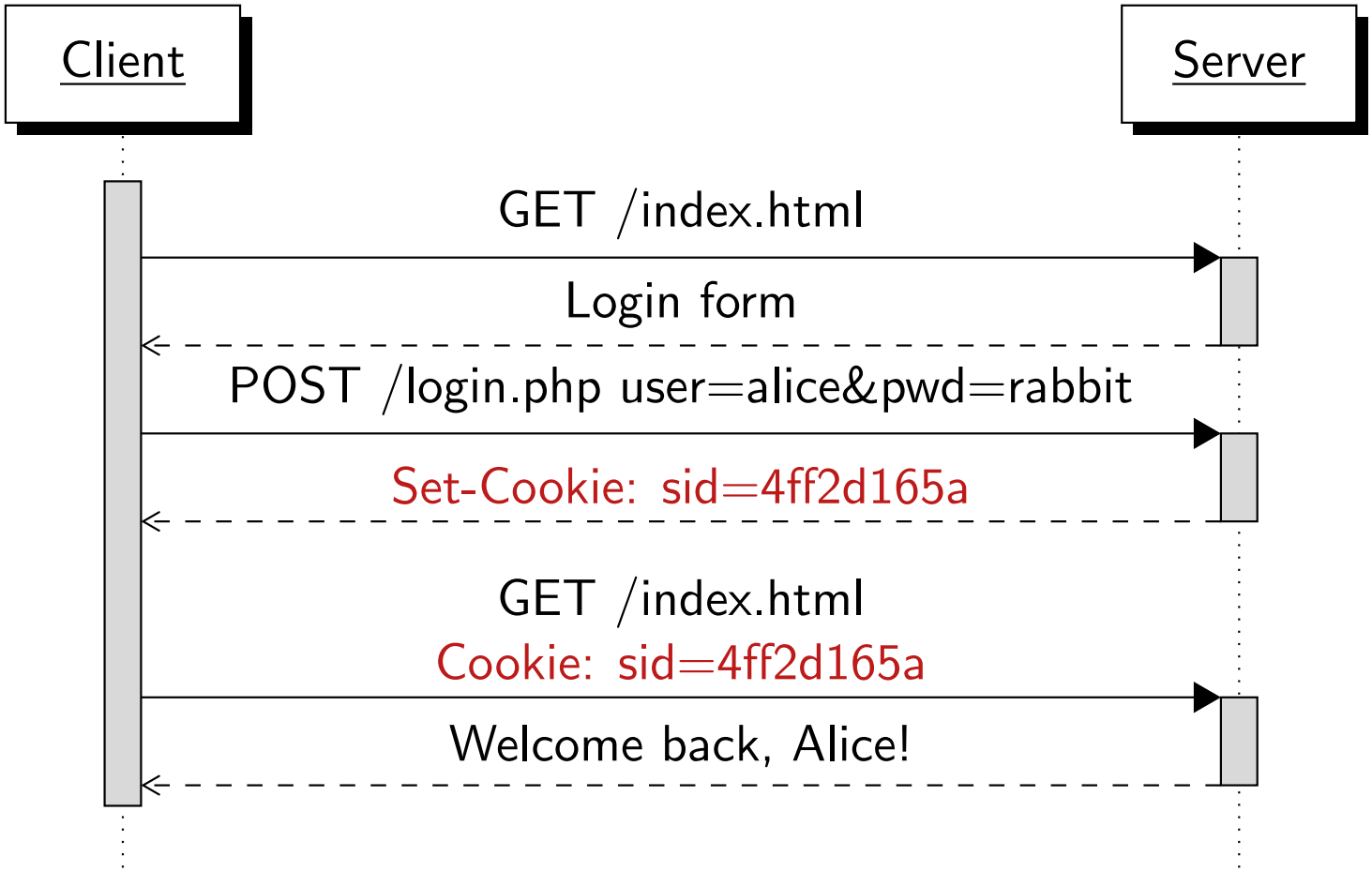
Web Security - Web Session Security

Stefano Calzavara

Università Ca' Foscari Venezia

Università
Ca'Foscari
Venezia

Authenticated Web Sessions



Session Management

Server-side State (PHP)

Persist the state information on the server, e.g., in a database, and use the cookie just to store a session identifier:

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

Client-side State (Flask)

Persist the state information directly into the cookie:

- a secure implementation requires the use of cryptography, at least for signing the cookie and possibly for encrypting it
- sometimes not possible, because cookies are limited in size (4 KB)
- no database: better scalability for high-volume applications

Session Integrity

A honest user's session enjoys integrity when an attacker cannot forge requests which get authenticated on the user's behalf [1, 2].

