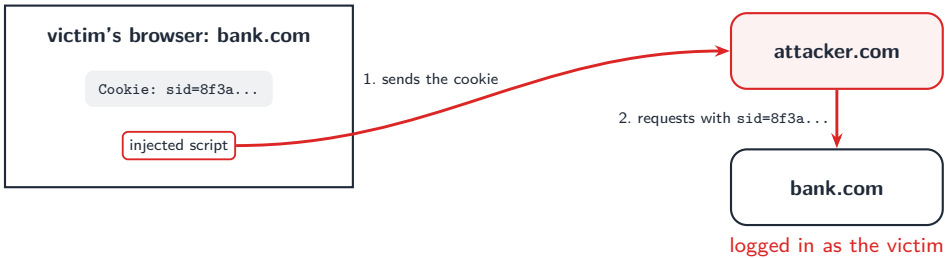
What can the injected script do with the user's session?



Stanford CS253, L06 Cross Site Scripting; RFC 6265: HTTP State Management Mechanism

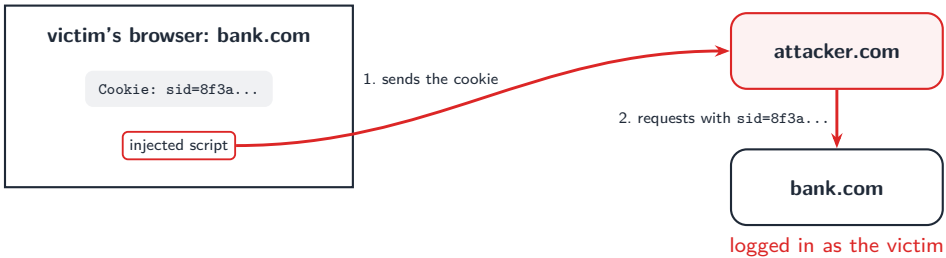
Session Hijacking with XSS (3/4)

What can the injected script do with the user's session?



Session Hijacking with XSS (4/4)

What can the injected script do with the user's session?



Defences next session: **HttpOnly** cookies, escaping, Content Security Policy.

XSS in Juice Shop (1/5)



Our target for today: the **XSS** category of the score board.

stars	challenge	kind	where
★	DOM XSS	DOM	product search
★	Bonus Payload	DOM	product search
★★	Reflected XSS	reflected	order tracking
★★★	API-only XSS	stored	product description
★★★	Client-side XSS Protection	stored	past a client-side check
★★★★	HTTP-Header XSS	stored	an HTTP header
★★★★	Server-side XSS Protection	stored	past a server-side check
★★★★	CSP Bypass	stored	a legacy page
★★★★★★	Video XSS	stored	the promo video

Juice Shop: challenges.yml, category XSS

XSS in Juice Shop (2/5)



Our target for today: the **XSS** category of the score board.

stars	challenge	kind	where
★	**DOM XSS**	**DOM**	**product search**
★	Bonus Payload	DOM	product search
★★	Reflected XSS	reflected	order tracking
★★★	API-only XSS	stored	product description
★★★	Client-side XSS Protection	stored	past a client-side check
★★★★	HTTP-Header XSS	stored	an HTTP header
★★★★	Server-side XSS Protection	stored	past a server-side check
★★★★	CSP Bypass	stored	a legacy page
★★★★★★	Video XSS	stored	the promo video

Juice Shop: challenges.yml, category XSS

XSS in Juice Shop (3/5)



Our target for today: the **XSS** category of the score board.

stars	challenge	kind	where
⋆	DOM XSS	DOM	product search
⋆	Bonus Payload	DOM	product search
⋆⋆	**Reflected XSS**	**reflected**	**order tracking**
⋆⋆⋆	API-only XSS	stored	product description
⋆⋆⋆	Client-side XSS Protection	stored	past a client-side check
⋆⋆⋆⋆	HTTP-Header XSS	stored	an HTTP header
⋆⋆⋆⋆	Server-side XSS Protection	stored	past a server-side check
⋆⋆⋆⋆	CSP Bypass	stored	a legacy page
⋆⋆⋆⋆⋆⋆	Video XSS	stored	the promo video

