Laboratorio de Race Condition

1 Descripción General

El objetivo de este laboratorio es que el estudiante gane experiencia en vulnerabilidades de Race Condition, pudiendo pasar de la teoría a la práctica. Una vulnerabilidad de Race Condition ocurre cuando múltiples procesos quieren acceder y manipular un determinado dato al mismo tiempo, el resultado de la ejecución será indeterminado, ya que estará dado por el orden en que esos procesos realizaron el acceso al dato en cuestión. Si un programa tiene una vulnerabilidad del tipo Race Condition, un atacante puede correr un conjunto de procesos en paralelo para hacer un "race" contra el programa vulnerable, con la intención de alterar su comportamiento.

Para este laboratorio, los estudiantes tendrán disponible un programa con una vulnerabilidad del tipo Race Condition; La tarea será desarrollar un exploit para explotar esta vulnerabilidad y obtener privilegios de root. A su vez cada uno de los estudiantes serán guiados a fin de que puedan atravesar diferentes tipos mitigaciones que son usadas para contrarrestar los ataques de Race Condition. Los estudiantes necesitarán evaluar en que situaciones sus ataques serán exitosos y en cuales no, así también explicando el porque de este resultado. Este laboratorio cubre los siguientes tópicos:

- Vulnerabilidad de Race condition
- Protecciones: Sticky symlink
- Principio del menor privilegio

Lecturas y Videos. Para una cobertura más detallada en Ataques de Race Condition puede consultar

- Capítulo 7 del Libro de SEED, *Computer & Internet Security: A Hands-on Approach*, 2nd Edition, by Wenliang Du. See details at `https://www.handsonsecurity.net`.
- Sección 6 del curso de SEED en Udemy, *Computer Security: A Hands-on Approach*, by Wenliang Du. See details at `https://www.handsonsecurity.net/video.html`.

Tópicos relacionados. Existen tres laboratorios relacionados a Race Condition. Uno es el laboratorio de Dirty COW, que explota una vulnerabilidad de Race Condition dentro del kernel del sistema operativo (Capítulo 8 del libro de SEED cubre este ataque). Los otros dos son los laboratorios de Meltdown y Spectre (Capítulos 13 y 14 del libro de SEED). Estos últimos explotan una vulnerabilidad de Race Condition dentro del CPU. Estos cuatro laboratorios proveen una cobertura integral del problema de la vulnerabilidad de Race Condition en diferentes niveles: a nivel aplicación, a nivel kernel y finalmente a nivel hardware.

Entorno de Laboratorio. Este laboratorio ha sido testeado en nuestra imagen pre-compilada de una VM con Ubuntu 20.04, que puede ser descargada del sitio oficial de SEED. Sin embargo, la mayoría de nuestros laboratorios pueden ser realizados en la nube para esto Ud. puede leer nuestra guía que explica como crear una VM de SEED en la nube.

2 Configuración del Entorno de Laboratorio

2.1 Desactivando las Contramedidas

Ubuntu viene con mecanismos para proteger a los programas frente a los ataques de Race Condition. Uno de ellos funciona restringiendo el acceso a los links simbólicos dentro de directorios que poseen el sticky bit activado (por ejemplo `/tmp`) permitiendo el acceso solamente al creador del link si el directorio no fue creado por quién es el dueño de este link simbólico. Otro mecanismo de seguridad que fue implementado en Ubuntu 20.04, evita que el usuario root realice cambios en archivos dentro del directorio `/tmp`, que no fueron creados por este usuario (es decir el usuario root). Como primer paso necesitamos desactivar estas protecciones, ejecutando los siguientes comandos:

```
// On Ubuntu 20.04, use the following:
$ sudo sysctl -w fs.protected_symlinks=0
$ sudo sysctl fs.protected_regular=0

// On Ubuntu 16.04, use the following:
$ sudo sysctl -w fs.protected_symlinks=0

// On Ubuntu 12.04, use the following:
$ sudo sysctl -w kernel.yama.protected_sticky_symlinks=0
```

2.2 El Programa Vulnerable

El siguiente programa es en apariencia inofensivo. Pero contiene una vulnerabilidad de Race Condition.

