Lab 7 : Threats mitigation and attacks prevention

6COSC017W

Dr. Ayman El Hajjar

Week 10

Note

- **NOTE**- In this lab the three machines and the host machine. They are setup as below in my test environment:
- Your machine IP addresses might be different. At the time of writing the labs, my machines had the following IP addresses:
  1. The attacker machine - That is your Kali Linux machine 192.168.56.101
  2. A legitimate client - That is a the windows virtual machine 192.168.56.102
  3. The OWASP Web server -that is the vulnerable machine 192.168.56.104
- **NOTE**- In this lab, you are working on protecting your webserver from unauthorized access. To do this, all commands will be entered on the OWASP server terminal. When you check if you command is working, you do this using the browsers of both KALI or Windows depending on what you are testing. This will be explained in the lab.

Lab environment

Setup your lab environment first

- Before you start your lab, Take notes of the IP addresses of your three VMs.

Understanding ports

- When two computers communicate across the network, they need a way of making sure that information flows from somewhere to somewhere else.
- They do so using a connection each end of which is a socket. A socket is the term used to refer to a combination of an address and a port.
- While many network devices have what we might call physical ports that we plug ethernet cables into, the term port in IP networking refers to a designated, logical space where traffic or communication for a particular process or service is intended to go.
- Ports are given numbers from zero to 65,535. This allows many services to listen and communicate on one network interface
  - The port numbers are divided into three general ranges. Zero through 1,023 is called the well-known ports range.
  - 1,024 to 49,151 is called the registered ports range.
  - And the ports above that are called dynamic. The first range of ports is where many common services run.
  - For example, SSH usually runs on port 22. HTTPS on 443 and TP on 123 and so on.

Preventing Unauthorized access

- When we have services running on our system, chances are we want certain clients to be able to connect to them.
- But it's also likely that we want to prevent access to those systems by people who we don't intend to have connect to them.
- In order to make this happen, we can use a firewall which is software that monitors network traffic and, using sets of rules, decides whether to allow the traffic to pass through, to redirect it somewhere else, or to prevent it from reaching its intended destination.
- we will use ufw firewall on the vulnerable machine to protect our devices from unauthorized access.

ufw

- In a terminal, we first need to check if the firewall is active or not. Write the following command :

  ufw status

- If the status is inactive, we can enable the firewall and activate it on system startup by typing:

  ufw enable

- You can check again using

  ufw status

- Now this is quite simple, however now we blocked all remote connections, no remote systems will be able to connect (genuine users or malicious).
- Everything is blocked from the access externally across the network.
- If you try from another machine to access the web server now, you will not be able to as all external communication are note allowed.
- That's not what we want.
  - Let us first check on the vulnerable machine which ports are actually open. you can do this by typing **sudo lsof -i -P -n | grep LISTEN**
    - You can see there is quite a few open but since the firewall is on, all external connection will be blocked.
  - Let us first open HTTP (80) and HTTPS (443) ports. these needs to allow traffic since we have a web server. if you have noticed from the previous list of open ports, both are setup to use TCP.

    ufw allow 80/tcp

    ufw allow 443/tcp
  - If you check the status of the firewall, you will be able to see the new rules you created. You can use the variable numbered to number the rules.

    ufw status numbered
  - To delete the first one (80/tcp) you type

    ufw delete 1
  - To delete the second one (443/tcp) you will also need to type

    ufw delete 1

    as it is now the first rule and so on or
  - You can type **sudo ufw deny 80/tcp** to deny access

  This will reverse the rule you wrote to allow traffic from port 80 tcp. Or you can delete a specific rule by typing

  ufw delete deny 80/tcp

- We can specify rules in the basic format that we've seen with a port and a protocol, or we can get a little more specific, allowing or denying access to or from certain networks or addresses.

- Let's write a rule that allows HTTP and HTTPS access only from systems on our network.
  - To do that, you will need to use an app designation instead of an explicit coordinate protocol.
  - Some apps need to have more than one port configured and to easily add common apps, we can tell UFW which one we want to configure.
  - To check what apps are configured in ufw we can type

    ufw app list
  - We can see what apps UFW knows about with UFW app list.
  - We can get some more information with ufw app info and the app name. we want to allow our Apache web server to be accessed by the network domain only except the Kali Linux machine.
  - To check which protocol the app use we type

    ufw app info "Apache Full"
  - To allow access for our network only.