Juice Shop: challenges.yml, category XSS

XSS in Juice Shop (4/5)



Our target for today: the **XSS** category of the score board.

stars	challenge	kind	where
⋆	DOM XSS	DOM	product search
⋆	Bonus Payload	DOM	product search
⋆⋆	Reflected XSS	reflected	order tracking
⋆⋆⋆	**API-only XSS**	**stored**	**product description**
⋆⋆⋆	Client-side XSS Protection	stored	past a client-side check
⋆⋆⋆⋆	HTTP-Header XSS	stored	an HTTP header
⋆⋆⋆⋆	Server-side XSS Protection	stored	past a server-side check
⋆⋆⋆⋆	CSP Bypass	stored	a legacy page
⋆⋆⋆⋆⋆⋆	Video XSS	stored	the promo video

Juice Shop: challenges.yml, category XSS

XSS in Juice Shop (5/5)



Our target for today: the **XSS** category of the score board.

stars	challenge	kind	where
⋆	DOM XSS	DOM	product search
⋆	Bonus Payload	DOM	product search
⋆⋆	Reflected XSS	reflected	order tracking
⋆⋆⋆	API-only XSS	stored	product description
⋆⋆⋆	Client-side XSS Protection	stored	past a client-side check
⋆⋆⋆⋆	HTTP-Header XSS	stored	an HTTP header
⋆⋆⋆⋆	Server-side XSS Protection	stored	past a server-side check
⋆⋆⋆⋆	CSP Bypass	stored	a legacy page
⋆⋆⋆⋆⋆⋆	Video XSS	stored	the promo video

Most ask for the same harmless proof: `alert(‘xss‘)` must run.

Juice Shop: challenges.yml, category XSS

Three Kinds of XSS (1/3)



Where does the code come from, and who puts it into the page?

reflected: the server copies the request into the page



Juice Shop: order tracking

Stanford CS253, L06 Cross Site Scripting; OWASP: DOM Based XSS

Three Kinds of XSS (2/3)



Where does the code come from, and who puts it into the page?

reflected: the server copies the request into the page



Juice Shop: order tracking

stored: the code waits in the database



Juice Shop: product description

Stanford CS253, L06 Cross Site Scripting; OWASP: DOM Based XSS

Three Kinds of XSS (3/3)



Where does the code come from, and who puts it into the page?

reflected: the server copies the request into the page



Juice Shop: order tracking

stored: the code waits in the database



Juice Shop: product description

DOM: the page's own JavaScript writes the URL into the page

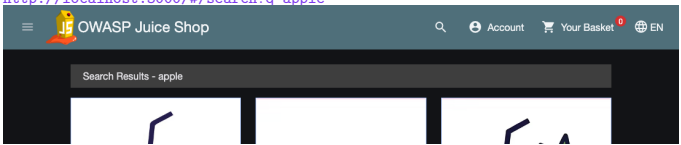


Juice Shop: product search

the server may never see the code

Stanford CS253, L06 Cross Site Scripting; OWASP: DOM Based XSS

DOM XSS in Juice Shop's Search (1/4)

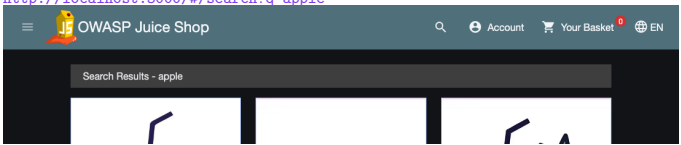


Juice Shop v20.4 on 127.0.0.1

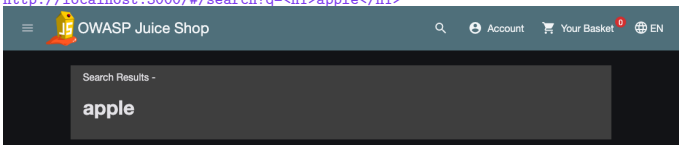
DOM XSS in Juice Shop's Search (2/4)



