Web Attacks 101

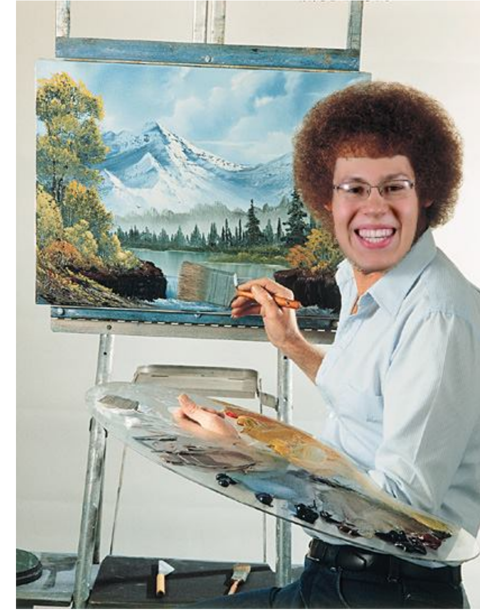
Dave Kukfa

Topics

- whoami
- Tools
- HTTP Basics
- Cross-Site Scripting (XSS)
- SQL Injection (SQLi)
- Resources

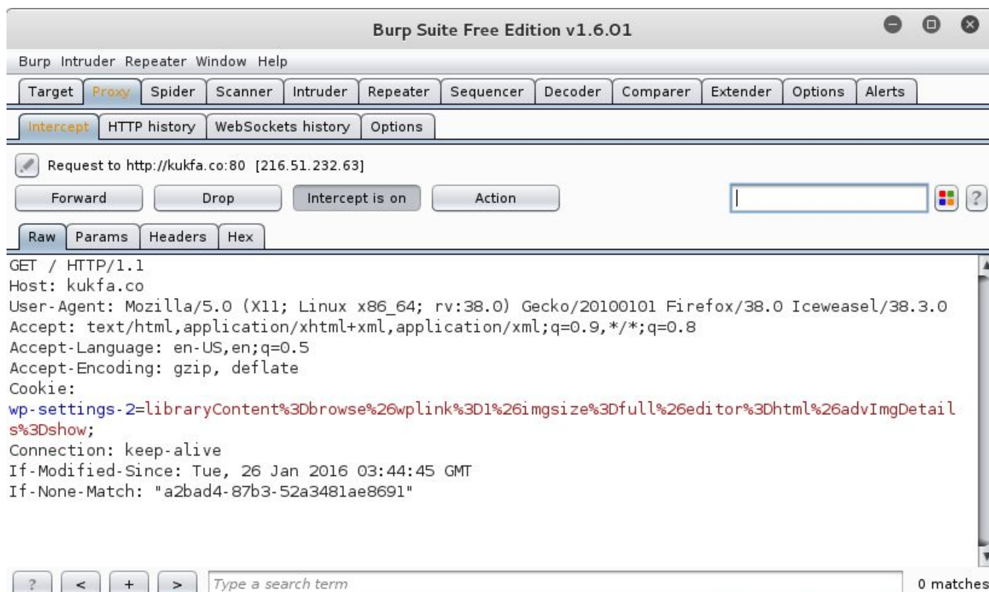
whoami

- Dave Kukfa
- 3rd year CSEC
- Web
- Pentesting
- Reversing
- http://kukfa.co



Tools

- Burp Suite
  - Your swiss army knife
  - Web proxy
    - MITM'ing traffic
    - Intercept requests/responses
    - Modify data
  - Many other useful features
- Scanners
  - Burp Suite Pro, OWASP ZAP
  - Enterprise solutions
  - Be careful



HTTP Basics

- Hypertext Transfer Protocol
  - Protocol used to access web sites
- Requests (think **client** -> **server**)
  - Methods (two most common)
    - GET
    - POST
- Responses (think **server** -> **client**)

HTTP GET

- Used to retrieve data
- Most common HTTP method
- What happens when I browse to http://kukfa.co?

```
GET / HTTP/1.1
Host: kukfa.co
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:38.0) Gecko/20100101 Firefox/38.0 Iceweasel/38.3.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

- This is a GET request captured in Burp Suite
- Notice the HTTP headers providing information to the web server

HTTP POST

- Used to perform actions
  - Logging in, submitting a form, upload a file, etc.

```
POST /wp-login.php HTTP/1.1
Host: kukfa.co
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:38.0) Gecko/20100101 Firefox/38.0 Iceweasel/38.3.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://kukfa.co/wp-login.php
Cookie: wordpress_test_cookie=WP+Cookie+check
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 98