    ufw allow from 192.168.56.0/24 app "Apache Full"
  - In this lab we need to exclude the attack device ip address kali. 192.168.56.101 as they are all on the same network

    ufw deny from 192.168.56.101 to any app "Apache Full"

Rules cancel each other

- If you open a browser on Kali and try to access the OWASP server , you will find that you can still access it.
- This is because rules can cancel each other. What we just did is that we allows all traffic and then we blocked traffic from 101.
- Rules with smaller values precedes the other rules so blocked traffic from 192.168.56.101 is cancelled by the first rule when we allowed traffic from all the network.

```
root@owaspbwa:~# ufw allow from 192.168.56.0/24 to any app "Apache Full"
Rule added
root@owaspbwa:~# ufw deny from 192.168.56.101 to any app "Apache Full"
Rule added
root@owaspbwa:~# ufw status
Status: active

To                         Action      From
--                         ------      ----
Apache Full                ALLOW       192.168.56.0/24
Apache Full                DENY        192.168.56.101

root@owaspbwa:~#
```

Figure 1: UFW rules cancel each other

- Let us delete both rules.

  ufw delete 1

  ufw delete 1

  - **Note:** Another way of doing this is by deleting only rule 1. The rule that will remain is to deny 192.168.56.101. You can then create the rule to allow all traffic from 192.168.56.0/24

- Let us try this the other way around.
- We first block traffic from Kali IP.

  ufw deny from 192.168.56.101 to any app "Apache Full"

- We then allow traffic from all other IPs on the network.

  ufw allow from 192.168.56.0/24 app "Apache Full"

- If you open a browser on Kali now and try to access the OWASP server , you will find that you cannot access it anymore. You can still access it from your host machine or from your Windows machine.
- ufw stores rules in a series of files in ls -l /etc/ufw
- Administrators usually write those rules in a text file -specially if they have many rules defining the access control (internally and externally)
- Let us now delete all rules to continue with the next activity

IPTABLES

- Iptables is a software package that acts as a firewall.
- Rules for the Iptables firewall software can seem complex and difficult to understand.
- These rules are put together in arrangements called chains, and using these, packets are evaluated against rules one at a time.
- If a rule or condition matches the packet, whatever action is specified by that rule is taken.
- If a packet doesn't match, the next rule is evaluated against it, and so on, until there's a match, or until the end of the chain is reached.
- Chains have a default action to take if a packet gets all the way through without matching anything. Chains can refer to other chains, making it fairly easy to set up conditional flows.

- The standard chains are for:
  - input- information coming into the system.
  - forward- in the case where the system is doing that
  - routing for systems behind it
- There are also a few predefined actions that we can use either in rules, or as default actions. They are:
  - allow, which permits a packet to pass through.
  - Drop, which ignores traffic.
  - reject, which actively responds that traffic is refused.
    - The difference here is that the traffic being treated to a drop response will time out after a while, and a client whose traffic is being rejected will be immediately notified that a connection is not possible
- Let us take a look at the Iptables chains on our vulnerable machine.
- TO do so type:

  iptables -L -n

  AS you can see this is a long list of outputs. To scroll up you press on shift + Page UP (PUP) and to scroll down press on shift + Page Down (PgDown)

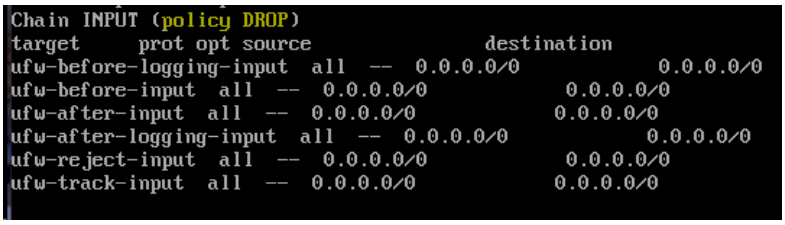

Figure 2: IPTables Input Policy Drop

- We can see in figure 2 above that the default policy for input is DROP
- If a packet makes it all the way through the rule chain without being matched and acted on, the firewall will drop the packet. The input chain has a list of rules on it
- The chain will be evaluated in order. As our packet goes down in the chain.
- Note: If you add any rule using ufw you will be able to see it. Try **sudo ufw deny from 192.168.56.101 to any app "Samba"** You can see this below in figure 3