http://localhost:3000/#/search?q=apple



http://localhost:3000/#/search?q=<h1>apple</h1>

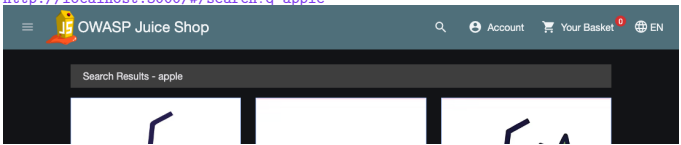


Juice Shop v20.4 on 127.0.0.1

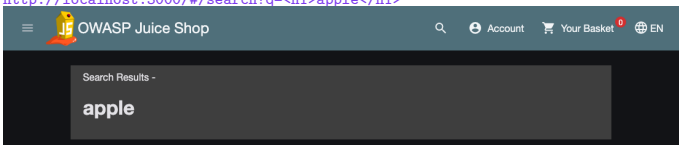
DOM XSS in Juice Shop's Search (3/4)



http://localhost:3000/#/search?q=apple



http://localhost:3000/#/search?q=<h1>apple</h1>



Juice Shop search

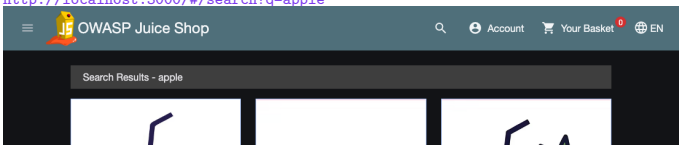
```
q = apple
  #searchValue innerHTML:
     apple
q = <h1>apple</h1>
  #searchValue innerHTML: <h1
     >apple</h1>
```

Juice Shop v20.4 on 127.0.0.1

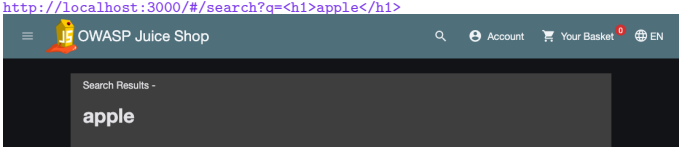
DOM XSS in Juice Shop's Search (4/4)



http://localhost:3000/#/search?q=apple



http://localhost:3000/#/search?q=<h1>apple</h1>



Juice Shop search

```
q = apple
  #searchValue innerHTML:
     apple
q = <h1>apple</h1>
  #searchValue innerHTML: <h1
     >apple</h1>
```

The search text is **parsed as HTML**. The challenge: do the same with Juice Shop's `<iframe src="javascript:...">`.

Juice Shop v20.4 on 127.0.0.1

Where the Search Goes Wrong (1/4)



search-result.component.html, Juice Shop (Angular)

```
11 <span id="searchValue" [innerHTML]="searchValue"></span>
```



Juice Shop: search-result.component.ts, challenge *DOM XSS*; Angular: Security

Where the Search Goes Wrong (2/4)



search-result.component.html, Juice Shop (Angular)

```
11 <span id="searchValue" [innerHTML]="searchValue"></span>
```

search-result.component.ts

```
145 this.dataSource.filter = queryParam.toLowerCase()
146 this.searchValue = this.sanitizer.bypassSecurityTrustHtml(queryParam)
```



Juice Shop: search-result.component.ts, challenge *DOM XSS*; Angular: Security

Where the Search Goes Wrong (3/4)

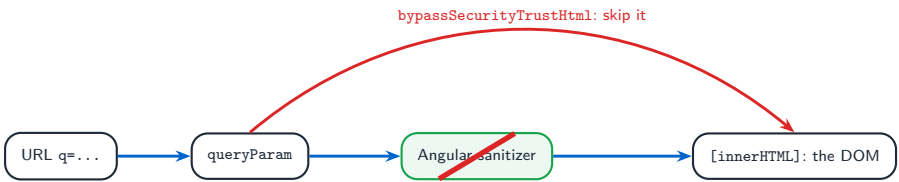


search-result.component.html, Juice Shop (Angular)

```
11 <span id="searchValue" [innerHTML]="searchValue"></span>
```

search-result.component.ts

```
145 this.dataSource.filter = queryParam.toLowerCase()
146 this.searchValue = this.sanitizer.bypassSecurityTrustHtml(queryParam)
```