Three classic attacks against session integrity:

1. session hijacking: if the attacker can steal the victim's cookies, they can impersonate the victim at the server
2. session fixation: if the attacker can fix the victim's cookies to a known value, they can impersonate the victim at the server
3. cross-site request forgery: if the attacker can forge requests from the victim's browser, such requests might look legitimate to the server

Cookie Security: Web Attackers

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

Alert!

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

1. `good.com` directly loads scripts from `evil.com`, e.g., by embedding advertisement or libraries within its web pages
2. `good.com` suffers from a cross-site scripting (XSS) vulnerability, which allows the attacker to inject a script at `good.com`

We will discuss XSS in detail in the next lectures.

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, thus offering confidentiality against web attackers.

Secure Cookies

Cookies marked with the `Secure` attribute are only sent over HTTPS and made inaccessible to JavaScript running in HTTP pages.

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

Leaking Non-Secure Cookies on HTTPS Websites

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://unrelated.com`
2. The attacker corrupts the response from `http://unrelated.com` so that it triggers a request to `http://www.good.com`
3. The browser now tries to access `http://www.good.com`
4. Although the request fails, the session cookies are leaked in clear!

Pre-Sessions and Session Fixation

Many web applications use pre-sessions (unauthenticated sessions) to keep track of state information even before login.

Example

Most e-commerce applications allow users to add items to their shopping cart before login for usability reasons. The content of the cart is persisted upon login to enable payments.

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

Let us assume that `www.good.com` uses pre-sessions, but does not refresh session cookies upon login:

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 authenticates at `good.com`, which does not refresh the cookie value, hence the forced cookie identifies the victim's session
4. Since the attacker knows the cookie, they can hijack the victim's session at `good.com`

Preventing Session Fixation

Two simple options to prevent session fixation:

1. Refresh cookies upon authentication: when the credentials are verified, change the value of the cookie and start a new session
2. Enforce cookie integrity: ensure that the attacker cannot set cookies for the target web application, as discussed in the next slides

It is recommended to follow both practices to provide protection to the largest number of accessing clients (it might be hard to enforce cookie integrity in legacy clients).

Cookies Lack Integrity Against Network Attackers

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://unrelated.com`
2. The attacker corrupts the response from `http://unrelated.com` 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 exploiting the `Domain` attribute of the cookie!

Enforcing Cookie Integrity on HTTPS Websites

The `Secure` attribute does not enforce cookie integrity!

- in legacy browsers, it simply does not provide any integrity protection
- in modern browsers, cookies with the `Secure` attribute cannot be set or overwritten over HTTP, but the attacker can forge non-Secure cookies with the same name before the Secure cookies are set!

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

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

Alert!

Cookies prefixes are not supported by all browsers!

