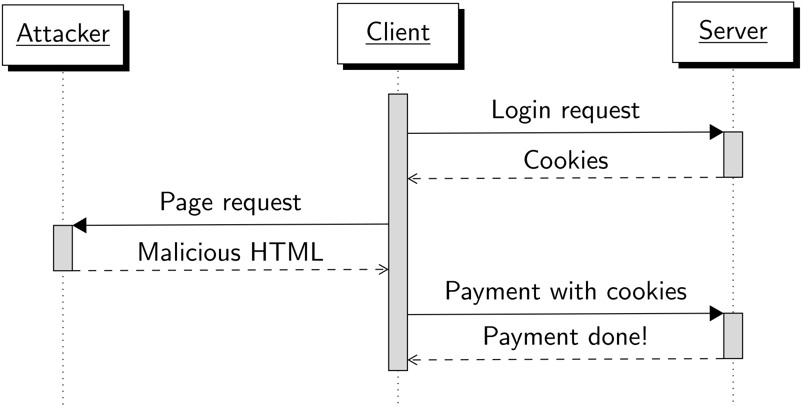
Security II - CSRF & XSSI

Stefano Calzavara

Università Ca' Foscari Venezia

February 24, 2020

CSRF: Recap


Attacker
Client
Server
Login request
Cookies
Page request
Malicious HTML
Payment with cookies
Payment done!

How to Prevent CSRF?

CSRF is enabled by the attachment of session cookies to HTTP requests forged by malicious pages.

Server-Side Fixes

Do not authenticate requests based on cookies alone: there are many different techniques, each with pros and cons [1].

Client-Side Fixes

Change the way cookies work: modern browsers offer a native protection mechanism via the SameSite cookie attribute.

Referer Checking

A possible defence against CSRF is checking the content of the `Referer` header of each security-sensitive HTTP request. This header contains the URL of the page which sent the request.

Alert!

This is effective, yet there are at least two problematic cases:

1. some legitimate HTTP requests might lack the `Referer` header
2. some legitimate HTTP requests might come with an unexpected value of the `Referer` header

Do you see why this might happen?

Referer Checking

Ensure that your checks over the `Referer` value are appropriate!

Example

Let us assume you want to protect `www.good.com`:

- notice that the scheme must be part of the check, e.g., check for `https://www.good.com`
- a smart attacker could try to bypass this check by sending HTTP requests from `https://www.good.com.evil.com`
- beware of untrusted subdomains like `evil.good.com`

Origin Checking

Rather than checking the value of the `Referer` header, one can check the value of `Origin` header (from CORS)

- privacy-friendly version of `Referer`, which can be stripped away by benign websites using Referer Policy
- always sent along with XHR requests
- in modern browsers: also sent in cross-origin POST requests

In general, Origin checking should be preferred over Referer checking, but the two mechanisms share similar limitations.

Custom Headers

Another defence relies on custom headers, e.g., `CSRF-Protection: 1`. The presence of the header suffices, since SOP prevents the inclusion of custom headers on cross-origin requests.

Alert!

Compared to Referer / Origin checking, this mechanism is simpler to implement correctly, but it is also less flexible:

- restricts security-sensitive requests to same-origin pages (yet this can be relaxed by using CORS)
- requires the web application logic to be built on top of JS and XHR

Secret Tokens

The most common defence against CSRF deployed in the wild is the inclusion of secret tokens as part of security-sensitive requests.

Example

```
<form method="post" action="/items/12345">
   <input type="submit" name="like" value="1"/>
   <input type="hidden" name="token" value="ff34821b"/>
</form>
```

The expected value of the secret token is typically stored in the user's session at the server side.

Secret Tokens

This works because security-sensitive requests are not authenticated by the cookies alone:

- the attacker cannot read content from the DOM of a page on another origin, hence cannot access the token from the form
- the attacker can force the browser into sending an HTTP request with the session cookies, but will not be able to attach the right token to it as a parameter
- tokens offer better flexibility than header-based approaches: they are the most popular defence against CSRF as of now and are supported by many frameworks

Popular Pattern: Double Submit

The double submit pattern is a popular approach to the use of tokens:

- the token is still embedded as a parameter of each sensitive HTTP request, as in the previous example, but the right value of the token is stored inside a cookie
- every time a sensitive HTTP request is received, the server checks that the value of the cookie matches the value of the parameter

This is particularly useful when sessions rely just on client-side state.