Juice Shop: search-result.component.ts, challenge *DOM XSS*; Angular: Security

Where the Search Goes Wrong (4/4)

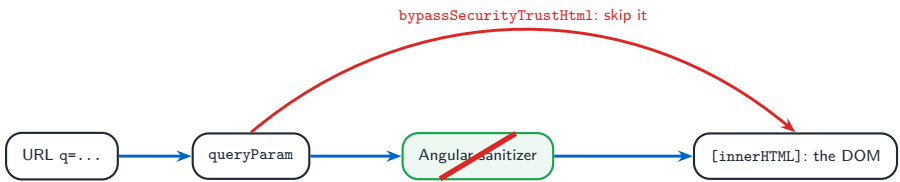


search-result.component.html, Juice Shop (Angular)

```
11 <span id="searchValue" [innerHTML]="searchValue"></span>
```

search-result.component.ts

```
145 this.dataSource.filter = queryParam.toLowerCase()
146 this.searchValue = this.sanitizer.bypassSecurityTrustHtml(queryParam)
```



Frameworks **escape by default**. XSS often comes from code that **switches it off**.

Juice Shop: search-result.component.ts, challenge *DOM XSS*; Angular: Security

Stored XSS: Product Descriptions (1/4)



The same component, another field. Who writes the descriptions?

search-result.component.ts, Juice Shop

```
111 trustProductDescription (tableData: any[]) {
112   for (let i = 0; i < tableData.length; i++) {
113     tableData[i].description = this.sanitizer.bypassSecurityTrustHtml(tableData[i].
        description)
114   }
```

Juice Shop: search-result.component.ts, challenge *API-only XSS*; codefix restfulXssChallenge

Stored XSS: Product Descriptions (2/4)



The same component, another field. Who writes the descriptions?

search-result.component.ts, Juice Shop

```
111 trustProductDescription (tableData: any[]) {
112   for (let i = 0; i < tableData.length; i++) {
113     tableData[i].description = this.sanitizer.bypassSecurityTrustHtml(tableData[i].
        description)
114   }
```

- The descriptions come from the **database**: shown to **every** visitor

Juice Shop: search-result.component.ts, challenge *API-only XSS*; codefix restfulXssChallenge

Stored XSS: Product Descriptions (3/4)



The same component, another field. Who writes the descriptions?

search-result.component.ts, Juice Shop

```
111 trustProductDescription (tableData: any[]) {
112   for (let i = 0; i < tableData.length; i++) {
113     tableData[i].description = this.sanitizer.bypassSecurityTrustHtml(tableData[i].
        description)
114   }
```

- The descriptions come from the **database**: shown to **every** visitor
- The REST API let users **change** them: **broken access control**

Juice Shop: search-result.component.ts, challenge *API-only XSS*; codefix restfulXssChallenge

Stored XSS: Product Descriptions (4/4)



The same component, another field. Who writes the descriptions?

search-result.component.ts, Juice Shop

```
111 trustProductDescription (tableData: any[]) {
112   for (let i = 0; i < tableData.length; i++) {
113     tableData[i].description = this.sanitizer.bypassSecurityTrustHtml(tableData[i].
        description)
114   }
```

- The descriptions come from the **database**: shown to **every** visitor
- The REST API let users **change** them: **broken access control**
- Juice Shop's fix: **remove the bypass**; and stop users editing products

Juice Shop: search-result.component.ts, challenge *API-only XSS*; codefix restfulXssChallenge

Reflected XSS: Order Tracking (1/4)



The server copies the order id from the URL into its answer.

routes/trackOrder.ts, Juice Shop

```
14 // Truncate id to avoid unintentional RCE
15 const id = !utils.isChallengeEnabled(challenges.reflectedXssChallenge) ? String(req.params
       .id).replace(/[^\w-]+/g, '') : utils.trunc(req.params.id, 60)
```

what happens to the id

Juice Shop: routes/trackOrder.ts, challenge *Reflected XSS*

Reflected XSS: Order Tracking (2/4)



