



Sistemas antiforenses de post-explotación

Albert Puigsech, Rubén Garcia

apuigsech@7a69ezine.org , rubeng@7a69ezine.org

1. Introducción

Las técnicas actuales de explotación de servicios remotos han avanzado considerablemente, de la misma forma, la gama de shellcodes utilizadas ha crecido e incorporado complejas funcionalidades de anti-detección, como puede ser el polimorfismo.

A pesar de que todo ello facilita mucho el trabajo de los atacantes, al final siempre es necesaria una llamada a la syscall `execve`, lo que plantea una serie de problemas.

- El acceso a la syscall `execve` puede estar denegado si el host usa algún sistema de protección moderno.
- La llamada a `execve` requiere que el fichero a ejecutar este en el disco, por lo que si `"/bin/sh"` no existe (algo muy común en entornos chroot) la shellcode no se ejecuta correctamente.
- El host no dispondrá de herramientas que el intruso pueda necesitar, por lo que será necesario subirlas, lo que dejará huellas en el disco del host.

Se plantea entonces la necesidad de una shellcode capaz de solventar estos problemas. La solución la encontramos en el `'userland exec'`.

2. Userland Execve

El procedimiento que permite la ejecución local de un programa sin utilizar la syscall `execve` recibe el nombre de `'userland exec'` o `'userland execve'`.

Consiste en un mecanismo que simula de forma correcta y ordenada la mayor parte de las tareas que el núcleo realiza para la carga en memoria de un fichero ejecutable e iniciar su ejecución. Este proceso puede llegar a resumirse en tres pasos.

- Carga de las secciones necesarias del binario a la memoria.
- Inicialización del contexto de la pila.
- Salto al punto de partida o entry point.

El objetivo principal del `'userland exec'` es permitir la carga de binarios prescindiendo totalmente de la syscall `execve` que proporciona el núcleo del sistema, lo que nos soluciona el primero de los problemas planteados. Así mismo, al tratarse de una

implementación propia, podemos adaptar el funcionamiento a nuestras necesidades permitiendo que el fichero ELF no sea leído del disco, sino de algún otro sitio, como puede ser un socket, de manera que solucionamos los otros dos problemas; no es necesario que el fichero `"/bin/sh"` esté visible por el proceso explotado porque podemos leerlo de la red, y por otra parte podremos ejecutar herramientas que no estén el host de destino.

La primera implementación pública de un `execve` en espacio de usuario fue realizada por "the grugq" [1], su funcionamiento y codificación son impecables, pero tiene algunas desventajas.

- No sirve para realizar ataques reales.
- El código es demasiado extenso y difícil de portar.

Debido a ello decidimos ponernos manos a la obra y desarrollar otro `'userland execve'`, igual de funcional, pero con un código más simple y orientado al uso de exploits. El resultado final ha sido la "shellcode ELF loader".

3. Shellcode ELF loader

La shellcode ELF loader o SELF es una nueva y sofisticada técnica de post-explotación basada en el `userland execve`. Permite el envío y ejecución de un binario ELF en una maquina remota sin almacenarlo en disco ni alterar el sistema de ficheros original.

El objetivo del shellcode elf loader es dotar a los exploits de un verdadero sistema anti-forense de post-explotación y que al mismo tiempo sea sencillo de utilizar. Esto es, dar la posibilidad a un atacante de ejecutar tantas aplicaciones como quiera.

4. Diseño e implementación

Obtener un diseño efectivo no ha resultado ser una tarea fácil, hemos considerado diferentes opciones y la mayoría de ellas han sido desechadas.

Al final, hemos optado por el diseño más ingenioso, el que permite mayor flexibilidad, portabilidad y facilidad de uso.

El resultado final consiste en un entramado de múltiples piezas independientes las unas de las otras, que realizan una tarea determinada y funcionan juntas en armonía. Estas piezas son tres: el `lXObject`, el `builder` y el `jumper`.

Estos elementos nos van a permitir ejecutar un binario en una maquina remota con suma facilidad. El `lxobject` es un objeto especial que contiene todo lo necesario para sustituir el ejecutable original de un proceso huésped por uno nuevo.

El `builder` y `jumper` son los códigos encargados de construir un `lxobject`, transportarlo desde una maquina local (atacante) a una maquina remota (atacada) y activarlo.

4.1 El `lxobject`

¿Que demonios es esto? un `lxobject` es un objeto autocargable y auto ejecutable, es decir, un objeto especialmente diseñado para sustituir por completo el proceso huésped original en el que se encuentre por un binario ELF que transporta e iniciar su ejecución. Cada `lxobject` se construye en la maquina del atacante utilizando "el `builder`" y posteriormente se envía a la maquina atacada donde lo recoge "el `jumper`" que se encarga de activarlo.

De esta forma, se puede considerar como un misil, el cual es transportado desde algún lugar hasta el punto de impacto, siendo la carga explosiva un ejecutable. Este "misil" esta compuesto por tres partes ensambladas: un binario elf estático, un contexto de pila preconstruido y una shellcode cargadora.