Listing 1: The vulnerable program (`vulp.c`)

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

int main()
{
   char* fn = "/tmp/XYZ";
   char buffer[60];
   FILE* fp;

   /* get user input */
   scanf("%50s", buffer );

   if(!access(fn, W_OK)){                  ①
      fp = fopen(fn, "a+");                ②
      fwrite("\n", sizeof(char), 1, fp);
      fwrite(buffer, sizeof(char), strlen(buffer), fp);
      fclose(fp);
   }
   else printf("No permission \n");
}
```

El programa de arriba tiene como dueño al usuario root y es un programa `Set-UID`; Este programa agrega una cadena ingresada como input por parte del usuario al final del archivo temporal `/tmp/XYZ`. Dado que este programa corre con privilegios de root (es decir su effective user ID es cero), puede sobreescribir cualquier archivo. Para evitar que accidentalmente sobreescriba los archivos de otros usuarios, lo

primero que hace este programa es chequear si el real user ID tiene permisos sobre el archivo `/tmp/XYZ`; para este propósito se usa la función `access()` en la línea ①. Si el real user ID es el correcto y puede acceder al archivo, el programa abre el archivo (Línea ②) y procede a escribir la cadena que ingresó el usuario como input.

A primera vista el programa parece no tener ningún tipo de problemas. Sin embargo contiene una vulnerabilidad de Race Condition: Debido a que existe una ventana de tiempo entre el chequeo realizado por la función (`access`) y el uso de (`fopen`), existe la posibilidad que el archivo usado en la función `access()` difiera del usado por la función `fopen()`, aunque ambos sean el mismo `/tmp/XYZ`. Si un atacante malicioso dentro de esa ventana de tiempo logra de alguna forma modificar el link simbólico de `/tmp/XYZ` a un archivo privilegiado como lo es `/etc/passwd`, podría alterar su contenido con el input ingresado en el programa, obteniendo así privilegios de root. Esto es posible debido a que el programa vulnerable corre con los privilegios del usuario root.

Configurando nuestro programa `Set-UID` Primero debemos compilar el código dado a continuación y activar el bit de `Set-UID` en nuestro programa cuyo dueño será el usuario root. Para lograr esto se deben ejecutar los siguientes comandos:

```
$ gcc vulp.c -o vulp
$ sudo chown root vulp
$ sudo chmod 4755 vulp
```

3 Tarea 1: Eligiendo nuestro Target

Para proceder a explotar la vulnerabilidad de Race Condition dentro del programa, vamos a elegir el archivo `/etc/passwd` como nuestro target, este archivo no puede ser escrito por usuarios no privilegiados. Explotando esta vulnerabilidad nuestro objetivo final será agregar un registro en este archivo, que será un usuario que tendrá privilegios de root. Dentro del archivo `/etc/passwd`, cada línea del mismo es una entrada y cada entrada representa un usuario del sistema, cada usuario tiene tiene campos que están separados por el caracter de dos puntos (:). La entrada para el usuario root se muestra a continuación:

```
root:x:0:0:root:/root:/bin/bash
```

En esta entrada, el tercer campo de este usuario (el campo de user ID) tiene un valor de cero. Esto significa que cada vez que este usuario se loguee en el sistema su id de proceso será siempre cero y es el valor de cero lo que le otorgará privilegios de root. Dicho de otro modo no es el nombre de usuario el que determina sus privilegios sino su user ID. Si quisieramos crear un usuario con privilegios de root, debemos de poner un cero en este campo.

Cada entrada de este archivo, contiene un campo de password, este es el segundo campo. En el ejemplo anterior este campo tiene el valor de `"x"` lo que indica que el password para este usuario esta siendo guardado en otro archivo llamado `/etc/shadow` (shadow file). Si somos fieles a este ejemplo entonces al aprovecharnos de la vulnerabilidad de Race Condition, deberíamos de modifiar dos archivos, el archivo password y el archivo shadow, lo cual no es difícil de hacer pero existe una forma más sencilla. En vez de poner el valor `"x"` en el campo de la entrada que corresponde al usuario, podemos poner un password de esta manera el sistema operativo no irá a buscarlo al archivo shadow.

El campo de password no contiene el password actual en texto plano; este contiene el password hasheado a través de un algoritmo de hashing de una sóla vía (one-way hashing). Para obtener este valor, se puede agregar un usuario nuevo al sistema usando el comando `adduser` y obtener su hash correspondiente desde

el archivo shadow o podríamos simplemente copiar el valor de la entrada del usuario `seed` ya que sabemos que su password es `dees`. Hay algo interesante para mencionar, existe un valor mágico para las cuentas sin password que es usado por `Ubuntu` al ser booteado usando el modo live CD, este valor mágico es `U6aMy0wojraho` (no confundir el sexto caracter con la letra `O` ya que es un cero). Si se coloca este valor en el campo del password para un usuario cualquiera, estaríamos usando un usuario sin password, de forma tal que podríamos loguearnos con un enter al momento que el prompt del sistema nos pregunte por el password.

Para verificar si el password mágico funciona o no. Necesitamos agregar siguiente entrada como usuario privilegiado al final del archivo `/etc/passwd`.

Tarea. Para verificar si el password mágico funciona o no. Necesitamos agregar siguiente entrada como usuario privilegiado al final del archivo `/etc/passwd`. Por favor reporte si puede loguearse en el sistema usando el usuario `test` sin tipear ningún password y si tiene privilegios de root.