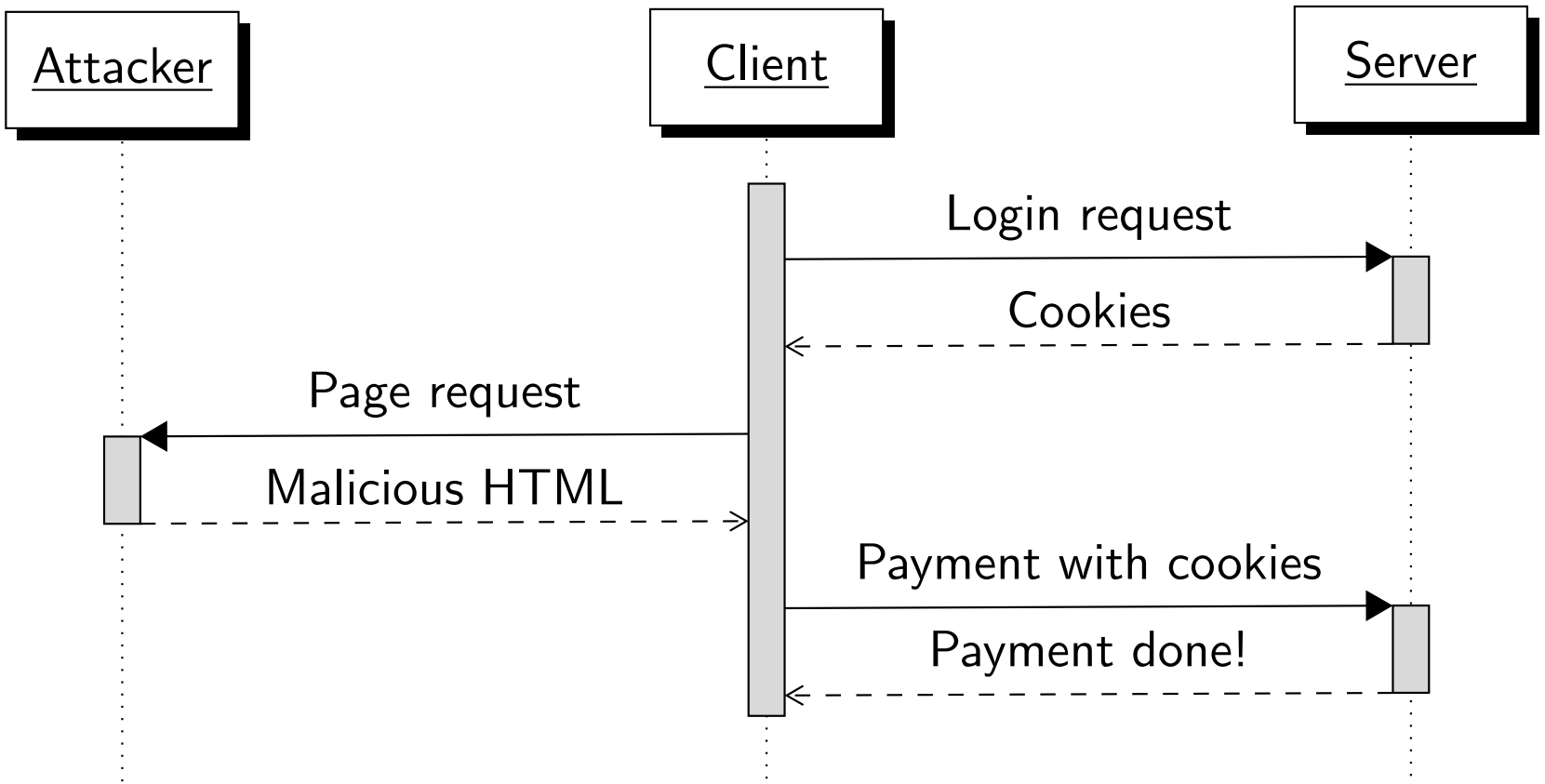
Cross-Site Request Forgery (CSRF)

Since cookies are automatically attached to HTTP requests by default, an attacker can force the creation of authenticated requests from the victim's browser, which might trigger security-sensitive actions (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

Cross-Site Request Forgery (CSRF)



Same-Site Cookies

Modern browsers support the `SameSite` cookie attribute, which can be used to prevent cookies from being attached to cross-site requests:

- site = registrable domain (e.g., google.com) and its subdomains
- effective protection against CSRF, which is enforced automatically in modern browsers (caveats apply!)
- legacy browsers can use other techniques to mitigate CSRF

We will discuss CSRF defenses in detail in the next lectures.

Questions

Let's check what you have understood so far!

1. What is the root cause of session hijacking?

Questions

Let's check what you have understood so far!

1. What is the root cause of session hijacking?
2. Does cookie confidentiality alone suffice to ensure that session hijacking is not possible?

Questions

Let's check what you have understood so far!

1. What is the root cause of session hijacking?
2. Does cookie confidentiality alone suffice to ensure that session hijacking is not possible?
3. What is the root cause of session fixation?

Questions

Let's check what you have understood so far!

1. What is the root cause of session hijacking?
2. Does cookie confidentiality alone suffice to ensure that session hijacking is not possible?
3. What is the root cause of session fixation?
4. Are session hijacking and session fixation equivalent from a security perspective, i.e., in terms of severity?

Questions

Let's check what you have understood so far!

1. What is the root cause of session hijacking?
2. Does cookie confidentiality alone suffice to ensure that session hijacking is not possible?
3. What is the root cause of session fixation?
4. Are session hijacking and session fixation equivalent from a security perspective, i.e., in terms of severity?
5. Are cookie confidentiality or integrity violated by CSRF?

Questions

Let's check what you have understood so far!

1. What is the root cause of session hijacking?
2. Does cookie confidentiality alone suffice to ensure that session hijacking is not possible?
3. What is the root cause of session fixation?
4. Are session hijacking and session fixation equivalent from a security perspective, i.e., in terms of severity?
5. Are cookie confidentiality or integrity violated by CSRF?
6. Is CSRF as dangerous as session hijacking?

Beyond Session Integrity: Cookie Forcing

We discussed attacks where the attacker tampers with the user's session. However, the attacker can also push the victim into the attacker's session!

Cookie Forcing

1. The attacker authenticates at `good.com` with their own credentials
2. The attacker forces the session cookie into the victim's browser, e.g., by forging HTTP traffic from `good.com`
3. The user later navigates `good.com` and is forced into the attacker's session (in a sense, this is the "dual" of session hijacking)

Obvious fix: ensure cookie integrity as discussed!

