Web security: web basics

Myrto Arapinis
School of Informatics
University of Edinburgh

March 19, 2015
Web applications

Client
(HTML, JavaScript)

Server
(PHP)

Database
(SQL)

HTTP
Protocol://host/FilePath?argt1=value1&argt2=value2

- **Protocol**: protocol to access the resource (http, https, ftp, ...)
- **host**: name or IP address of the computer the resource is on
- **FilePath**: path to the resource on the host
- Resources can be static (file.html) or dynamic (do.php)
- URLs for dynamic content usually include arguments to pass to the process (argt1, argt2)
HTTP requests

GET request

GET HTTP/1.1
Host: www.inf.ed.ac.uk
User-Agent: Mozilla/5.0
(X11; Ubuntu; Linux x86_64; rv:29.0)
  Gecko/20100101 Firefox/29.0
Accept: text/html,application/xhtml+xml,
  application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
HTTP responses

HTTP/1.1 200 OK
Server: Apache
Cache-control: private
Set-Cookie: JSESSIONID=B7E2479EC28064DF84DF4E3DBEE9C7DF; Path=/
Content-Type: text/html;charset=UTF-8
Date: Wed, 18 Mar 2015 22:36:30 GMT
Connection: keep-alive
Set-Cookie: NSC_xxx.fe.bd.vl-xd=fffffffffc3a035be45525d5f4f58455e445a423660;path=/
Content-Encoding: gzip
Content-Length: 4162

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
     xml:lang="en" lang="en">
<head>
<title>Informatics home | School of Informatics</title>
...</
Web security: security goals
Security goals

Web applications should provide the same security guarantees as those required for standalone applications

1. visiting evil.com should not infect my computer with malware, or read and write files
   **Defenses:** Javascript sandboxed, avoid bugs in browser code, privilege separation, *etc*

2. visiting evil.com should not compromise my sessions with gmail.com
   **Defenses:** same-origin policy – each website is isolated from all other websites

3. sensitive data stored on gmail.com should be protected
**Threat model**

**Web attacker**
- controls evil.com
- has valid SSL/TLS certificates for evil.com
- victim user visits evil.com

**Network attacker**
- controls the whole network: can intercept, craft, send messages

A Web attacker is weaker than a Network attacker
<table>
<thead>
<tr>
<th>OWASP TOP 10 Web security flaws (2013)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A1 – Injection</strong></td>
</tr>
<tr>
<td>Injection flaws, such as SQL, OS, and LDAP injection occur when untrusted data is sent to an interpreter as part of a command or query. The attacker’s hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.</td>
</tr>
<tr>
<td><strong>A2 – Broken Authentication and Session Management</strong></td>
</tr>
<tr>
<td>Application functions related to authentication and session management are often not implemented correctly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users’ identities.</td>
</tr>
<tr>
<td><strong>A3 – Cross-Site Scripting (XSS)</strong></td>
</tr>
<tr>
<td>XSS flaws occur whenever an application takes untrusted data and sends it to a web browser without proper validation or escaping. XSS allows attackers to execute scripts in the victim’s browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites.</td>
</tr>
<tr>
<td><strong>A4 – Insecure Direct Object References</strong></td>
</tr>
<tr>
<td>A direct object reference occurs when a developer exposes a reference to an internal implementation object, such as a file, directory, or database key. Without an access control check or other protection, attackers can manipulate these references to access unauthorized data.</td>
</tr>
<tr>
<td><strong>A5 – Security Misconfiguration</strong></td>
</tr>
<tr>
<td>Good security requires having a secure configuration defined and deployed for the application, frameworks, application server, web server, database server, and platform. Secure settings should be defined, implemented, and maintained, as defaults are often insecure. Additionally, software should be kept up to date.</td>
</tr>
<tr>
<td><strong>A6 – Sensitive Data Exposure</strong></td>
</tr>
<tr>
<td>Many web applications do not properly protect sensitive data, such as credit cards, tax IDs, and authentication credentials. Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes. Sensitive data deserves extra protection such as encryption at rest or in transit, as well as special precautions when exchanged with the browser.</td>
</tr>
<tr>
<td><strong>A7 – Missing Function Level Access Control</strong></td>
</tr>
<tr>
<td>Most web applications verify function level access rights before making that functionality visible in the UI. However, applications need to perform the same access control checks on the server when each function is accessed. If requests are not verified, attackers will be able to forge requests in order to access functionality without proper authorization.</td>
</tr>
<tr>
<td><strong>A8 – Cross-Site Request Forgery (CSRF)</strong></td>
</tr>
<tr>
<td>A CSRF attack forces a logged-on victim’s browser to send a forged HTTP request, including the victim’s session cookie and any other automatically included authentication information, to a vulnerable web application. This allows the attacker to force the victim’s browser to generate requests the vulnerable application thinks are legitimate requests from the victim.</td>
</tr>
<tr>
<td><strong>A9 – Using Components with Known Vulnerabilities</strong></td>
</tr>
<tr>
<td>Components, such as libraries, frameworks, and other software modules, almost always run with full privileges. If a vulnerable component is exploited, such an attack can facilitate serious data loss or server takeover. Applications using components with known vulnerabilities may undermine application defenses and enable a range of possible attacks and impacts.</td>
</tr>
<tr>
<td><strong>A10 – Unvalidated Redirects and Forwards</strong></td>
</tr>
<tr>
<td>Web applications frequently redirect and forward users to other pages and websites, and use untrusted data to determine the destination pages. Without proper validation, attackers can redirect victims to phishing or malware sites, or use forwards to access unauthorized pages.</td>
</tr>
</tbody>
</table>
Injection attacks
Injection attack

OWASP definition

Injection flaws, such as SQL, OS, and LDAP injection occur when untrusted data is sent to an interpreter as part of a command or query. The attacker’s hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.