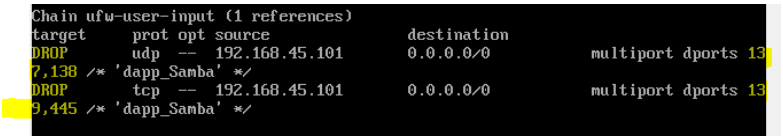

Figure 3: Samba UFW rules as seen in Iptables

- Samba works on port 137, 138, 139 and 445
- you can check these using the command we used before for
  - **sudo ufw app info "Samba"**
- The structure of an iptable is a complicated structure. To understand iptable commands structure type **man iptables**
- Before you continue VISIT THIS WEB PAGE (CLICK HERE) to see iptables with examples
- Aside from the rule for Samba already set in UFW let us write a rule that allows access through port 80 using TCP protocol so that users can access the website
- In order to allow the access we want, we need to make a rule that will match inbound traffic to port 80.
- To do that
  - We will start by the option -A to append the chain and the name of the chain. for example INPUT.
  - we can use the D-port, or destination port option, which we can take advantage of alongside the '-p' or protocol option.
  - we also want to allow TCP on only a specific port, so we use --dport option. **Note:** dport is the same as --destination-port option.
  - And then, in order to act on something that matches this rule, we need to use '-j,' or jump, and a target like accept. This could also be drop or return, or a different chain that's constructed to deal with special cases. The accept target tells the firewall to allow the packet through to wherever it's trying to get. So, with all these pieces, we'll put together an argument.
- Now we can allow web server traffic (shown below):
  - We added the two ports (http port 80, and https port 443) to the ACCEPT chain - allowing traffic in on those ports.
    - **sudo iptables -A INPUT -p tcp --dport 80 -j ACCEPT**
    - **sudo iptables -A INPUT -p tcp --dport 443 -j ACCEPT**
- Now let us deny web server traffic access from the attacker machine

- if you wish to block ip address 192.168.56.101 then type the command to port 80 as follows:

  sudo iptables -A INPUT -s 192.168.56.101 -p tcp --destination-port 80 -j DROP

- or port 443

  sudo iptables -A INPUT -s 192.168.56.101 -p tcp --destination-port 443 -j DROP

- Now let us deny all access from the attacker machine
  - if you wish to block ip address 192.168.56.101 then type the command as follows:

    sudo iptables -A INPUT -s 192.168.56.101 -j DROP

Save the configuration

- Now that we have all the configuration in, we can **list** the rules to see if anything is missing:

  sudo iptables -L -n

- Now we can finally **save** our firewall configuration:

  sudo iptables-save

Open the saved configuration file

- We can open the saved configuration file using nano tool:

  nano iptables-save

- Now we can finally **save** our firewall configuration: (shown below):

- You can always edit the text file and save it rather than entering rules via the command line:

  sudo iptables-save

Flush/Delete the configuration

- If you are looking to **remove** your iptables rules, you can flush them as shown below:

  sudo iptables -F

Flush/Delete the configuration

- If you are looking to **remove** a specific rule then you need to use -D option (To delete) instead of -A option (To Append) with the whole rule (exactly as it is written)
- For example if you wish to allow ip address 192.168.56.101 again to the vulnerable machine then type the command as follows:

  sudo iptables -D INPUT -s 192.168.56.101 -j DROP

READ THIS CAREFULLY

- All the activities from this point onwards on are only informative to explain different methods to protect against vulnerabilities you have identified in previous labs.

Preventing injection attacks

- According to OWASP, the most critical type of vulnerability found in Web applications is the injection of some type of code, such as SQL injection, OS command injection, HTML injection, and so on.
- These vulnerabilities are usually caused by a poor input validation by the application. In this activity, we will cover some of the best practices when processing user inputs and constructing queries that make use of them.
- The first thing to do in order to prevent injection attacks is to properly validate inputs. On the server side, this can be done by writing our own validation routines; although the best option is using the language's own validation routines, as they are more widely used and tested. A good example is filter_var in PHP or the validation helper in ASP.NET. For example, an e-mail validation in PHP would be similar to this:

```
function isValidEmail($email){
return filter_var($email, FILTER_VALIDATE_EMAIL);
}
```

- On the client side, validation can be achieved by creating JavaScript validation functions, using regular expressions. For example, an e-mail validation routine would be:

```
function isValidEmail (input)
{
var result=false;
var email_regex = /^[a-zA-Z0-9._-]+@([a-zA-Z0-9.-]+\.)+[azA-
Z0-9.-]{2,4}$/;
if ( email_regex.test(input) ) {
result = true;
}
return result;
}
```