The server copies the order id from the URL into its answer.

routes/trackOrder.ts, Juice Shop

```
14 // Truncate id to avoid unintentional RCE
15 const id = !utils.isChallengeEnabled(challenges.reflectedXssChallenge) ? String(req.params
       .id).replace(/[^\w-]+/g, '') : utils.trunc(req.params.id, 60)
```

	what happens to the id	
challenge **off**	`replace(/[^\w-]+/g, '')`: letters, digits, _, – only	**allow-list**

Juice Shop: routes/trackOrder.ts, challenge *Reflected XSS*

Reflected XSS: Order Tracking (3/4)



The server copies the order id from the URL into its answer.

routes/trackOrder.ts, Juice Shop

```
14 // Truncate id to avoid unintentional RCE
15 const id = !utils.isChallengeEnabled(challenges.reflectedXssChallenge) ? String(req.params
       .id).replace(/[^\w-]+/g, '') : utils.trunc(req.params.id, 60)
```

	what happens to the id	
challenge **off**	`replace(/[^\w-]+/g, '')`: letters, digits, _, – only	**allow-list**
challenge **on**	`trunc(id, 60)`: only shortened	**reflected**

Juice Shop: routes/trackOrder.ts, challenge *Reflected XSS*

Reflected XSS: Order Tracking (4/4)



The server copies the order id from the URL into its answer.

routes/trackOrder.ts, Juice Shop

```
14 // Truncate id to avoid unintentional RCE
15 const id = !utils.isChallengeEnabled(challenges.reflectedXssChallenge) ? String(req.params
       .id).replace(/[^\w-]+/g, '') : utils.trunc(req.params.id, 60)
```

	what happens to the id	
challenge **off**	`replace(/[^\w-]+/g, '')`: letters, digits, _, – only	**allow-list**
challenge **on**	`trunc(id, 60)`: only shortened	**reflected**

An order id has a known shape: **validate it**. Where text is free, **escape it**.

Juice Shop: routes/trackOrder.ts, challenge *Reflected XSS*

Quiz: Reflected, Stored or DOM? (1/4)

1. An error page prints the bad `?page=` value from the URL
2. A product review shows HTML from its author to every buyer
3. Page JavaScript writes `location.hash` into `innerHTML`

Stanford CS253, L06 Cross Site Scripting; OWASP: DOM Based XSS

Quiz: Reflected, Stored or DOM? (2/4)

1. An error page prints the bad `?page=` value from the URL — `reflected`
2. A product review shows HTML from its author to every buyer
3. Page JavaScript writes `location.hash` into `innerHTML`

Stanford CS253, L06 Cross Site Scripting; OWASP: DOM Based XSS

Quiz: Reflected, Stored or DOM? (3/4)

1. An error page prints the bad `?page=` value from the URL — `reflected`
2. A product review shows HTML from its author to every buyer — `stored`
3. Page JavaScript writes `location.hash` into `innerHTML` — ____

Stanford CS253, L06 Cross Site Scripting; OWASP: DOM Based XSS

Quiz: Reflected, Stored or DOM? (4/4)

1. An error page prints the bad `?page=` value from the URL — `reflected`
2. A product review shows HTML from its author to every buyer — `stored`
3. Page JavaScript writes `location.hash` into `innerHTML` — `DOM`

Stanford CS253, L06 Cross Site Scripting; OWASP: DOM Based XSS

Why Is XSS So Common? (1/3)

One page, several languages. Each has its own control characters.

```
<p>Hi DATA</p>
 <img alt='DATA'>
 <a href='DATA'>link</a>
 <div onclick='go(DATA)'>
 <script>let n = 'DATA'</script>
```

Stanford CS253, L06 Cross Site Scripting; OWASP XSS Prevention Cheat Sheet

Why Is XSS So Common? (2/3)

