Criptografía y Seguridad

Explotación de Vulnerabilidades: Desbordamiento de Búfer

Facultad de Ciencias
Universidad Nacional Autónoma de México

5 de marzo de 2024

Introdución

Los expertos en seguridad, comúnmente llamados hackers éticos, no solo se dedican a escribir código seguro, sino que una vez programado, explotan su propio código con el fin de asegurarse que este carezca de alguna vulnerabilidad. Convertirse tanto en el ladrón como en el policía son papeles que el hacker ético debe dominar y a pesar de que ambos papeles pueden parecer opuestos, utilizan técnicas similares de resolución de problemas [3].

La explotación de una vulnerabilidad, es el elemento básico del hacking. Todo programa consiste en un conjunto de reglas que, dada una entrada, nos entrega una salida. Estas reglas siguen un cierto flujo de ejecución que le dice a la computadora que hacer. Hallar la vulnerabilidad de un programa y explotarla consiste en hacer que el programa haga lo que deseamos, incluso si este no fue diseñado para hacerlo en un principio.

Las vulnerabilidades en un programa pueden tener graves consecuencias tanto para las personas como para las organizaciones. Pueden provocar violaciones de confidencialidad, mal funcionamiento del sistema o incluso ataques maliciosos. Por lo tanto, es esencial comprender las vulnerabilidades potenciales de un programa y cómo se pueden prevenir.

El búfer, por ejemplo, se ha convertido en una vulnerabilidad por si mismo, y es un claro ejemplo de como el programador puede no comprender cómo diseñar, crear e implementar código seguro y la importancia de este [2].

El desbordamiento de búfer puede aparecer en cualquier sitio, desde calendarios, calculadoras, videojuegos, servidores y código escrito por cualquier experto, y puede ser tan simple como un carácter fuera de lugar o ser tan complejo como miles de matrices mal manejadas ([4], pág.3).

Un desbordamiento de búfer puede ocurrir en cualquier momento si no se diseña con cuidado el código, y aun si el programador es meticuloso, puede aparecer por error humano. Lo único que podemos hacer para prevenirlo, es conocerlo, controlarlo y aprender a cómo es que alguien podría explotarlo.

Objetivos

Generales:

En esta práctica, el alumno se aprovechará de la vulnerabilidad de desbordamiento de búfer por medio la realización de un programa catalogado exploit. Se espera que al final de esta práctica,

el alumno sea capaz de analizar y saber como se explota el código inseguro, y a su vez corregir el programa para que deje de ser peligroso.

Particulares:

Aprender el término exploit, vulnerabilidad y desbordamiento de búfer, así como conocer algunas características del lenguaje C que pueden generar código inseguro.

Requisitos

- **Conocimientos previos:** Fundamentos en arquitectura y organización de computadoras. Lenguaje ensamblador. Manejo de bash, python y C.
- **Tiempo de realización sugerido:**
  8 horas.
- **Número de colaboradores:**
  2
- **Software a utilizar:**
  - gcc
  - gdb
  - bash
  - python
  - C

Planteamiento

La vulnerabilidad de desbordamiento de búfer o buffer overflow, es un problema de seguridad que involucra la memoria en donde el software o programa no considera o verifica sus límites de almacenamiento [7]. Esta vulnerabilidad puede ser explotada por los piratas informáticos para obtener acceso a un sistema informático, permitiéndoles insertar código malicioso en el sistema y provocar el robo o la destrucción de datos.

Las vulnerabilidades de desbordamiento de búfer han existido desde los primeros días de las computadoras y aún provocan dolores de cabeza a los desarrolladores de software. La mayoría de los gusanos de Internet utilizan la vulnerabilidad de desbordamiento de búfer para propagarse y las consecuencias de un ataque exitoso pueden ser graves, por lo que es esencial que se tomen medidas contra este tipo de ataques.

C [5] es un lenguaje de programación en el cual se puede generar fácilmente programas que son vulnerables al desbordamiento de búfer y esto se debe a su simplicidad y a que el lenguaje asume que el programador es responsable de la integridad de los datos. Si no hiciera esto y trasladará la responsabilidad al compilador, los archivos binarios resultarían más lentos debido a las comprobaciones de integridad de cada variable, y esto haría más complejo el lenguaje.

