Web security: web basics

Myrto Arapinis
School of Informatics
University of Edinburgh

March 19, 2015

Web applications

HTTP

Client
(HTML, JavaScript)

Server
(PHP)

Database
(SQL)

URLs

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
Web security: security goals

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
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

**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.

Service that prints the result back from the Linux program `whois`

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

```html
```

Possible implementation of `content.php`:

```php
<?php
  if ($_GET['domain']) {
    echo system("whois".$_GET['domain']);
  }
?>
```
Command injection: a simple example cont’d

- 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/; "); ?>
  ```

Defense: input escaping

```
<? 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.

**GET INPUT** | **Command executed**
--- | ---
www.example.com | whois 'www.example.com'
www.example.com; rm -rf/; | whois 'www.example.com rm -rf/;'

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

```
HTTP

Client (HTML, JavaScript)

Google

Server (PHP)

Database (SQL)
```
### 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>

- 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

Retrieve 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:

```
username | password
----------|----------
alice     | 01234    
bob       | 56789    
charlie   | 43210    
```

```
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:

```
username | password
----------|----------
alice     | 01234    
bob       | 56789    
charlie   | 43210    
```

```
−→ username | password
-----------|----------
eve        | 98765    
```

```
INSERT INTO user_accounts VALUES ('eve', 98765)
```

### Other SQL commands

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

- Semicolons separate commands:

  Example:

  ```
  INSERT INTO user_accounts VALUES ('eve', 98765);
  SELECT password FROM user_accounts
  WHERE username='eve'
  returns 98765
  ```
SQL injection: a simple example

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

login.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.

---

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

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

---

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';");
```