- For SQL Injection, it is also useful to avoid concatenating input values to queries. Instead, use parameterized queries; each programming language has its own version:

PHP with MySQL Code

```
#PHP with MySQLi:
$query = $dbConnection->prepare('SELECT * FROM table
WHERE name =
?');
$query->bind_param('s', $name);
$query->execute();
```

C#

```
string sql = "SELECT * FROM Customers WHERE CustomerId = @
CustomerId";
SqlCommand command = new SqlCommand(sql);
command.Parameters.Add(new SqlParameter("@CustomerId", System.
Data.SqlDbType.Int));
command.Parameters["@CustomerId"].Value = 1;
```

Java

```
String custname = request.getParameter("customerName");
String query = "SELECT account_balance FROM user_data
 WHERE user_
name =? ";
PreparedStatement pstmt = connection.prepareStatement(
 query );
pstmt.setString( 1, custname);
ResultSet results = pstmt.executeQuery( );
```

- Considering the fact that an injection occurs, it is also useful to restrict the amount of damage that can be done. So, use a low-privileged system user to run the database and web servers.
- Make sure the user that the applications allow to connect to the database server is not a database administrator.
- Disable or even delete the stored procedures that allow an attacker to execute system commands or escalate privileges, such as xp_cmdshell in MS SQL Server.

How it works

- The main part of preventing any kind of code injection attack is always a proper input validation, both on the client-side and server-side.
- For SQL Injection also, always use parameterized or prepared queries instead of concatenating SQL sentences and inputs. Parameterized queries insert function parameters in specified places of an SQL sentence, eliminating the need for programmers to construct the query themselves, by concatenation.
- In this activity, we have used the language's built-in validation functions, but you can create your own if you need to validate some special type of input by using regular expressions.
- Apart from doing a correct validation, we also need to reduce the impact of the compromise in case somebody manages to inject some code. This is done by properly configuring a user's privileges in the context of an operating system for a Web server and for both database and OS in the context of a database server.

Building proper authentication and session management

- Flawed authentication and session management are the second most critical vulnerability in web applications nowadays.
- Authentication is the process whereby users prove that they are who they say they are; this is usually done through usernames and passwords. Some common flaws in this area are permissive password policies and security through obscurity (lack of authentication in supposedly hidden resources).
- Session management is the handling of session identifiers of logged users; in Web servers this is done by implementing session cookies and tokens. These identifiers can be implanted, stolen, or "hijacked" by attackers by social engineering, cross-site scripting or CSRF, and so on. Hence, a developer must pay special attention to how this information is managed.
- In this activity, we will cover some of the best practices when implementing username/password authentication and to manage the session identifiers of logged users.
- If there is a page, form, or any piece of information in the application that should be viewed only by authorized users, make sure that a proper authentication is done before showing it.
- Make sure usernames, IDs, passwords, and all other authentication data are case-sensitive and unique for each user.

- Establish a strong password policy that forces the users to create passwords that fulfill, at least, the following requirements:
  - More than 8 characters, preferably 10.
  - Use of upper-case and lower-case letters.
  - se of at least one numeric character (0-9).
  - se of at least one special character (space, !, &, #, %, and so on).
  - orbid the username, site name, company name, or their variations (changed case, l33t, fragments of them) to be used as passwords.
  - Forbid the use of passwords in the "Most common passwords" list:https://www.teamsid.com/wor passwords-2015/
  - Never specify in an error message if a user exists or not or if the information has the correct format. Use the same generic message for incorrect login attempts, non-existent users, names or passwords not matching the pattern, and all other possible login errors. Such a message could be:
    - Login data is incorrect.
    - Invalid username or password.
    - Access denied.
- Passwords must not be stored in clear-text format in the database; use a strong hashing algorithm, such as SHA-2, scrypt, or bcrypt, which is especially made to be hard to crack with GPUs.
- When comparing a user input against the password for login, hash the user input and then compare both hashing strings. Never decrypt the passwords for comparison with a clear text user input.
- Avoid Basic HTML authentication.
- When possible, use multi-factor authentication (MFA), which means using more than one authentication factor to login:
  - Something you know (account details or passwords)
  - Something you have (tokens or mobile phones)
  - Something you are (biometrics)
- Implement the use of certificates, pre-shared keys, or other passwordless authentication protocols (OAuth2, OpenID, SAML, or FIDO) when possible.
- When it comes to session management, it is recommended that you use the language's built-in session management system, Java, ASP.NET, and PHP. They are not perfect, but surely provide a well designed and widely tested mechanism and they are easier to implement than any homemade version a development team, worried by release dates, could make.