Dada la relación costo-beneficio de robustecer la seguridad de C, se dejó mejor la responsabilidad de generar un código seguro al diseñador del programa. Por lo tanto, para diseñar un código seguro,

primero debemos entender como es que funciona la vulnerabilidad, como explotarla y finalmente arreglarla [9].

La vulnerabilidad de desbordamiento de búfer se genera cuando la memoria del programa recibe una cantidad de datos mayor a la que realmente puede procesar de acuerdo a como fue desarrollado. Sucede principalmente al usar funciones de C de bajo nivel inseguras como ***gets***, ***strcat*** y ***sprintf*** [12][6]. Estas funciones, principalmente se utilizan para escribir una cadena o variable en alguna parte de la memoria en una longitud determinada. Si intentamos escribir algo que sea más grande que lo declarado, puede sobreescribir las direcciones de memoria posteriores y con eso inyectar código malicioso.

Esto sucede principalmente por como está diseñada la memoria y como esta se comporta cuando se ejecuta un programa[1]. Cuando el sistema operativo ejecuta un programa, este llamará como función al método principal del código, pero el proceso real, el ejecutable, se mantendrá en la memoria de una manera específica.

En la figura 1, se puede apreciar un bloque de RAM, en el cual, la dirección de memoria $0xFFFF\ldots$ equivale a la memoria $1111\ldots$ basta 32 o 64 bits, dependiendo de la arquitectura, y $0x000\ldots$ es la parte inferior de la memoria.

La memoria cuenta con ciertas áreas de memoria [10] que siempre están asignadas a algunos procesos, por ejemplo, en la parte superior de la memoria, podemos encontrar los elementos vinculados con el Kernel, líneas de comando que podemos pasar a un programa o variables de entorno.

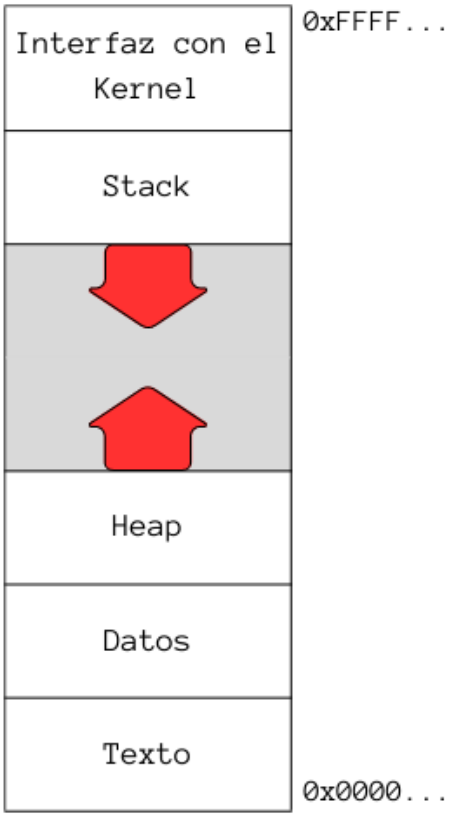

Figura 1. Bloque de RAM

En la parte inferior podemos encontrar una área de texto donde se ubicará el código real del programa a ejecutar. Las instrucciones de máquina que se han compilado se cargan en esta área y es exclusiva de lectura.

En el área superior al texto se encuentran los datos, las variables inicializadas y no inicializadas y seguido de esto, encontramos el heap. El heap es alojan se asigna los elementos dinámicos en la memoria. Se puede asignar grandes fragmentos para computar o almacenar.

Finalmente y antes del kernel, nos encontramos con el stack que contiene las variables locales para cada una de las funciones del programa a ejecutar. Por ejemplo, si llamamos a una nueva función, *scanf* [13] con algunos parámetros, estos se colocarán al final de stack.

El heap crece hacia arriba a medida que se agrega memoria y la pila crece en la dirección contraria, generando el preámbulo al desastre.

Si vemos un segmento más detallado como el de la figura 2, podemos percibir el problema. Es en el stack en donde se crean los registros de activación en las llamadas a subrutinas, es decir, se almacenan los parámetros con los que estas son llamadas y la dirección de retorno una vez que terminan.

