Exploiting para niños.
Protecciones implementadas por el S.O.
La Historia-

v0.1 ~ 23-IX-2011
v0.2 ~ 30-III-2014
vlan7

Viejos gobiernan este mundo
¿dónde está la juventud?
Eskorbuto, Criaturas al poder

==Protección NX

Si los Exploiters inyectan shellcodes en la pila para ejecutarlos, les impedimos ejecutar código en la pila, pues en condiciones normales un programa no necesita ejecutar código en esa zona de memoria.

Este es el objetivo de NX, que en Linux está disponible para versiones de kernel mayores que 2.6.8 (parches previos aparte [1])

Si una zona de memoria tiene activo el flag NX, esa zona se marcará como no ejecutable. En la práctica intentar ejecutar un shellcode en una zona NX nos llevará a que el programa aborte con una violación de segmento.

Si quisiéramos provocar digamos un stack overflow en un programa en ejecución, estaremos bajo el ámbito del programa a explotar, por lo que como regla general nuestro shellcode tendrá los permisos que tenga el programa explotado. Podemos comprobar los permisos de un binario *vuln* en la pila así:

```
readelf -l vuln |grep stack
```

==Bypass NX (ret2libc)

Este hack fue compartido por Solar Designer en Agosto-1997 en la lista de correo bugtraq [2], seguido de una respuesta no menos interesante [3]

Si una zona de memoria está marcada con el flag NX (o escribes o ejecutas, pero nunca ambos permisos a la vez), aunque tengamos permisos de escritura que nos permitan colocar un shellcode en ella, no podremos ejecutarlo.

De acuerdo, pero... ¿y si atacamos alguna de las librerías que el programa a explotar tenga mapeadas en memoria?

En Linux cualquier código en alto nivel va a usar (g)libc, y esta librería tiene llamadas como system() que sí estarán cargadas en un espacio de memoria ejecutable.

Bien, en lugar de construir un payload para acabar ejecutando el shellcode de nuestra elección... ¿y si construimos un payload para acabar ejecutando lo que le digamos a

system(), execve() o a cualquier otra función crítica similar que encontremos en la librería?

Manos a la obra pues. El proceso será localizar una llamada a system() dentro de libc además de la cadena con el argumento deseado, y una vez tengamos esas direcciones de memoria, construir en la pila un payload similar a:

```
[*system][@RET][*argumentos]
```

Bien, en la práctica con un payload así lograríamos que system() ejecutara los argumentos que le pasemos, supongamos que una shell, entonces resulta que cuando salgamos de la shell, cuando vaya a finalizar la función system() acabaremos provocando un Segmentation fault, debido básicamente a ciertos descuadres que nuestro payload ha creado en la pila.

A efectos prácticos ya hemos conseguido ejecutar lo que queríamos así que a quién le importa, pero podríamos tener un flujo de ejecución limpio y evitar el segfault si buscamos la dirección de memoria de una llamada a exit() en libc y construimos un payload de tal forma que retornemos a exit() cuando system() finalice. Así:

```
[*system][*exit][@RET][*argumentos]
```

Donde los argumentos esta vez son para system() y exit()

==ESP lifting

Este hack fue compartido por Nergal en la Phrack [4]

Una limitación de la técnica ret2libc tradicional es que no podemos encadenar más de dos funciones seguidas en la pila.

Lo que planteó Nergal fue intercalar instrucciones “pop ; ret” en el payload ret2libc. Teniendo en cuenta que la instrucción pop desplaza +4 bytes el registro ESP, si tenemos cierta habilidad construyendo el payload, estaremos provocando que, a medida que el flujo de ejecución avanza, la cima de la pila vaya apuntando a una función u otra, con lo que conseguiremos que se vayan ejecutando una a una las funciones que queramos.

El único límite en principio es el tamaño del buffer, aunque en buffers pequeños podríamos saltar a otra parte de la memoria y continuar nuestro payload allí, pero eso es otra historia...

==Borrowed code chunks

Este hack fue compartido por Krahmer dentro de la página oficial de Suse [5]

La convención de llamada a las funciones más común en Linux/x86 es la convención cdecl. Para llamar a una función en cdecl, básicamente primero se colocan uno a uno los argumentos en la pila e inmediatamente después llamamos a la función usando la instrucción call, que provoca que la CPU guarde el contenido del registro EIP en la pila con el fin de que cuando la función termine, el flujo de ejecución pueda retornar justo a la instrucción siguiente a call.

La instrucción call se ejecuta. Entonces nos encontramos justo al inicio de la función que

acabamos de llamar y para que las cosas funcionen [6], comienza lo que se conoce como *prólogo*.