One page, several languages. Each has its own control characters.

```
<p>Hi DATA</p>
 <img alt='DATA'>
 <a href='DATA'>link</a>
 <div onclick='go(DATA)'>
 <script>let n = 'DATA'</script>
```

element: < &

attribute: ' " &

URL: the scheme itself

event handler: JavaScript, then HTML

script: ' \ and </script>

Stanford CS253, L06 Cross Site Scripting; OWASP XSS Prevention Cheat Sheet

Why Is XSS So Common? (3/3)

One page, several languages. Each has its own control characters.

```
<p>Hi DATA</p>
 <img alt='DATA'>
 <a href='DATA'>link</a>
 <div onclick='go(DATA)'>
 <script>let n = 'DATA'</script>
```

element: < &

attribute: ' " &

URL: the scheme itself

event handler: JavaScript, then HTML

script: ' \ and </script>

Slip up in **one** place, and the whole page is vulnerable.

Stanford CS253, L06 Cross Site Scripting; OWASP XSS Prevention Cheat Sheet

Attributes with Special Meanings (1/3)

Escaping keeps data in the attribute. But what does the attribute *do*?

what data: and javascript: URLs are for

```
<a href='javascript:alert("hi")'>Say hi</a>       run JavaScript on click
<img src='...' />  an image inside the page
```

Attributes with Special Meanings (2/3)

Escaping keeps data in the attribute. But what does the attribute *do*?

what data: and javascript: URLs are for

```
<a href='javascript:alert("hi")'>Say hi</a>       run JavaScript on click
<img src='...' />  an image inside the page
```

let the user choose a URL ...

```
<a href='USER_URL'>my homepage</a>          javascript: runs on click
<iframe src='USER_URL'></iframe>            data:text/html makes a page
<script src='USER_URL'></script>            never safe, even escaped
```

Attributes with Special Meanings (3/3)

Escaping keeps data in the attribute. But what does the attribute *do*?

what data: and javascript: URLs are for

```
<a href='javascript:alert("hi")'>Say hi</a>       run JavaScript on click
<img src='...' />  an image inside the page
```

let the user choose a URL ...

```
<a href='USER_URL'>my homepage</a>          javascript: runs on click
<iframe src='USER_URL'></iframe>            data:text/html makes a page
<script src='USER_URL'></script>            never safe, even escaped
```

the fix: an allow-list of schemes

```
22 def safe_url(u):                        # allow-list of URL schemes
23     return u if urlparse(u).scheme in ("http", "https") else "#"
```

Parsers, Parsers, Everywhere (1/4)

Data inside a <script>. Why is escaping quotes not enough?

```
<script>
  let username = '</script><script>alert(1)//'
</script>
```

the page as sent

Parsers, Parsers, Everywhere (2/4)

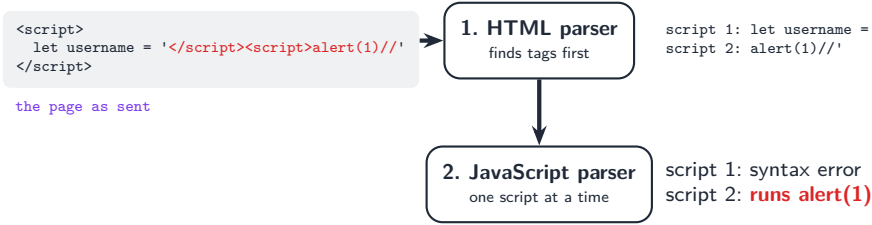
Data inside a <script>. Why is escaping quotes not enough?



the page as sent

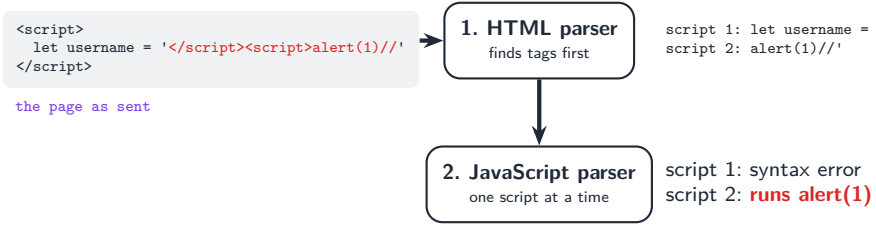
Parsers, Parsers, Everywhere (3/4)

Data inside a <script>. Why is escaping quotes not enough?