```
test:U6aMy0wojraho:0:0:test:/root:/bin/bash
```

Después de hacer esta tarea, elimine la entrada agregada anteriormente en el archivo `/etc/passwd`. En la próxima tarea, haremos que esta entrada sea añadida como un usuario normal y no como un usuario privilegiado. Claramente esto no es posible en una situación normal, pero lo haremos explotando la vulnerabilidad de Race Condition de nuestro programa vulnerable que correrá con privilegios de root.

Advertencia. En el pasado, muchos estudiantes al ejecutar el ataque han vaciado accidentalmente el archivo `/etc/passwd` (esto puede ocurrir porque se pueden dar race conditions a nivel kernel). Si Ud. pierde este archivo, no podrá loguearse en el sistema. Para evitar esta situación, recomendamos que se haga una copia del archivo original o se realize un snapshot de la Máquina Virtual, de esta manera se podrá volver el estado anterior del sistema ante cualquier problema que pueda surgir.

4 Tarea 2: Lanzando el Ataque de Race Condition

El objetivo de este ataque es explotar la vulnerabilidad de Race Condition en nuestro programa vulnerable con el bit de `Set-UID` activado. La meta final es obtener privilegios de root. La parte más crítica de este ataque es hacer que el archivo `/tmp/XYZ` apunte al archivo de password, dentro de la ventana tiempo que existe entre la invocación de las funciones `access` y `fopen`.

4.1 Tarea 2.A: Simulando una Máquina Lenta

Haremos de cuenta que nuestro programa se encuentra corriendo en una máquina lenta, y hay un lapso de tiempo de 10 segundos después de la llamada a la función `access()`. Para simular esto, usaremos la función `sleep(10)` de la siguiente manera:

```
if (!access(fn, W_OK)) {
     sleep(10);
     fp = fopen(fn, "a+");
     ...
```

Al recompilar el programa `vulp` una vez agregado este cambio en el código, este será pausado y le cederá el control al sistema operativo durante 10 segundos. Su tarea es hacer algo de forma manual, de forma tal que cuando el programa retome la ejecución (después de 10 segundos), el programa lo ayude a agregar una cuenta root en el sistema. Por favor demuestre como logró hacer esto.

No será posible modificar el archivo `/tmp/XYZ`, ya que está hardcodeado dentro del código, pero puede cambiar la referencia de su link simbólico. Por ejemplo, puede hacer un link simbólico de `/tmp/XYZ` a `/dev/null`, de esta forma el contenido que se ingrese y se quiera escribir en `/tmp/XYZ` terminará escribiéndose en `/dev/null`. En el siguiente ejemplo se muestra como crear este link, (la opción `"f"` removerá todo link simbólico existente sobre un archivo si es que existe alguno y procederá a crear uno nuevo)

```
$ ln -sf /dev/null /tmp/XYZ
$ ls -ld /tmp/XYZ
lrwxrwxrwx 1 seed seed 9 Dec 25 22:20 /tmp/XYZ -> /dev/null
```

4.2 Tarea 2.B: El Ataque Real

En la tarea anterior, se puede decir que de alguna forma hemos hecho "trampa" haciendo que el programa corra más lento para poder así lanzar el ataque. Este tipo de escenario definitivamente no es realista. En esta sección procederemos a lanzar el ataque real. Antes de hacer esta tarea, asegurése de eliminar la línea que contiene la función `sleep()` del programa `vulp`.

La estrategia más típica es correr el ataque de forma paralela al programa vulnerable, esperando poder hacer esa operación crítica dentro de la ventana de tiempo. Desafortunadamente, es muy difícil poder lograr una sincronización perfecta para esto y es por ello que el éxito del ataque es una cuestión probabilística. La probabilidad de que el ataque sea exitoso es baja si esa ventana de tiempo es corta, pero podemos correr el ataque reiteradas veces para incrementar nuestras posibilidades de éxito. Sólo necesitamos explotar la Race Condition una sóla vez.

Escribiendo el programa de ataque. En el ataque simulado de la sección anterior hemos usado `"ln -s"` para modificar el link simbólico. Ahora debemos hacerlo dentro del programa, podemos usar la función de C `symlink()` para crear este link simbólico. Dado que `Linux` no permite crear un link si ya existe uno, primero debemos eliminar el existente. El siguiente fragmento de código en C muestra como eliminar un link existente y crear uno nuevo en el archivo `/tmp/XYZ` hacia `/etc/passwd`. Por favor escriba el programa de ataque.