The Dangers of Cookie Forcing

Cookie forcing is not necessarily dangerous, but can be nasty [3].

Example

Since `google.com` stores all the search history of authenticated users, an attacker can perform cookie forcing on `google.com` to eventually access the complete search history of the victim (since the attack).

Example

Since `paypal.com` binds a credit card number to a personal account, an attacker can perform cookie forcing on `paypal.com` to eventually get access to the credit card number of the victim.

Session Expiration

Cookies are deleted by default when the web browser is closed, but web applications can modify this through the `Expires` attribute:

- long session lifetimes improve usability, but might harm security
- make the "Remember Me" functionality an opt-in, not a default

Alert!

Do not rely on the `Expires` attribute for session expiration:

- although a cookie is not around anymore since it has expired on the client, the corresponding session might still be open at the server!
- the `Expires` attribute alone does not reduce the window of time where a stolen / brute-forced cookie can be used by the attacker!

Session Expiration

How to implement session expiration?

Server-Side State

Expiration is simple to implement: just invalidate the session identifier by deleting the corresponding entries from the database.

Client-Side State

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

Note that the blacklist requires the use of server-side state, which may be undesired when the adoption of client-side state is preferred!

Case Study: Sessions in PHP

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

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

Modern versions of PHP switched to cookie-based sessions by default: see `session.use_only_cookies` in `php.ini`

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 `$_SESSION` variable can then be used as a dictionary to bind session data to keys, e.g., `$_SESSION["name"] = "Alice"`
3. later invocations to `session_start()`, e.g., in other PHP pages, retrieve the content of `$_SESSION` through the cookie

You can change the default name of the session cookie by setting a new value for `session.name` in `php.ini`

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 `session.cookie_secure`/`session.cookie_httponly` in `php.ini`
- the `session.use_strict_mode` option in `php.ini` forces the use of valid session identifiers generated by the web application. This does not prevent all session fixation attacks, but it is a useful mitigation
- use the `session_regenerate_id()` function to refresh the session identifier when you feel the need, i.e., at least upon login
- use the `session_destroy()` function to terminate the session upon logout (you still need to call `session_start()` first)
- call `session_unset()` before `session_destroy()` to purge even the temporary memory for extra safety

A Glance at Sessions in Flask

Flask sessions are based on client-side state by default:

- the content of the session is saved in the cookie as JSON
- the cookie is marked as `HttpOnly` by default (but not `Secure`)
- the cookie is cryptographically signed, but not encrypted, so you can read its content (decoder)
- most popular authentication library: Flask-Login operating on top of the existing session mechanism of Flask
- do you prefer PHP-style sessions with server-side state? You can also activate them in Flask! Take a look into Flask-Session

Flask-Login: Fresh Logins

When a user logs in, their session is marked as fresh:

- The session cookie is marked without the `Expires` attribute, meaning that it is deleted when the browser is closed.
- However, a second "remember me" cookie with a longer duration can also be set upon authentication: logging in with this cookie marks the session as non-fresh.
- The `@login_required` decorator does not check freshness, which is fine for most pages. But the most sensitive actions, e.g., update of credit information, should require a fresh session.
- You can use the `@fresh_login_required` to require the use of a fresh session. If the session is non-fresh, a new login is required.

Flask-Login: Session Protection

When a session is first established, Flask-Login saves a hash of the IP address and the user agent performing the login request.

Flask-Login offers three levels of session protection to mitigate the risks of session hijacking:

- `"strong"`: if the hash of the request does not match the hash of the session, the session does not start (no authentication).
- `"basic"`: if the hash of the request does not match the hash of the session, the session starts but it is marked as non-fresh.
- `None`: the hash of the session is not used.

References

[1] Stefano Calzavara, Alvise Rabitti, Alessio Ragazzo, and Michele Bugliesi.
Testing for integrity flaws in web sessions.
In *ESORICS 2019*, pages 606–624, 2019.

[2] OWASP Foundation.
OWASP Testing Guide (Chapter 4.6).
`https://www.owasp.org/index.php/OWASP_Testing_Guide_v4_Table_of_Contents`, 2016.

[3] Xiaofeng Zheng, Jian Jiang, Jinjin Liang, Hai-Xin Duan, Shuo Chen, Tao Wan, and Nicholas Weaver.
Cookies lack integrity: Real-world implications.
In Jaeyeon Jung and Thorsten Holz, editors, *24th USENIX Security Symposium, 2015*, pages 707–721, 2015.