Parsers, Parsers, Everywhere (4/4)

Data inside a <script>. Why is escaping quotes not enough?



Fixes: **hex encode** the data and decode it in JavaScript, or put it in a <template> element.

Contexts Which Are Never Safe (1/3)

```
<script>USER_DATA_HERE</script>
<!-- USER_DATA_HERE -->
<USER_DATA_HERE href='/'>Link</USER_DATA_HERE>
<div USER_DATA_HERE='some value'></div>
<style>USER_DATA_HERE</style>
```

Contexts Which Are Never Safe (2/3)

```
<script>USER_DATA_HERE</script>
<!-- USER_DATA_HERE -->
<USER_DATA_HERE href='/'>Link</USER_DATA_HERE>
<div USER_DATA_HERE='some value'></div>
<style>USER_DATA_HERE</style>
```

HTML parsers are **lax**: they accept and repair broken markup.

Contexts Which Are Never Safe (3/3)

```
<script>USER_DATA_HERE</script>
<!-- USER_DATA_HERE -->
<USER_DATA_HERE href='/'>Link</USER_DATA_HERE>
<div USER_DATA_HERE='some value'></div>
<style>USER_DATA_HERE</style>
```

HTML parsers are **lax**: they accept and repair broken markup.

Postel's robustness principle: "be liberal in what you accept".
Good for compatibility, **bad for security**.

Where Escaped Data Is Safe (1/3)

context	safe with	
HTML element body	HTML escaping	**yes**
HTML attribute	HTML escaping, **in quotes**	**yes**
JavaScript string		**yes**

Stanford CS253, L06 Cross Site Scripting; OWASP XSS Prevention Cheat Sheet

Where Escaped Data Is Safe (2/3)

context	safe with	
HTML element body	HTML escaping	**yes**
HTML attribute	HTML escaping, **in quotes**	**yes**
JavaScript string	hex, or `<template>`	**yes**

Stanford CS253, L06 Cross Site Scripting; OWASP XSS Prevention Cheat Sheet

Where Escaped Data Is Safe (3/3)

context	safe with	
HTML element body	HTML escaping	**yes**
HTML attribute	HTML escaping, **in quotes**	**yes**
JavaScript string	hex, or `<template>`	**yes**
URL attribute	a scheme allow-list	**with care**
nested: `onclick="setTimeout(...)"`	JS twice, then HTML	**avoid**

Stanford CS253, L06 Cross Site Scripting; OWASP XSS Prevention Cheat Sheet

Quiz: Which Escaping? (1/5)

`DATA` comes from the user. What is needed?

1. `<td>DATA</td>`
2. `<input value=DATA>`
3. `<a href='DATA'>profile</a>`
4. `<script>var q = 'DATA'</script>`

Stanford CS253, L06 Cross Site Scripting; OWASP XSS Prevention Cheat Sheet

Quiz: Which Escaping? (2/5)

DATA comes from the user. What is needed?

1. `<td>DATA</td>` — escape < and &
2. `<input value=DATA>`
3. `<a href='DATA'>profile</a>`
4. `<script>var q = 'DATA'</script>`

Stanford CS253, L06 Cross Site Scripting; OWASP XSS Prevention Cheat Sheet

Quiz: Which Escaping? (3/5)

`DATA` comes from the user. What is needed?

1. `<td>DATA</td>`	escape < and &
2. `<input value=DATA>`	add quotes, then escape
3. `<a href='DATA'>profile</a>`	
4. `<script>var q = 'DATA'</script>`	

Stanford CS253, L06 Cross Site Scripting; OWASP XSS Prevention Cheat Sheet

Quiz: Which Escaping? (4/5)

DATA comes from the user. What is needed?