We are going to look at:

- command injection attacks
- SQL injection attacks
Command injection: a simple example

- Service that prints the result back from the linux program whois

- Invoked via URL like (a form or Javascript constructs this URL):
  

- Possible implementation of content.php

```php
<?php
  if ($_GET['domain']) {
    <? echo system("whois".$_GET['domain']); ?>
  }
?>
```
This script is subject to a command injection attack! We could invoke it with the argument

```
www.example.com; rm -rf /;
```

http://www.example.com/domain.php?domain=example.php;

```
rm -r /;
```

Resulting in the following PHP

```
<? echo system("whois www.example.com; rm -rf/;"); ?>
```
<? echo system("whois".escapeshellarg($_GET['domain'])); ?>

escapeshellarg() adds single quotes around a string and quotes/escapes any existing single quotes allowing you to pass a string directly to a shell function and having it be treated as a single safe argument

<table>
<thead>
<tr>
<th>GET INPUT</th>
<th>Command executed</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://www.example.com">www.example.com</a></td>
<td>whois 'www.example.com'</td>
</tr>
<tr>
<td><a href="http://www.example.com">www.example.com</a>; rm -rf;/</td>
<td>whois 'www.example.com rm -rf/;'</td>
</tr>
</tbody>
</table>
Command injection recap

- Injection is generally caused when data and code share the same channel:
  - "whois" is the code and the filename the data
  - **But** ';;' allows attacker to include new command

- **Defenses** include input validation, input escaping and use of a less powerful API
Web applications

Client (HTML, JavaScript)

Server (PHP)

Database (SQL)

HTTP
Databases

<table>
<thead>
<tr>
<th>username</th>
<th>password</th>
</tr>
</thead>
<tbody>
<tr>
<td>alice</td>
<td>01234</td>
</tr>
<tr>
<td>bob</td>
<td>56789</td>
</tr>
<tr>
<td>charlie</td>
<td>43210</td>
</tr>
</tbody>
</table>

user_accounts

- Web server connects to DB server:
  - Web server sends **queries** or **commands** according to incoming HTTP requests
  - DB server returns associated values
  - DB server can **modify/update** records
- SQL: commonly used database query language
SQL SELECT

Retrieves a set of records from DB:

```
SELECT field FROM table WHERE condition -- SQL comment
```

returns the value(s) of the given field in the specified table, for all records where condition is true

Example:

<table>
<thead>
<tr>
<th>username</th>
<th>password</th>
</tr>
</thead>
<tbody>
<tr>
<td>alice</td>
<td>01234</td>
</tr>
<tr>
<td>bob</td>
<td>56789</td>
</tr>
<tr>
<td>charlie</td>
<td>43210</td>
</tr>
</tbody>
</table>

```
SELECT password FROM user_accounts WHERE username='alice'
```
returns the value 01234
SQL INSERT

Retrieve a set of records from DB:

```
INSERT INTO table VALUES record -- SQL comment
```

adds the value(s) a new record in the specified table

Example:

```
\begin{tabular}{|l|l|}
\hline
username & password  \\
\hline
alice   & 01234  \\
bob     & 56789  \\
charlie & 43210  \\
\hline
\end{tabular}
\quad \rightarrow \quad
\begin{tabular}{|l|l|}
\hline
username & password  \\
\hline
alice   & 01234  \\
bob     & 56789  \\
charlie & 43210  \\
eve     & 98765  \\
\hline
\end{tabular}
```

```
INSERT INTO user_accounts VALUES ('eve', 98765)
```
Other SQL commands

- **DROP TABLE table**: deletes entire specified table

- Semicolons separate commands:

  **Example:**

  ```sql
  INSERT INTO user_accounts VALUES ('eve', 98765);
  SELECT password FROM user_accounts
  WHERE username='eve'
  ```

  returns 98765
The web server logs in a user if the user exists with the given username and password.

**login.php:**

```php
$conn = pg_pconnect("dbname=user_accounts");
$result = pg_query($conn,
  "SELECT * from user_accounts
  WHERE username = " .$_GET['user'].""
  AND password = " .$_GET['pwd']."";");
if(pg_query_num($result) > 0) {
  echo "Success"
  user_control_panel_redirect();
}
```

It sees if results exist and if so logs the user in and redirects them to their user control panel.
SQL injection: a simple example

Login as admin:
SQL injection: a simple example

Login as admin:

http://www.example.com/login.php?user=admin’--&pwd=f

pg_query(conn,
    "SELECT * from user_accounts
    WHERE username = 'admin' -- ' AND password = 'f';";);
SQL injection: a simple example

Login as admin:
http://www.example.com/login.php?user=admin’--&pwd=f
pg_query(conn,
    "SELECT * from user_accounts
    WHERE username = 'admin' -- ' AND password = 'f';");

Drop user_accounts table:
SQL injection: a simple example

Login as admin:
http://www.example.com/login.php?user=admin’--&pwd=f

pg_query(conn,
    "SELECT * from user_accounts
    WHERE username = 'admin' -- ' AND password = 'f';";
);

Drop user_accounts table:
http://www.example.com/login.php?user=admin’;
    DROP TABLE user_accounts --&pwd=f

pg_query(conn,
    "SELECT * from user_accounts;
    WHERE user = 'admin'; DROP TABLE user_accounts;
    -- ' AND password = 'f';";
);
Defense: prepared statements

- Creates a template of the SQL query, in which data values are substituted
- Ensures that the untrusted value is not interpreted as a command

```php
$result = $conn->query_params(
    SELECT * from user_accounts WHERE username = $1
    AND password = $2,
    array($_GET['user'], $_GET['pwd']));
```