Double Submit: Cookie Confidentiality

Since the double submit pattern stores a secret token inside a cookie, the confidentiality of the cookie must be ensured:

- mark the cookie with the `Secure` attribute to prevent its disclosure
- perhaps surprisingly, notice that the `HttpOnly` attribute does not provide any help here!

No HttpOnly?

- In case of XSS, a malicious script can read the token from the DOM
- Tokens are normally attached to forms by JS accessing the cookie

Double Submit: Cookie Integrity

Recall that cookies offer no integrity guarantee against network attackers in their default configuration:

- consider the use of the `__Secure-` prefix
- to protect legacy browsers lacking support for cookie prefixes, ensure the token is generated from a session-dependent secret
- otherwise, the attacker's token could be forced into the victim's browser, i.e., the victim's session would be unprotected

Do you see how a possible attack would work?

SameSite Cookies

Cookies marked with the `SameSite` attribute can be configured so that they are not attached to cross-site requests:

- "site" = registrable domain, e.g., `google.com` and its subdomains
- `SameSite=Strict`: applies this policy to every HTTP request
- `SameSite=Lax`: relaxes this restriction in the case of top-level navigations with a safe method, e.g., resulting from clicking a link

This defence does not offer protection to legacy browsers, hence also traditional defences like tokens should be implemented!

Migrating to SameSite Cookies

While the `SameSite` attribute is widely supported, it has unfortunately not been largely adopted by developers:

- browser vendors discussed the idea of automatically enforcing the attribute, with Google Chrome taking the lead on this (version 80)
- Google Chrome now marks all cookie as `SameSite=Lax` by default
- if web developers do not want this new default behaviour, they can mark their cookies as `SameSite=None`
- cookies marked as `SameSite=None` must also be marked `Secure`

Login CSRF

If the login form of a web application is not protected against CSRF, the attacker can force the victim into authenticating using the attacker's account: this attack is known as login CSRF.

```
<form action="https://www.good.com/login" method="POST">
    <input name="username" value="attacker">
    <input name="password" value="ev1l.pwd">
</form>
<script>document.forms[0].submit()</script>
```

This attack sounds bizarre! Can you figure out cases where this is a real security problem?

The Dangers of Login CSRF

Login CSRF is not always dangerous, yet...

Example

Since `google.com` stores all the search history of authenticated users, an attacker can exploit a login CSRF on `google.com` to access the complete search history of the victim.

Example

Since `paypal.com` binds a credit card number to a personal account, an attacker can exploit a login CSRF on `paypal.com` to leak the credit card number of the victim.

Preventing Login CSRF

To prevent login CSRF, you can rely on existing CSRF defences:

- header checks upon login form submission: like traditional CSRF, also login CSRF is enabled by cross-site requests
- secret tokens in the login form: since login CSRF happens before the session, you must setup an unauthenticated session for the token
- SameSite cookies: require the presence of a SameSite cookie upon login form submission. This solution is always effective against web attackers, but what about network attackers?

CSRF Prevention: Summary

Observe that:

- Checking the content of the `Referer` / `Origin` header or just the presence of custom headers might work, but this is often impractical
- Secret tokens are better for most applications, but implementation is not straightforward. Most importantly, the security of tokens relies on a correct enumeration of all security-sensitive requests [2]
- SameSite cookies are a simple and elegant solution against CSRF, which solves the issues of tokens, but only protects modern browsers

XSS vs CSRF

Both XSS and CSRF bypass the protection offered by SOP. Notice that:

- if a web application is vulnerable against XSS, none of the proposed defences against CSRF is effective. This means that XSS is a more serious security concern than CSRF in most cases
- in some cases, CSRF can be just as dangerous as XSS. For example, CSRF can sometimes lead to account takeover. Can you think about real-world examples where this might happen?

Bottom line: do not take any of these two vulnerabilities lightly!

Cross Site Script Inclusion (XSSI)

A less known attack abusing cross-site requests is called Cross Site Script Inclusion or XSSI for short [3].