1. `<td>DATA</td>` — escape < and &
2. `<input value=DATA>` — add quotes, then escape
3. `<a href='DATA'>profile</a>` — http(s) allow-list, escape
4. `<script>var q = 'DATA'</script>`

Stanford CS253, L06 Cross Site Scripting; OWASP XSS Prevention Cheat Sheet

Quiz: Which Escaping? (5/5)

`DATA` comes from the user. What is needed?

1. `<td>DATA</td>`	escape < and &
2. `<input value=DATA>`	add quotes, then escape
3. `<a href='DATA'>profile</a>`	http(s) allow-list, escape
4. `<script>var q = 'DATA'</script>`	hex, or a <template>

Stanford CS253, L06 Cross Site Scripting; OWASP XSS Prevention Cheat Sheet

Hands-On: Juice Shop XSS (1/5)



On your own Juice Shop (`http://localhost:3000`), 25 minutes:

1. Search for `<h1>apple</h1>`. Inspect the element: text or HTML?

OWASP Juice Shop; Pwning OWASP Juice Shop

Hands-On: Juice Shop XSS (2/5)



On your own Juice Shop (`http://localhost:3000`), 25 minutes:

1. Search for `<h1>apple</h1>`. Inspect the element: text or HTML?
2. Solve **DOM XSS** with the input given in its description

OWASP Juice Shop; Pwning OWASP Juice Shop

Hands-On: Juice Shop XSS (3/5)



On your own Juice Shop (`http://localhost:3000`), 25 minutes:

1. Search for `<h1>apple</h1>`. Inspect the element: text or HTML?
2. Solve **DOM XSS** with the input given in its description
3. Open its code <>: mark line 146, pick the fix

OWASP Juice Shop; Pwning OWASP Juice Shop

Hands-On: Juice Shop XSS (4/5)



On your own Juice Shop (`http://localhost:3000`), 25 minutes:

1. Search for `<h1>apple</h1>`. Inspect the element: text or HTML?
2. Solve **DOM XSS** with the input given in its description
3. Open its code <>: mark line 146, pick the fix
4. Same for **Bonus Payload** and **API-only XSS**

OWASP Juice Shop; Pwning OWASP Juice Shop

Hands-On: Juice Shop XSS (5/5)



On your own Juice Shop (`http://localhost:3000`), 25 minutes:

1. Search for `<h1>apple</h1>`. Inspect the element: text or HTML?
2. Solve **DOM XSS** with the input given in its description
3. Open its code <>: mark line 146, pick the fix
4. Same for **Bonus Payload** and **API-only XSS**
5. Search the frontend code for `bypassSecurityTrust`: how many?

Only on your own copy, bound to 127.0.0.1. Never on a real site.

OWASP Juice Shop; Pwning OWASP Juice Shop

Summary (1/5)

- XSS: data becomes **JavaScript in the target's page**, with the user's session

Summary (2/5)

- XSS: data becomes **JavaScript in the target's page**, with the user's session
- **Reflected**, **stored**, **DOM**: who writes the code into the page
- Juice Shop: frameworks protect by default; **bypasses** cause the bugs

Summary (3/5)

- XSS: data becomes **JavaScript in the target's page**, with the user's session
- **Reflected**, **stored**, **DOM**: who writes the code into the page
- Juice Shop: frameworks protect by default; **bypasses** cause the bugs
- Every **context** has its own control characters and escaping

Summary (4/5)

- XSS: data becomes **JavaScript in the target's page**, with the user's session
- **Reflected**, **stored**, **DOM**: who writes the code into the page
- Juice Shop: frameworks protect by default; **bypasses** cause the bugs
- Every **context** has its own control characters and escaping
- Some contexts are **never safe**; avoid nested ones

Summary (5/5)

- XSS: data becomes **JavaScript in the target's page**, with the user's session
- **Reflected**, **stored**, **DOM**: who writes the code into the page
- Juice Shop: frameworks protect by default; **bypasses** cause the bugs
- Every **context** has its own control characters and escaping
- Some contexts are **never safe**; avoid nested ones