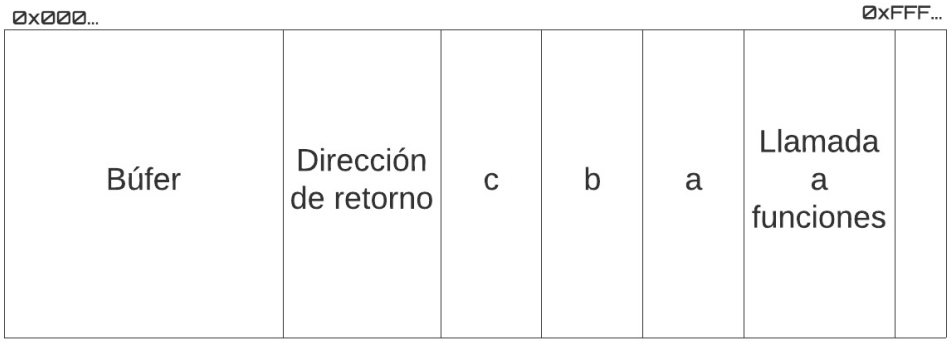

Figura 2. Sección de Stack

Si tenemos un programa que está llamando a una función qie recibe 3 parámetros digamos a, b y c, el aspecto del stack es como el mostrado en la figura 2.

Cuándo se escribe algo más grande que el buffer y este sobreescribe la dirección de retorno del stack, tenemos un problema de desbordamiento de buffer, creando así esta vulnerabilidad la cual el ciberdelincuente puede aprovechar para ingresar código malicioso, como lo que se busca en esta práctica.

En conclusión, esta práctica está enfocada principalmente en explotar la vulnerabilidad de desbordamiento de buffer. Se darán y usarán herramientas como gdb que ayuden a realizar esta labor, para la creación de un programa catalogado exploit.

Desarrollo

Una vez comprobado que se dispone del software necesario para la realización de la práctica, se puede iniciar con la identificación de la prueba de penetración que se va a realizar. Como se cuenta con el código fuente al cual se pretende realizar una explotación, se puede concluir que el mejor camino es un pentest de caja blanca.

Los dos programas anexados realizados en C cuentan con un diseño inseguro, utilizando funciones inseguras del lenguaje C, por lo que son vulnerables al desbordamiento de búfer y sobre estos se realizará la práctica.

Los programas que se van a explotar se encuentran cada uno dentro de una sub carpeta, y dentro de la carpeta **Practica03**. Por lo que el primer paso es identificarlos y analizar las herramientas que se cuentan para la realización de una prueba de penetración de caja blanca.

En la fase de recopilación de la prueba de penetración se documentan las herramientas necesarias para llevar a cabo el ataque, así como las versiones utilizadas, por lo que se deberán anexar en el reporte.

Posteriormente, analiza el diseño del código, identifica la vulnerabilidad, su causa y planea como aprovechar estas condiciones en los siguientes programas:

- simple-password-verification.c

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

char *login= "root";


int granted()
{
  printf("Lograste llegar hasta aquí. ¡Felicidades!\n");
  printf("Acceso Autorizado...");
  // Esta linea de código puede no funcionar en algunos SO
  // Realiza la practica en Kali Linux
  system("gnome-terminal -x sh -c \"./a.out\"");
  return 0;
}

int main()
{
  char password[16];
  printf("¡Bienvenido!\n");
  printf("Anota la contraseña porfavor:");
  gets(password);

  if(strcmp(password, login))
  {
    printf("Lo siento la contraseña es incorrecta. \nAcceso Denegado");
  }
  else
  {
    granted();
  }
}
```

- simple-verification.c

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

int secretMsg()
{
```

```
6   printf("Lo lograste, el secreto es tomar mucha awuita\n");
7   printf("Lograste llegar hasta aquí. Felicidades");
8   return 0;
9 }
10
11 int main()
12 {
13     char pass[16];
14     printf("Anota una palabra e intenta descubrir el secreto: \n");
15     gets(pass);
16
17     if (pass[0] == '\0')
18     {
19         printf("No lo lograste. Intenta de nuevo");
20         return 0;
21     } else if(strlen(pass) < 11)
22     {
23         secretMsg();
24     } else
25     {
26         printf("No lo lograste. Intenta de nuevo");
27         return 0;
28     }
29 }
```

Cuando se identifique las causas de las vulnerabilidades, se proseguirá con la explotación de las mismas, por lo que en esta etapa de explotación, la meta es sobreescribir el puntero base e ignorar los condicionales para ejecutar las funciones aun sin tener la contraseña o cumplir los requerimientos.