```
unlink("/tmp/XYZ");
symlink("/etc/passwd","/tmp/XYZ");
```

Corriendo el programa vulnerable y monitoreando los resultados. Debido a que necesitamos correr el programa vulnerable muchas veces, escribiremos un script para automatizar este processo. Para evitar escribir a mano el input que recibe el programa vulnerable `vulp`, podemos usar input redirection. Es decir, crearemos un archivo con nuestro input y obtendremos su contenido desde el programa vulnerable usando `"vulp < inputFile"`. También podemos usar un pipe (se dará un ejemplo de esto más adelante)

Como se mencionó anterioremente, puede pasar un tiempo hasta que nuestro ataque sea exitoso y pueda modificar el archivo de password, es por esto que necesitamos contar algún tipo de mecanismo para detectar si nuestro ataque fue exitoso o no. Hay muchas formas de lograr esto; la más fácil es monitorear el timestamp del archivo. El siguiente shell script corre el comando `"ls -l"` que imprime información del archivo, incluyendo su fecha de modificación. Comparando esta información con la obtenida anteriormente, podemos determinar si el archivo fue modificado o no.

El siguiente shell script ejecuta un loop que se encarga de correr el programa vulnerable (`vulp`) con el input producido por el comando `echo` (vía un pipe). Es su tarea decidir cual será el contenido de este

input. Si el ataque es exitoso es decir si el archivo `passwd` es modificado, el script detendrá su ejecución. Necesita tener paciencia, normalmente suele tomar 5 minutos.

```
#!/bin/bash

CHECK_FILE="ls -l /etc/passwd"
old=$($CHECK_FILE)
new=$($CHECK_FILE)
while [ "$old" == "$new" ]    ← Check if /etc/passwd is modified
do
   echo "your input" | ./vulp ← Run the vulnerable program
   new=$($CHECK_FILE)
done
echo "STOP... The passwd file has been changed"
```

Verificando el éxito Cuando el script termine su ejecución, verifique que todo haya ido bien, logueándose como usuario test y verificando que tenga privilegios de root. Una vez hecho esto termine el proceso del script presionando `Ctrl-C` en la ventana de la terminal donde ejecutó el ataque.

Nota. Si después de 10 minutos, el ataque no tiene éxito, detenga el ataque y chequee quien es el dueño del archivo `/tmp/XYZ`. Si el dueño de este archivo es el usuario root, borre el archivo manualmente y lance el ataque nuevamente hasta que tenga éxito. Por favor documente su observación en el informe del laboratorio. En la Tarea 2.C, explicaremos el porque de este comportamiento y mejoraremos el ataque.

4.3 Tarea 2.C: Mejorando el Método de Ataque

En la Tarea 2.B, puede ser que el ataque no haya funcionado, aún si se realizó todo bien, verifique quien es el dueño del archivo `/tmp/XYZ`. Encontrará que el dueño del archivo `/tmp/XYZ` es el usuario root (normalmente debería de ser el usuario `seed`).

Si esto sucede, su ataque jamás funcionará. Esto se debe a que el programa de ataque corre con los privilegios del usuario `seed` por ende no puede operar sobre el archivo `/tmp/XYZ` a través de la función `unlink()`. Esto es porque el directorio `/tmp` tiene activado el "sticky" bit, esto quiere decir que sólo el dueño del archivo puede borrarlo, aunque la carpeta tenga permisos de escritura para todos los usuarios del sistema.

En la Tarea 2.B, le hemos permitido usar los privilegios de root para borrar el archivo `/tmp/XYZ` y lanzar el ataque nuevamente. Esta condición no deseada ocurre de forma aleatoria por lo tanto repitiendo el ataque una y otra vez (con la "ayuda" del usuario root) puede que hacer que eventualmente el ataque funcione. Obviamente un escenario en donde el usuario root nos provee la ayuda para ser exitosos en el ataque no es un escenario realista, es por eso que queremos evitar esta dependencia y lograr el éxito del ataque sin la ayuda del root.

La razón principal de esta condición no deseada en nuestro programa de ataque, está dada por un problema dentro del mismo, un problema de Race Condition que es exactamente el mismo problema que estamos tratando de explotar en el programa vulnerable, ¡Que ironía! En el pasado, cuando descubrimos esta situación, nuestros estudiantes eran notificados y se les decía que debían borrar el archivo y tratar el ataque nuevamente. Gracias a uno de mis estudiantes, quién determinó cual era el problema, pudimos entender lo que estaba pasando y trabajar en una solución que mejore el ataque.

El problema radica en que el programa de ataque hace un cambio de contexto justo después de haber borrado el archivo `/tmp/XYZ` (usando `unlink()`), pero antes de crear el link a otro archivo (usando

`symlink()`). Hay que recordar que las acciones de borrado y creación de link simbólicos no son atómicas (estas requieren de dos llamadas al sistema por separado), por lo tanto si un cambio de contexto ocurre en el medio (es decir después de borrar `/tmp/XYZ`) y el programa vulnerable `Set-UID` logra ejecutar `fopen(fn, "a+")`, se creará un nuevo archivo donde el usuario root será el dueño. Una vez ocurrido este escenario el programa de ataque no puede hacer cambios en el archivo `/tmp/XYZ` dado que no es el dueño.

Básicamente, el resultado de usar las funciones `unlink()` y `symlink()` produce en nuestro programa de ataque una Race Condition. Por lo tanto mientras tratamos de explotar la Race Condition de nuestro programa vulnerable, este de forma accidental "explota" una Race Condition en nuestro programa de ataque, contrarrestándo la explotación deseada.

Para resolver este problema, necesitamos que las funciones `unlink()` y `symlink()` sean atómicas. Afortunadamente, existe una llamada al sistema que nos permite lograr esto. Siendo más preciso nos permite intercambiar dos links simbólicos de forma atómica. El siguiente programa crea dos links simbólicos `/tmp/XYZ` y `/tmp/ABC`, usando la llamada al sistema `renameat2` que los intercambia atómicamente. Esto nos permite cambiar a donde apunta `/tmp/XYZ` evitando introducir una Race Condition.