log=dave&pwd=pass123&wp-submit=Log+In&redirect_to=http%3A%2F%2Fkukfa.co%2Fwp-admin%2F&testcookie=1
```

HTTP Query String

- ?user=dave&pass=pass123
  - ? denotes the start of the parameters
  - & splits parameters up
  - name=value format
- Query string sent in the URL of a GET request
  - http://website.com/login.php?user=dave&pass=pass123
  - The parameters can be seen in the URL
    - Shows up in history, bookmarks, etc.
  - GET should not be used when dealing with sensitive information
- Query string sent in the body of a POST request

Cookies

- Piece of data sent with each request
- Allows the web server to map HTTP requests to the user sending them
- This is how authentication works on web apps

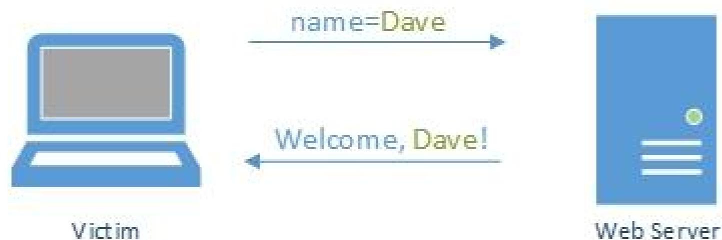
```
GET /845105168348/ HTTP/1.1
Host: google-gruyere.appspot.com
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:38.0) Gecko/20100101 Firefox/38.0 Iceweasel/38.3.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: https://google-gruyere.appspot.com/845105168348/login?uid=dave&pw=pass123
Cookie: GRUYERE=66539788|dave||author
Connection: keep-alive
```

Cross-Site Scripting (XSS)

- End result: attacker inputs malicious HTML/script that runs in the victim's browser
  - Steal user's session cookie, alter the page's appearance, exploit kit (pwn the victim's OS)...
- Attacker submits input to a vulnerable function on the website
  - Somehow, due to the way the function processes input, the user input is returned as part of the website's source code
  - The victim's browser processes the source code in order to display the web page
  - So if the user input is malicious code, the victim's browser will execute it
- Two most common forms
  - Reflected
  - Stored
- Attacks target the user

Consider an example page

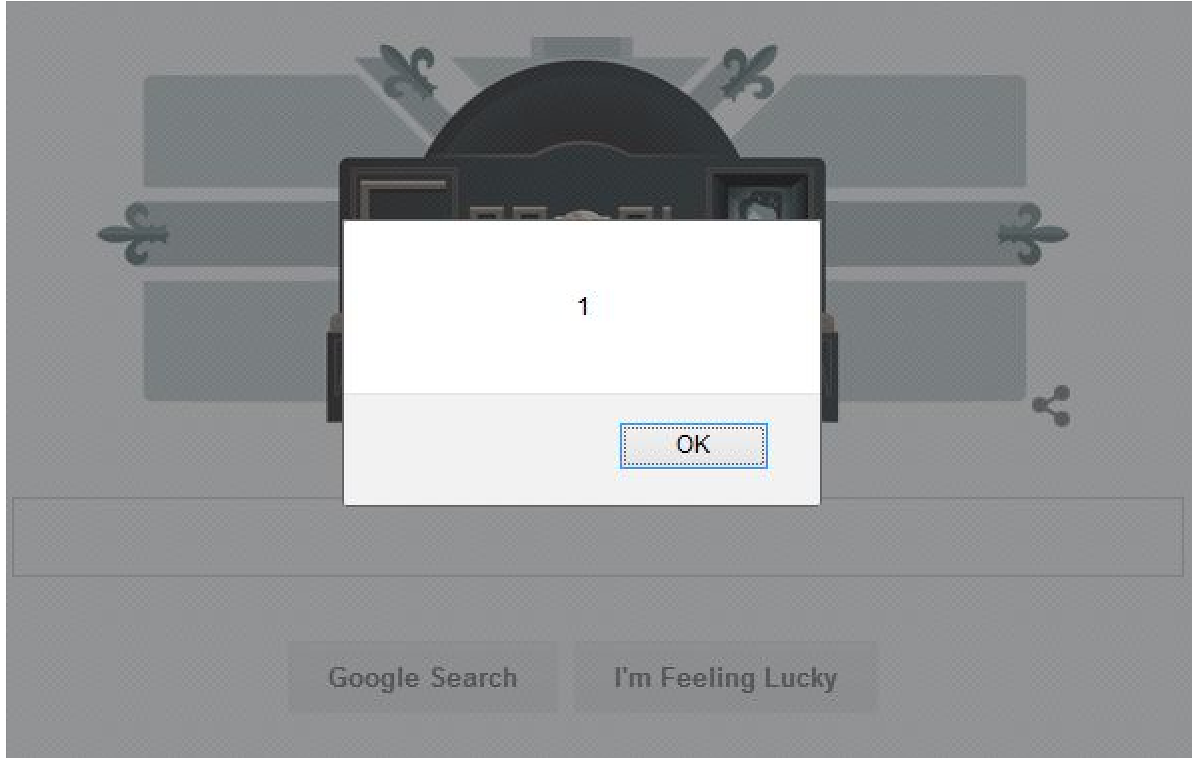
- Has a function where users enter their name, and receive a greeting
  - http://vulnerable.com/welcome.php?name=Dave
  - “Welcome to my website, Dave!”



Reflected XSS

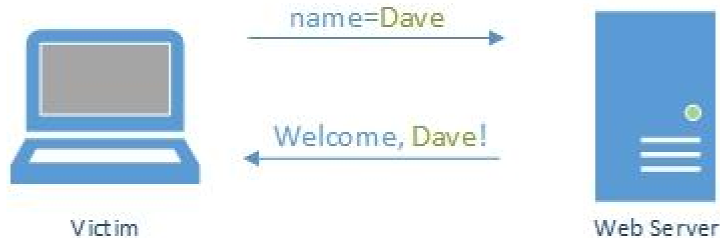
- Attacker has a payload (malicious code (s)he wants the user to run)
  - <script>alert(1)</script>
- The payload is sent in a request from the victim’s browser
  - Common scenario: the attacker gets the victim to click a malicious link
  - http://vulnerable.com/welcome.php?name=<script>alert(1)</script>
- The server processes the input, and issues a response that contains the payload
  - Let’s assume our welcome function processes user input without any checks
  - The input goes directly in the source code
  - “Welcome to my website, <script>alert(1)</script>!”
- The victim’s browser will execute the payload