Se tendrá que averiguar en que parte el búfer sobreescribe el puntero base, basado en el código fuente. Identifica cuantos caracteres es necesario ingresar en la terminal para que suceda el desbordamiento. Las cadenas o strings son el mejor amigo de los hackers. Crea cadenas inteligentes que te ayuden al momento de buscar la longitud exacta que soporta el programa.

Una práctica común entre los expertos de seguridad es utilizar una cadena con “A” repetidas, las n-veces que sea necesario hasta desbordar el búfer. La letra “A” es una gran aliada, ya que es fácil de identificar en las direcciones de memoria, puesto que en hexadecimal se representa como “0x41” y si al momento de analizar la memoria se encuentra series de “0x41414141” es fácil identificar que es el búfer sobreescribiendo el stack donde se guarda el código.

Utiliza *gdb* (Gnu Project Debugger)[8] la cual es herramienta que permite, entre otras cosas, correr el programa con la posibilidad de detenerlo cuando se cumple cierta condición, avanzar paso a paso, analizar que ha pasado cuando un programa se detiene o cambiar algunos parámetros del programa como el valor de las variables.

Con *gdb* se puede correr el programa con un exploit, y recibir mensajes más explícitos de falla de segmentación y que fue lo que intento hacer la computadora cuando eso sucedió, en otras palabras, analiza la pila segmento a segmento. También crea puntos de quiebre o breaks, para analizar la pila de llamadas antes y después de ingresar datos, lo cual es útil para encontrar el puntero de retorno.

Cuando se encuentra el puntero de retorno [11] lo siguiente es sobreescribirlo. Deberás realizar un exploit que alimente tu código y corrompa la pila al sobreescribirla con la dirección de memoria a la función que se quiere llegar, otorgando la posibilidad de saltar las verificaciones de los códigos y garantizar el acceso a las funciones internas.

Entrada

Se crearán dos archivos de Python llamados miniexploit-pass.py y miniexploit-ver.py, que vulnere simple-password-verification.c y simple-verification.c respectivamente. De manera que al momento de correr el ejecutable y alimentarlo con su exploit correspondiente, sea capaz corromper la pila al sobreescribirla.

Salida

Mensajes secretos de funciones internas del programa sin cumplir con las condiciones programadas para adquirilos. También se deberán anexar en sus respectivas carpetas las versiones reparadas de cada codigo en C, de manera que ya no sean vulnerables a desbordamiento de búfer.

Se deja al estudiante la libertad de utilizar cualquier bandera que sea de utilidad del compilador gcc. En caso de utilizar una o más banderas, documentar y justificar por qué decidieron aplicarlas en el resporte de la práctica,

Procedimento

Se entregará un pdf debidamente documentado con el análisis de los archivos vulnerables siguiendo las fases del prueba de penetración, así como cuatro códigos diferentes. Dos de estos códigos serán exploits, o programas de explotación que cada uno corrompa una pila de los programas antes mencionados y logre el cometido de acceder a funciones internas sin validación previa.

Los otros dos programas serán versiones corregidas de los programas inseguros y mal diseñados, entregados para la realización de esta práctica. Deberás analizar a profundidad el código y justificar tus cambios en el reporte final.

Cabe resaltar que el usar funciones seguras no previene del todo una vulnerabilidad de desbordamiento de búfer. Investiga el cómo generar código seguro, así como las buenas prácticas de programación que se deben seguir como métodos de prevención.

Por otro lado, el reporte deberá incluir las siguientes secciones:

- **Recopilación**: Donde se debe incluir todo lo que se necesita para obtener la información mediante la prueba de penetración. Por ejemplo, ¿qué problema se quiere resolver o qué se quiere saber?, ¿qué información es necesaria para ello?, ¿qué herramientas contamos?, ¿para qué?
- **Análisis**: Identificarás la vulnerabilidad y como aprovecharla. Documentarás ampliamente todo lo que se logre encontrar en código.
- **Explotación**: En esta etapa se documentará la creación del exploit. Como se llegó al puntero base, y como se usaron las herramientas para lograrlo. Agrega evidencias como capturas de pantalla de guía.
- **Post-explotación**: Explicarás tu exploit resultante. Darás conclusiones a las cuales llegaste respecto al código inseguro. Explicarás como corregir el código de la práctica y anexarás la corrección en su respectiva carpeta.
- **Imagina**: En esta fase imagina todo lo que se puede realizar utilizando esta vulnerabilidad. Explícalo detalladamente y reflexiona el alcance de lo que acabas de realizar.