```
#define _GNU_SOURCE

#include <stdio.h>
#include <unistd.h>
int main()
{
   unsigned int flags = RENAME_EXCHANGE;

   unlink("/tmp/XYZ"); symlink("/dev/null",   "/tmp/XYZ");
   unlink("/tmp/ABC"); symlink("/etc/passwd", "/tmp/ABC");

   renameat2(0, "/tmp/XYZ", 0, "/tmp/ABC", flags);
   return 0;
}
```

Tareas. Por favor realice el ataque nuevamente con esta nueva estrategia, si todo es hecho de manera correcta, su ataque debería de ser exitoso.

5 Tarea 3: Contramedidas

5.1 Tarea 3.A: Aplicando el Principio del Menor Privilegio

El problema fundamental del programa vulnerable mostrado en este laboratorio es que viola el *Principio del Menor Privilegio*. El programador sabe que quien ejecute este programa puede llegar a ser un usuario potencialmente privilegiado y es ese el sentido del chequeo que se realiza a través de la función `access()` limitar sus privilegios de acceso. Sin embargo esta óptica no es la correcta, un mejor enfoque es aplicar el *Principio del Menor Privilegio* este declara que si un usuario no necesita determinado privilegio, ese privilegio no le debe de ser otorgado.

Podemos usar la llamada al sistema `seteuid` para desactivar temporalmente el privilegio de root y luego activarlo si es necesario. Por favor use esta estrategia para corregir la vulnerabilidad de nuestro programa y repita el ataque. ¿Podrá realizar un ataque exitoso? Por favor reporte sus observaciones y provea las respectivas explicaciones al respecto.

5.2 Tarea 3.B: Usando las protecciones nativas de `Ubuntu`

Las versión de Ubuntu 10.10 y posteriores incluyen protecciones nativas para prevenir ataques a vulnerabilidades de Race Condition. En esta tarea, ud. deberá de activar estas protecciones usando los siguientes comandos:

```
// On Ubuntu 16.04 and 20.04, use the following command:
$ sudo sysctl -w fs.protected_symlinks=1

// On Ubuntu 12.04, use the following command:
$ sudo sysctl -w kernel.yama.protected_sticky_symlinks=1
```

Realice su ataque después de activar estas protecciones. Por favor describa sus observaciones y explique los siguientes puntos: (1) ¿Cómo funcionan estas protecciones? (2) ¿Cuales son sus limitaciones?

6 Informe del Laboratorio

Debe enviar un informe de laboratorio detallado, con capturas de pantalla, para describir lo que ha hecho y lo que ha observado. También debe proporcionar una explicación a las observaciones que sean interesantes o sorprendentes. Enumere también los fragmentos de código más importantes seguidos de una explicación. No recibirán créditos aquellos fragmentos de códigos que no sean explicados.

Agradecimientos

Este documento ha sido traducido al Español por Facundo Fontana