XSSI in Practice

1. The victim authenticates at `good.com` and later visits `evil.com`
2. The page at `evil.com` loads a script from `good.com`
3. Since the script inclusion request contains the victim's cookies, the script might be dynamically generated to include private information
4. The page at `evil.com` uses JS to exfiltrate the secret from the script

Scoping in JavaScript

JavaScript variables live in the global scope by default

- ... even when declared within a function!
- you can make variables local to a function by using the var keyword

```
var globalVariable1 = 5;          // A global variable

function globalFunction() {
    var localVariable = 2;        // A local variable
    globalVariable2 = 3;          // Another global variable
    window.globalVariable3 = 4;   // Yet another global
}
```

Scoping in JavaScript

While C++ or Java make use of block scoping, JavaScript utilizes the so-called function scoping:

- the JS engine creates a new scope for each encountered function
- an identifier that is locally defined within a function is associated with the function scope, irrespective of blocks
- you can enforce block scoping by using the `let` keyword

Advice: there is nothing using `var` that `let` can't do better...

Scoping in JavaScript

What's the output of the following piece of code?

```
var age = 100;
if (age > 12) {
  var dogYears = age * 7;
  console.log('You are ${dogYears} dog years old!');
}
console.log('Value of dogYears: ${dogYears}');
```

Stealing Secrets via XSSI: Part 1

How do we attack here?

```
// snippet of the file https://good.com/js/pay.js
function doPayment() {
  info = {ccn: "verysecret"};
  // payment logic implementation
}
```

Stealing Secrets via XSSI: Part 1

How do we attack here?

```
// snippet of the file https://good.com/js/pay.js
function doPayment() {
  info = {ccn: "verysecret"};
  // payment logic implementation
}
```

Exploit

```
<script src="https://good.com/js/pay.js"/>
leak(info);
```

Stealing Secrets via XSSI: Part 2

How do we attack here?

```
// snippet of the file https://good.com/js/pay.js
function doPayment() {
  var info = {ccn: "verysecret"};
  // payment logic implementation
  return JSON.stringify(info);
}
```

Stealing Secrets via XSSI: Part 2

How do we attack here?

```
// snippet of the file https://good.com/js/pay.js
function doPayment() {
  var info = {ccn: "verysecret"};
  // payment logic implementation
  return JSON.stringify(info);
}
```

Exploit

```
JSON.stringify = function(x) { leak(x); }
<script src="https://good.com/js/pay.js"/>
```

Inheritance in JavaScript

Inheritance in JavaScript is not based on classes, but directly on objects known as prototypes.

```
var o1 = {a: 1};    // prototype is Object.prototype

var o2 = Object.create(o1);  // prototype is o1

console.log(o2.a);  // prints 1
```

Method invocations traverse the prototype chain looking for a valid implementation, up to the root `Object.prototype`.

Stealing Secrets via XSSI: Part 3

How do we attack here?

```
// snippet of the file https://good.com/js/pay.js
function doPayment() {
  var data = ["ccn1","ccn2","ccn3"];
  var x = data.slice(1);
  // payment logic implementation
}
```

Stealing Secrets via XSSI: Part 3

How do we attack here?

```
// snippet of the file https://good.com/js/pay.js
function doPayment() {
  var data = ["ccn1","ccn2","ccn3"];
  var x = data.slice(1);
  // payment logic implementation
}
```

Exploit

```
Array.prototype.slice = function(x) { leak(this); }
<script src="https://good.com/js/pay.js"/>
```

Preventing XSSI

XSSI is different from CSRF, yet the attack vector is the same:

- any of the proposed defences against CSRF is useful against XSSI
- however, XSSI makes the attack surface on web apps even larger!
- XSSI can also be prevented by defensive programming

```
function doPayment() {
  var info = {ccn: "verysecret"};
  var myserialize = function(x) { ... };
  return mysrialize(info);
}
```

Preventing XSSI

A better, general solution leverages SOP:

1. script code is never generated on the fly based on session cookies, but always pulled from a static file
2. sensitive and dynamic data values are kept in a separate file, which cannot be interpreted by the browser as JavaScript
3. when the static JavaScript gets executed, it sends an XHR to the file containing the secret data
4. use CORS to selectively grant read access to third parties

References

Adam Barth, Collin Jackson, and John C. Mitchell.
Robust defenses for cross-site request forgery.
In *ACM CCS*, pages 75–88, 2008.

Stefano Calzavara, Mauro Conti, Riccardo Focardi, Alvise Rabitti, and Gabriele Tolomei.
Mitch: A machine learning approach to the black-box detection of CSRF vulnerabilities.
In *IEEE EuroS&P*, pages 528–543, 2019.

Sebastian Lekies, Ben Stock, Martin Wentzel, and Martin Johns.
The unexpected dangers of dynamic javascript.
In *USENIX Security Symposium*, pages 723–735, 2015.