4.1.1. Binario ELF estático

Es la primera pieza del `lxobject`, es el binario ELF que queremos cargar y ejecutar en un host remoto. No es más que un fichero ejecutable normal y corriente, compilado estáticamente y para la arquitectura y sistema operativo en donde se va a ejecutar.

Hemos optado por no trabajar con ejecutables dinámicos porque esto añadiría una innecesaria complejidad al código de carga, incrementando considerablemente el porcentaje de posibles errores en el proceso.

4.1.2 Contexto de pila

Es la segunda pieza del `lxobject`, el contexto de pila que necesitara el binario. Todo proceso dispone de una zona de memoria llamada pila, en el cual las funciones almacenan sus variables locales. Durante la de carga de un binario, el kernel rellena esta región con una serie de datos iniciales, necesarios

para la posterior ejecución. A esto se le conoce como 'contexto inicial de la pila'.

Para facilitar la portabilidad y sobre todo el proceso de carga, hemos optado por utilizar un contexto de pila preconstruido. Esto es, se crea en nuestra maquina local y se ensambla junto con el binario ELF. Para crearlo solo es necesario conocer el formato y añadir los datos en el orden correcto. Igualmente, no es necesario añadir ningún vector auxiliar al trabajar con ejecutables estáticos

No hay restricciones con respecto al número de argumentos y variables de entorno, un gran número de argumentos originara un contexto de pila más grande, nada más.

Como el contexto es construido en la maquina del atacante, que generalmente será diferente de la atacada, será necesario conocer el espacio de direcciones que comprende la pila. Esto se hace automáticamente y no plantea ningún problema.

4.1.3. Shellcode cargadora

Es la tercera y mas importante de las piezas que constituyen el `lxobject`.

Consiste en una shellcode que tiene por misión realizar todo el proceso de carga y ejecución del binario transportado, una sencilla pero potente implementación de `userland execve()`.

El loader no realiza ningún procedimiento de construcción del contexto de pila, esto se lleva a cabo en el `builder`. Así, se dispone de un contexto preconstruido que simplemente se debe copiar a su espacio de direcciones correcto dentro del proceso.

A pesar de tener que codificar un loader distinto para cada arquitectura las operaciones son sencillas y concretas. Siempre que sea posible se pueden diseñar loaders híbridos, esto es, códigos capaces de funcionar en mismas arquitecturas pero distintos sistemas operativos unix, donde varían los métodos de llamadas de `syscalls`. El loader incluido en la implementación del apéndice es un híbrido capaz de funcionar en los sistemas linux y bsd de maquinas x86 de 32 bits.

4.2. El `builder`

Es el encargado de ensamblar las tres piezas que constituyen un `lxobject` y posteriormente enviarlo a una maquina remota. Funciona como una simple

tool de línea de comandos, su formato es el siguiente:

```
./builder <host> <port> <exec>
<argv> <envp>
```

donde:

host, port = dirección de la maquina atacada y puerto donde el jumper esta ejecutándose y esperando

exec = fichero binario ejecutable que queremos ejecutar

argv y envp = cadena de argumentos y cadena de variables de entorno que necesita el binario

Por ejemplo, si queremos realizar algunos escaneos de puertos desde la maquina atacada, ejecutaríamos un nmap de la siguiente forma:

```
./builder 172.26.0.1 2002 nmap-
static "-P0;p;23;172.26.1-30"
"PATH=/bin"
```

Básicamente las operaciones de ensamblaje son las siguientes:

- se crea una zona de memoria lo suficientemente grande para almacenar al fichero ejecutable estático, al shellcode loader y al contexto inicial de pila.

```
elf_new = (void*)malloc(elf_new_size);
```

- se introduce el ejecutable en la zona de memoria previamente inicializada y se limpian aquellos campos que describen la tabla de cabeceras de sección puesto que no será usada por nuestro ejecutable estático.

```
ehdr_new->e_shentsize = 0;
```

```
ehdr_new->e_shoff = 0;
```

```
ehdr_new->e_shnum = 0;
```

```
ehdr_new->e_shstrndx = 0;
```

- se construye el contexto inicial de pila. Son necesarias dos strings, la primera contiene los argumentos y la segunda contiene las variables de entorno.

Cada elemento dentro de estas cadenas es separado del resto mediante un delimitador. Un ejemplo:

```
<argv> = "arg1;arg2;arg3;-h"
```

```
<envp> = "PATH=/bin;SHELL=sh"
```

Una vez construido el contexto se añade una nueva cabecera en la tabla de cabeceras de programa del ejecutable. Esta cabecera es de tipo PT_STACK y contiene la información que necesitara el shellcode loader para copiarla en el lugar correcto.