Justo al inicio del prólogo, el registro apuntador base ebp es el marco de pila donde nos encontrábamos justo antes de entrar a la función. Lo primero que haremos será un *backup* de ebp, que nos permitirá que cuando la función finalice podamos volver al marco de pila donde nos encontrábamos justo antes del call. Después haremos que ebp pase a apuntar a la cima de la pila y decrementaremos ESP para reservar espacio a las variables locales.

```
push ebp          ;save the previous frame pointer
mov ebp, esp      ;esp (top of the stack) becomes new ebp
sub esp, 0x20     ;reserve 32 bytes for local variables
```

Un momento, si ESP apunta a la cima de la pila y necesitamos que la pila crezca para poder alojar las variables locales, ¿por qué lo decrementamos? Para hacer la pila más grande... ¿no deberíamos incrementarlo? Bien, puede parecer confuso, pero resulta que aunque ESP apunta a la cima de la pila, ESP siempre es la dirección de memoria más baja de toda la pila. Esto es debido a que en arquitecturas como x86 la pila crece hacia direcciones de memoria más bajas, por lo tanto la cima de la pila siempre será la dirección de memoria más baja.

Bueno, después del prólogo ya podemos leer (o sobrescribir) los argumentos que habíamos apilado justo antes del call utilizando el registro apuntador base ebp. El primer argumento estará en [ebp+8], el segundo en [ebp+12], el tercero en... y mientras dure la función, en [ebp] estará el viejo ebp del que hicimos backup y en [ebp+4] tendremos el EIP que guardó la instrucción call en la pila.

Para que las cosas puedan seguir funcionando, cuando la función finalice se desencadenará un proceso que conocemos como *epílogo*, que se encargará de dar marcha atrás a todos los cambios que provocó el prólogo.

El caso es que en x64 todo esto cambia radicalmente :P porque la convención de llamada a las funciones en x64 establece que los argumentos deberán pasarse a través de registros, por orden: rdi, rsi, rdx, rcx, r8 y r9.

Bien, si no podemos servirnos de la pila para pasar parámetros a las funciones, parece que los Exploiters tienen un obstáculo para utilizar las técnicas ret2libc tradicionales en x64. Lo que planteó Krahmer para resolver este problema fue lo siguiente...

Supongamos que el código del programa a explotar contiene las instrucciones “pop rdi ; ret”. Si llegamos a controlar la pila (bang! overflow-) y guardamos en ella la dirección de memoria donde comienza la instrucción “pop rdi”, entonces acabamos de recuperar el control sobre el paso de parámetros.

Si además encontramos otros fragmentos (*chunks*) de código que nos den el control sobre los registros que necesitemos para ejecutar la función deseada, y los encadenamos como si de un puzzle se tratara, habremos sorteado la protección NX en x64.

Krahmer bautizó esta técnica con el nombre de *Borrowed code chunks*. Actualmente el

nombre más extendido para esta técnica es *Return oriented programming* o ROP, en la que profundizaremos un poco más enseguida.

==Protección ASLR

Si los Exploiters inyectan shellcodes en la pila para ejecutarlos, y para ello necesitan conocer la dirección de memoria donde comienza su shellcode, les complicamos las cosas haciendo que la pila resida en direcciones de memoria distintas a cada ejecución de los programas.

Este es el objetivo de ASLR, que en Linux está disponible para versiones de kernel mayores que 2.6.12 [7]

==Bypass ASLR+NX (ROP)

Este hack fue compartido por Shacham [8]

Las instrucciones ROP se llaman gadgets, y son usadas en nuestro beneficio para construir un payload en la pila y ejecutarlo tras provocar el overflow que nos de el control sobre el flujo de ejecución.

La idea para encontrar un gadget es la siguiente. La instrucción RET se codifica en x86 siempre con el opcode C3, por lo tanto si buscamos en un volcado hexadecimal el byte C3, los bytes anteriores podrían alterar el flujo de ejecución en nuestro beneficio, y si lo hacen acabamos de encontrar un rop-gadget, por ejemplo “pop x ; pop y ; ret”

Cuando no existían protecciones y la pila era ejecutable, traducíamos el código ASM de nuestro shellcode a sus opcodes correspondientes, y el shellcode era el *core* del payload.

Un payload ROP en cambio se compone de direcciones de memoria, donde cada una de estas direcciones será un puntero hacia la primera instrucción de cada gadget.

La última instrucción de cada gadget será un ret, con lo cual cada vez que un gadget finalice retornaremos a la pila y allí nos espera el siguiente puntero al siguiente gadget, y así sucesivamente, por lo que si encontramos gadgets suficientes y somos hábiles encadenándolos, lograremos que nuestro payload acabe ejecutando la función que queramos.

Únicamente nos servimos de la pila para colocar en ella nuestro payload, por lo tanto y dado que no estamos ejecutando nada en la pila no nos afecta la protección NX.