Payload in action



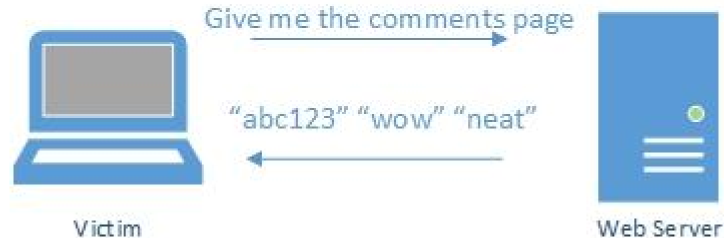
Why reflected?

- The payload is sent to the server before being returned to the user
- AKA, it is *reflected* off the server



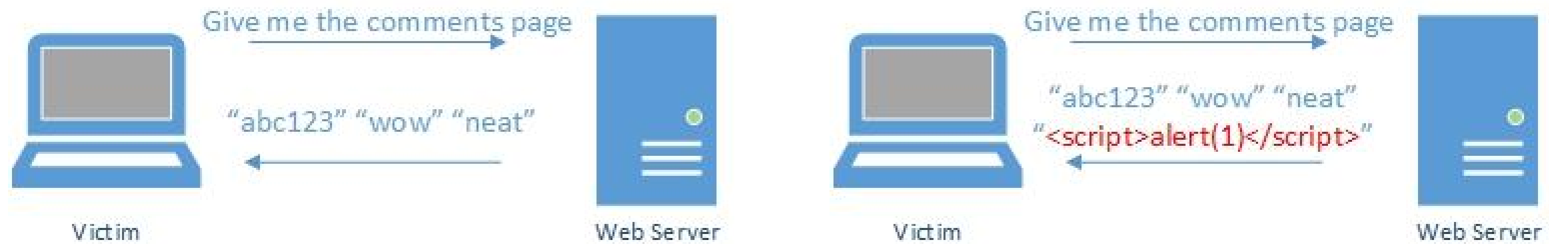
Consider a different page

- Comment page
- Users submit comments, which are stored in a database
- Every time someone loads the page, all the comments are pulled from the database and are included in the page



Stored XSS

- Attacker submits his/her payload in a malicious comment
    - <script>alert(1)</script>
- The payload is stored in the database, and included in the page
- Now **every time** someone loads the page, the payload is executed



- More difficult to detect than reflected XSS
    - http://vulnerable.com/welcome.php?name=<script>alert(1)</script>
    - http://vulnerable.com/comments.php

SQL Injection (SQLi)

- End result: attacker executes his/her own SQL commands on the website's backend database
  - Steal data, delete data, deface website, attempt to escalate privileges (pwn the web server)…
- Submitting input that escapes out of the intended structure of the SQL query
- Can then write his/her own SQL
- Attacks target the backend database (not the user)
  - Although it impacts users, the direct focus is on the server side

Consider a login function

- User submits a username and a password
- Username and password are entered into a SQL query
- Suppose we login with u:dave and p:pass123
- The structure of the query looks like:
  - SELECT * FROM users WHERE username='dave' and password='pass123';
- What if we were to enter
  - u:' p:pass123

Now we have..

- SELECT * FROM users WHERE username=’’’ and password=’pass123’;
- This generates a SQL error
  - Odd number of single quotes

Hmm..

- Let’s try u:’ or 1=1; # p:pass123
- SELECT * FROM users WHERE username=’’ or 1=1; #’ and password=’pass123’;
  - # starts a SQL comment, the pink text has been commented out
- This is **valid SQL**, and will return **all users** in the table
- The payload can also be crafted so the single quotes match up
  - u:’ or ‘1’=’1
  - SELECT * FROM users WHERE username=’’ or ‘1’=’1’;
  - A comment isn’t necessary

Resources

- Web Application Hacker’s Handbook
  - Highly recommended
- Set up a VM lab
  - Kali w/ Burp Suite
  - DVWA (http://www.dvwa.co.uk/)
  - PentesterLab (https://pentesterlab.com/)
  - OWASP WebGoat (https://github.com/WebGoat/WebGoat)
  - Many others
- http://kukfa.co/resources/web-application-cheat-sheet/

Questions?

- dxk2652@rit.edu
- Future presentation: defending against these attacks? evading filters and web application firewalls? live demo?