- Always use HTTPS for login and logged in pages—obviously, by avoiding the use of SSL and only accepting TLS v1.1, or later, connections.
- To ensure the use of HTTPS, HTTP Strict Transport Security (HSTS) can be used. It is an opt-in security feature specified by the web application through the use of the Strict-Transport-Security header; it redirects to the secure option when http:// is used in the URL and prevents the overriding of the "invalid certificate" message, for example, the one that shows when using Burp Suite. For more information, you could check: https://www.owasp.org/index.php/HTTP_Strict_ Transport_Security
- Always set HTTPOnly and Secure cookies' attributes.
- . Set reduced, but realistic session expiration times. Not so long that an attacker may be able to reuse a session when the legitimate user leaves, and not so short that the user doesn't have the opportunity to perform the tasks the application is intended to perform.

How it works

- Authentication mechanisms in Web applications are very often reduced to a username/ password login page. Although not the most secure option, it is the easiest for users and developers; and when dealing with passwords, their most important aspect is their strength.
- As we have seen throughout this book, the strength of a password is given by how hard it is to break, be it by brute force, dictionary, or guessing. The first tips in this recipe are meant to make passwords harder to brute-force by establishing a minimum length and using mixed character sets, harder to guess by eliminating the more intuitive choices (user name, most common passwords, company name); and harder to break if leaked, by using strong hashing or encryption when storing them.
- As for session management: the expiration times, uniqueness, and strength of session ID (already implemented in the language's in-built mechanisms), and security in cookie settings are the key considerations.
- The most important aspect when talking about authentication security probably, is that no security configuration or control or strong password is secure enough if it can be intercepted and read through a man in the middle attack; so, the use of a properly configured encrypted communication channel, such as TLS, is vital to keep our users' authentication data secure.

Preventing cross-site scripting

- Cross-site scripting, as seen previously, happens when the data shown to the user is not correctly encoded and the browser interprets it as a script code and executes it. This also has an input validation factor, as a malicious code is usually inserted through input variables.
- In this Activity, we will cover the input validation and output encoding required for developers to prevent XSS vulnerabilities in their applications.
- The first sign of an application being vulnerable to XSS is that in the page it reflects the exact input given by the user. So, try not to use user-given information to build output text.
- When you need to put user-provided data in the output page, validate such data to prevent the insertion of any type of code. We already saw how to do that in the Preventing injection attacks section.
- If, for some reason, the user is allowed to input special characters or code fragments, sanitize or properly encode the text before inserting it in the output.
- For sanitization, in PHP, filter_var can be used; for example, if you want to have only e-mail valid characters in the string:

```
<?php
$email = "john(.doe)@exa//mple.com";
$email = filter_var($email, FILTER_SANITIZE_EMAIL);
echo $email;
?>
For encoding, you can use htmlspecialchars in PHP:
<?php
$str = "The JavaScript HTML tags are <script> for opening, and </
script> for closing.";
echo htmlspecialchars($str);
?>
```

- In .NET, for 4.5 and later implementations, the System.Web.Security.AntiXss namespace provides the necessary tools. For .NET Framework 4 and prior, we can use the Web Protection library: http://wpl.codeplex.com/.
- Also, to prevent stored XSS, encode or sanitize every piece of information before storing it and retrieving it from the database.
- Don't overlook headers, titles, CSS, and script sections of the page, as they are susceptible of being exploited too.

How it works

- Apart from a proper input validation and not using user inputs as output information, sanitization and encoding are key aspects in preventing XSS.
- Sanitization means removing the characters that are not allowed from the string; this is useful when no special characters should exist in input strings.
- Encoding converts special characters to their HTML code representations; for example, "&" to "&" or "¡" to "<". Some applications allow the use of special characters in input strings; for them sanitization is not an option, so they should encode the inputs before inserting them into the page and storing them in the database.