Por otra parte, eliminar funciones críticas de las librerías no es una defensa válida contra ROP, ya que ahora en lugar de saltar a la dirección de memoria donde comienza la función en la librería, como se haría en las técnicas ret2libc tradicionales, la idea a tener en cuenta a la hora de construir nuestro payload ROP es ir construyendo la función a base de gadgets.

Para lograr que una función acabe ejecutándose debemos encontrar gadgets que nos den el control sobre los registros necesarios para ejecutar la función deseada. Pongamos que queremos servirnos de la syscall execve() para ejecutar una shell, entonces sería algo como:

```
execve("/bin/sh\0", *"/bin/sh\0", (char **)NULL);
```

Necesitamos controlar ebx, ecx y edx para establecer el primero, segundo y tercer argumento de la syscall. Podríamos aprovechar que la sección .data tiene permisos de escritura y no le afecta ASLR para almacenar en ella la cadena con la shell. Necesitamos control sobre eax para guardar el número de syscall asignado a execve. Por último necesitamos un gadget que contenga la mítica instrucción “int 0x80” para invocar al kernel.

Existen herramientas que automatizan el proceso de búsqueda de gadgets como ropeme [9] o, de forma más reciente, ROPgadget [10]

Obviamente, para que no nos afecte ASLR, debemos construir los gadgets a partir del desensamblado del propio programa a explotar, ya que las direcciones de memoria no cambian, como si ASLR no existiera. Podríamos también reutilizar código de librerías que el programa tenga mapeadas en memoria y que no estén compiladas con ASLR. O bien aprovecharnos de alguna característica que nos permita obtener información en tiempo de ejecución de direcciones de memoria, lo que en jerga se conoce como un *infoleak*.

En x86 ROP se sostiene debido a que el juego de instrucciones de x86 es de tamaño variable, por lo tanto, debido a un curioso efecto en arquitecturas de tamaño variable que Izik denominaba *doble sentido* [11] es posible encontrar en un ejecutable a explotar múltiples combinaciones de instrucciones que alteren el flujo de ejecución en nuestro beneficio.

Roemers describe ROP para SPARC, una arquitectura de tamaño fijo. [12]

==ROP sin RET

Este hack fue compartido por Shacham y Stephen Checkoway [13]

La última instrucción de cada gadget debe provocar que el flujo de ejecución vuelva a la pila a por el siguiente puntero al siguiente gadget. Lo normal es que esta instrucción sea ret.

En el paper básicamente se analiza el comportamiento de las distintas instrucciones existentes en x86 para ver si es posible sustituir ret por instrucciones equivalentes, por ejemplo "pop registro ; jmp [registro]".

Además de aumentar las posibilidades de encontrar gadgets útiles en el código para beneficio del Exploiter, otro efecto a su favor podría ser el camuflaje.

Ante un IDS que asuma que los gadgets de un exploit ROP deben acabar siempre con un byte c3 (instrucción RET), parece lógico pensar que si retornamos a la pila utilizando instrucciones equivalentes, las probabilidades de que el payload pase desapercibido aumentan en favor del Exploiter.

==========

That's all for now. I hope I managed to prove that exploiting buffer overflows should be an art.
Solar Designer

==Referencias

[1] Linux kernel patch from the Openwall Project (historical), http://www.openwall.com/linux/
[2] Getting around non-executable stack (and fix), http://seclists.org/bugtraq/1997/Aug/63
[3] Defeating Solar Designer's Non-executable Stack Patch, http://insecure.org/sploits/non-executable.stack.problems.html
[4] The advanced return-into-lib(c) exploits. PaX case study, http://www.phrack.com/issues.html?issue=58&id=4&mode=txt
[5] x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique, http://www.suse.de/~krahmer/no-nx.pdf
[6] Cómo funcionan las cosas. David Macaulay, Neil Ardley. Círculo de Lectores. Barcelona, 1989
[7] Linux kernel ASLR Implementation, https://xorl.wordpress.com/2011/01/16/linux-kernel-aslr-implementation/
[8] The Geometry of Innocent Flesh on the Bone: Return-into-libc without function Calls (on the x86), http://cseweb.ucsd.edu/~hovav/dist/geometry.pdf
[9] ROPEME tool, http://www.vnsecurity.net/2010/08/ropeme-rop-exploit-made-easy/
[10] ROPgadget tool, http://www.shell-storm.org/project/ROPgadget/
[11] Advanced Buffer Overflow Methods [or] Smack the Stack. Cracking the VA-Patch, https://events.ccc.de/congress/2005/fahrplan/events/491.en.html
[12] Finding the Bad in Good Code. Automated Return-Oriented Programming Exploit Discovery, http://cse.ucsd.edu/~rroemer/doc/thesis.pdf
[13] Escape From Return-Oriented Programming. Return-oriented Programming without Returns (on the x86), http://cseweb.ucsd.edu/~hovav/dist/noret.pdf