Se debe incluir la bibliografía correspondiente. El carecer de referencias invalidará la práctica entera. Deberá contener los datos obtenidos completos y debidamente explicados.

Preguntas

1. ¿Qué significa que una prueba de penetración sea de caja blanca?
2. ¿Por qué es nesesario aprender a explotar código?
3. Menciona el significado de los siguientes registros.
   - EAX
   - EBX
   - ECX
   - EDX
   - ESI
   - EDI
   - EBP
   - ESP
   - EIP
4. ¿Qué es little endian y big endian? ¿Cuál usa tu procesador? ¿Qué arquitectura tiene tu computadora?
5. ¿Qué es un segmento y que es un offset?
6. ¿Qué realiza la instrución lea en ensamblador?
7. ¿Crees que esta vulnerabilidad desapareció con las nuevas funciones seguras como *fgets*?
8. Investiga tres casos famosos en lo cuales se aprovecho la vulnerabilidad de desbordamiento de búffer. ¿Cuáles fueron sus repercuciones?
9. Investiga el bus Off-by-One. ¿Cómo funciona? ¿Cómo se previene?
10. ¿Se usa codigo shell solo en exploits? Argumenta tu respuesta.

Bibliografía

[1] Martha Irene Romero Castro y col. *Introducción a la seguridad informática y el análisis de vulnerabilidades*. Editorial Área de Innovación y Desarrollo,S.L., 2018. DOI: 10.17993/IngyTec.2018.46.

[2] Crispin Cowan y col. *Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade*. Department of Computer Science y Engineering Oregon Graduate Institute of Science Technology, 2002. DOI: 0.1109/DISCEX.2000.821514.

[3] Jon Erickson. *Hacking: The art of exploitation*. 2da edición. William Pollock, 2008.

[4] James C. Foster y col. *Buffer Overflow Attacks: Detect, Exploit, Prevent*. Andrew Williams, 2005.

[5] Gnu. *The GNU C Reference Manual*. Gnu.org. Mayo de 2023 [Online]. URL: https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html.

[6] Zhimin Gu, Jiandong y Yao Jun Qin. *Buffer Overflow Attacks on Linux Principles Analyzing and Protection*. Department of Computer Science, Beijing Institute of Technology.

[7] Samanvay Gupta. «Buffer Overflow Attack». En: *IOSR Journal of Computer Engineering (IOSRJCE)* 1 (2012), págs. 10-23.

[8] Pedro Alves (Red Hat) y col. *GDB: The GNU Project Debugger*. Sourceware.org. Feb. de 2023 [Online]. URL: https://www.sourceware.org/gdb.

[9] Michael Howard y David LeBlanc. *Writing Secure Code*. 2da edición. Microsoft Press, 2003.

[10] Adrian Losada. *Explotación de software en arquitecturas x86 (I): Introducción y explotación de buffer overflows*. Softtek.com. Feb. de 2023 [Online]. URL: https://blog.softtek.com/es/explotaci%C3%B3n-de-software-en-arquitecturas-x86-i-introducci%C3%B3n-y-explotaci%C3%B3n-de-un-buffer-overflow.

[11] OWASP y col. *Buffer Overflow Attack*. OWASP.org. Feb. de 2023 [Online]. URL: https://owasp.org/www-community/attacks/Buffer_overflow_attack.

[12] Raphael Tawil. *Eliminating Insecure Uses of C Library Functions*. Systems Group, Department of Computer Science, ETH Zurich, 2012. DOI: 10.3929/ethz-a-007215322.

[13] Tyler Whitney, Colin Robertson y Andrew Rogers. *scanf,* $_s scanf_l, wscanf_{,w} scanf_l$. Microsoft.com. Mayo de 2023 [Online]. URL: https://learn.microsoft.com/es-es/cpp/c-runtime-library/reference/scanf-scanf-l-wscanf-wscanf-l?view=msvc-170.