- se introduce el shellcode ELF loader y se guarda su offset en el campo e_ident de la cabecera elf del ejecutable.

```
memcpy(elf_new + elf_new_size -
PG_SIZE + LOADER_CODESZ, loader,
LOADER_CODESZ);
```

```
ldr_ptr = (unsigned long
*)&ehdr_new->e_ident[9];
```

```
*ldr_ptr = elf_new_size - PG_SIZE
+ LOADER_CODESZ;
```

- el lobject esta listo, ahora se envía al host y puerto indicados.

```
connect(sfd, (struct sockaddr
*)&srv, sizeof(struct sockaddr)
```

```
write(sfd, elf_new, elf_new_size);
```

4.3. El jumper

El jumper es la shellcode que deberá ser usada por un exploit en el proceso de explotación de un servicio vulnerable. Su misión es activar el lobject y para ello debe realizar, al menos, las siguientes operaciones:

- abrir un socket y esperar la llegada del lobject.
- almacenarlo en alguna zona de memoria.
- activarlo saltando al loader.

Estas son las operaciones mínimas necesarias pero hay que tener en cuenta que es una simple shellcode y por lo tanto se podrá añadir (previamente) cualquier otro tipo de funcionalidad: romper un chroot, elevar privilegios, etc.

Durante el diseño del jumper se plantearon un par de problemas:

- 1) como obtener un `lxobject`?

Esto tiene fácil solución, se pueden aplicar algunas de las técnicas conocidas para shellcodes como asociarse a un puerto mediante `bind()` y esperar nuevas conexiones o buscar en la tabla de descriptores del proceso aquellos que se correspondan con un socket, etc.

Adicionalmente se podrían añadir sistemas de cifrado pero supondría hacer shellcodes gigantescas, muy complicadas de manejar.

- 2) y donde almacenarlo?

Hay tres posibles soluciones:

- a) almacenarlo en el heap del proceso huésped. Para hacerlo se debe localizar la `vaddr` de `break` usando `brk(0)` pero tiene el inconveniente de que podríamos sobrescribir o desmapear al propio `lxobject` en el proceso de carga.
- b) almacenarlo en la pila del proceso. Siempre que se conozca la `vaddr` correcta y se disponga de espacio suficiente este método es válido, aunque también puede existir la posibilidad de que el stack no sea ejecutable.
- c) almacenarlo en una nueva zona de memoria mapeada usando la syscall `mmap()`.

Este es el más recomendable y el que nosotros hemos utilizado en nuestro código.

Gracias a las características del jumper, su codificación puede ser personalizada y adaptada a diferentes contextos.

5. Multiejecución

El código incluido en este artículo es una implementación genérica y básica del shellcode ELF loader que permite la ejecución de un binario una sola vez.

Si queremos ejecutar ese binario un número indefinido de veces (para pasar diferentes argumentos, aprovechar otras funcionalidades o lo que sea) sería necesario construir y enviar un `lxobject` nuevo cada vez. Aunque esto tiene algunas

desventajas en la mayoría de las situaciones es válido y suficiente. Ahora bien, que ocurre si lo que deseamos hacer es poder ejecutarlo múltiples veces pero desde la propia máquina remota, sin construir ni enviar un `lxobject` para cada nueva carga y ejecución?

Para afrontar este problema hemos diseñado otra técnica denominada "multi-ejecución". La multiejecución es una implementación derivada más avanzada. Su principal característica es que la construcción de un `lxobject` se realiza en la máquina remota, así, un binario permite múltiples ejecuciones.

Sería algo similar a estar trabajando con una shell remota. Un ejemplo de herramienta que incorpora un entorno de multiejecución es el proyecto `gits` o "ghost in the system".

6. Gits

`Gits` es un entorno multiejecución que ha sido diseñado para operar en máquinas remotas atacadas y para limitar la cantidad de evidencias forenses. Debe ser considerado como una demostración, una extensión del shellcode ELF loader con más características. Está compuesto por una lanzadera y una shell. La shell es la parte más importante, permite obtener binarios y ejecutarlos tantas veces como se quiera gracias a su capacidad de reconstrucción del contexto de pila y parchado de los binarios automáticamente en la máquina remota.

7. Conclusión

Las técnicas de análisis forense son cada día más sofisticadas y completas, donde antes no se dejaba rastro ahora aparece una pista, donde antes dejábamos 1 pista ahora aparecen 100, una batalla continua entre el que no quiere ser detectado y el que quiere detectar. Utilizar la memoria y no tocar el disco son buenas medidas para no dejar evidencias, el shellcode ELF loader desempeña esta tarea de post-explotación con sencillez y elegancia.

8. Referencias

- [1] The Design and Implementation of `ul_exec` - the `grugq`
<http://securityfocus.com/archive/1/348638/2003-12-29/2004-01-04/0>
- [2] Remote Exec - the `grugq`
<http://www.phrack.org/show.php?p=62&a=8>

[3] Ghost In The System Project
<http://www.7a69ezine.org/gits>