Basic security configuration guide

- Default configurations of systems, including operating systems and Web servers, are mostly created to demonstrate and highlight their basic or most relevant features, not to be secure or protect them from attacks.
- Some common default configurations that may compromise the security are the default administrator accounts created when the database, web server, or CMS was installed, and the default administration pages, default error messages with stack traces, among many others.
- In this activity, we will cover an important critical vulnerability, Security Misconfiguration.
- If possible, delete all the administrative applications such as Joomla's admin, WordPress' admin, PhpMyAdmin, or Tomcat Manager. If that is not possible, make them accessible from the local network only; for example, to deny access from outside networks to PhpMyAdmin in an Apache server, modify the httpd.conf file (or the corresponding site configuration file):

```
<Directory /var/www/phpmyadmin>
Order Deny,Allow
Deny from all
Allow from 127.0.0.1 ::1
Allow from localhost
Allow from 192.168
Satisfy Any
</Directory>
```

- This will first deny access from all addresses to the phpmyadmin directory; second, it will allow any request from the localhost and addresses beginning with "192.168", which are local network addresses.
- Change all administrators' passwords from all CMSs, applications, databases, servers, and frameworks with others that are strong enough. Some examples of these applications are:
  - Cpanel
  - Joomla
  - WordPress
  - PhpMyAdmin
  - Tomcat manager
- Disable all unnecessary or unused server and application features. On a daily or weekly basis, new vulnerabilities are appearing on CMSs' optional modules and plugins. If your application doesn't require them, there is no need to have them active.
- Always have the latest security patches and updates. In production environments, it may be necessary to set up test environments to prevent failures that leave the site inoperative because of compatibility issues with the updated version or other problems.
- Set up custom error pages that don't reveal tracing information, software versions, programming component names, or any other debugging information. If developers need to keep a record of errors or some identifier is necessary for technical support, create an index with a simple ID and the error's description and show only the ID to the user. So when the error is reported to a support personnel, they will check the index and will know what type of error it was.
- Adopt the "Principle of least privilege". Every user, at every level (operating system, database, or application), should only be able to access the information strictly required for a correct operation, never more.
- Taking into account the previous points, build a security configuration baseline and apply it to every new implementation, update or release, and to current systems.
- Enforce periodic security testing or auditing to help detect misconfigurations or missing patches

How it works

- Talking about security and configuration issues, we are correct if we say "The devil is in the detail." The configuration of a web server, a database server, a CMS, or an application should find the point of equilibrium between being completely usable and useful and being secure for both users and owners.
- One of the most common misconfigurations in a Web application is that there is some kind of a Web administration site accessible to all of the Internet; this may not seem such a big issue, but we should know that an admin login page is much more attractive to crooks that any web-mail as the former gives access to a much higher privilege level and there are lists of known, common, and default passwords for almost every CMS, database, or site administration tool we can think of. So, our first recommendations are in the sense of not exposing these administrative sites to the world and removing them if possible.
- Also, the use of a strong password and changing those that are installed by default (even if they are "strong") is mandatory when publishing an application to the internal company network and much more so to the Internet. Nowadays, when we expose a server to the world, the first traffic it receives is port scans, login page requests, and login attempts; even before the first user knows the application is active.

Redirect validation

- Unvalidated redirects and forwards is the tenth most critical security issue for web applications according to OWASP; it happens when an application takes a URL or an internal page as a parameter to perform a redirect or forward operation. If the parameter is not correctly validated, an attacker could abuse it making it to redirect to a malicious Web site.
- In this activity we will see how to validate that the parameter we receive for redirection or forwarding is the one that we intend to have when we develop the application.
    1.
    2. Don't want to be vulnerable? Don't use it. Whenever it's possible, avoid the use of redirects and forwards.
    3. If it is necessary to make a redirection, try not to use user-provided parameters (request variables) to calculate the destination.
    4. If the use of parameters is required, implement a table that works as a catalog of redirections, using an ID instead of a URL as the parameter the user should provide.

5. Always validate the inputs that will be involved in a redirect or forward operation; use regular expressions or whitelists to check that the value provided is a valid one.

How it works

- Redirects and forwards are one of the favorite tools of phishers and other social engineers and sometimes we don't have any control over the security of the destination; so, even when it is not our application, a security compromise on that part may affect us in terms of reputation. That's why the best choice is not to use them.
- If the said redirect is to a known site, such as Facebook or Google, it is possible that we can establish the destinations in a configuration file or a database table and have no need of a client-provided parameter to do it.
- If we build a database table containing all the allowed redirect and forward URLs, each one with an ID, we can ask for the ID as parameter instead of the destination itself. This is a form of whitelist that prevents the insertion of forbidden destinations.
- Finally, and again, validation. It is very important that we always validate every input from the client, as we don't know what we can expect from our users. If we validate correctly the destination of a redirect, we can prevent, besides a malicious forward or redirect, a possible SQL Injection, XSS, or Directory Traversal. Hence, it's relevant.