Arquitectura del Computador

Trabajo Final

Desbordamiento de buffers

Integrantes	Legajo
Alonso Pablo	A-4121/1
Avelin Federico	A-4077/1
Rivosecchi Alejandro	R-4097/5

Motivación para la elección del trabajo

Nuestra motivación fue adquirir conocimientos sobre las vulnerabilidades que permiten corromper los programas hechos en C en relación al desbordamiento de buffers, para así escribir códigos más fiables y seguros.

También sabíamos que el trabajo al ser implementado sobre un servidor web tiene la ventaja que nos iba permitir aprender acerca de los servidores web y el protocolo HTTP: como es la comunicación con los clientes, que tipos de solicitudes existen, como se realizan y responden.

Además nos resultó interesante para profundizar sobre cómo los programas escritos en C manejan su memoria, como cuanto espacio reserva para los distintos tipos de variables, como las ordena, qué métodos usa para alinear.

Desbordamiento de buffer

Esto hace referencia al tipo de error que se comete cuando se está escribiendo sobre un espacio de memoria reservado para eso, el buffer, pero por no realizarse una verificación del tamaño asignado a este, o realizar una mala verificación, se escriben datos sobre porciones de memorias que no le corresponden ya al buffer.

Esta clase de errores pueden tener consecuencias muy graves a lo que respecta la seguridad del programa, ya que, con los adecuados conocimientos y una adecuada implementación se puede cambiar el rumbo de ejecución del programa a decisión de un tercero que no esta autorizado a hacerlo.

Es un problema que se puede dar en programas escritos en C ya que este posee funciones que realizan escritura de datos sin verificar el tamaño del buffer.

Desarrollo del trabajo

Para abordar el trabajo comenzamos leyendo las instrucciones dadas por la cátedra y luego la información disponible sobre el trabajo en la página del MIT.

El primer problema con el que nos encontramos fue como hacíamos correr la imagen de Ubuntu 16.04.1 LTS que contiene el servidor web (zoobar web application), ya que lo intentabamos hacer con máquinas que emulaban i686 pero debíamos hacerlo sobre i386.
Finalmente corrimos la imagen usando el emulador qemu o kvm, ambos funcionaron con las correspondientes flags.

Luego de esto comenzamos a estudiar los archivos de código fuente del servidor. Este artículo sobre el protocolo HTTP nos ayudo a comprender el funcionamiento del servidor. El siguiente paso fue buscar vulnerabilidades en las funciones que podrían ser aprovechadas mediante el desbordamiento de buffer y esto fue lo que encontramos.

Nota: Incluimos pseudo-diagramas de pila para la mayoría de las funciones en los cuales figuran los valores más relevantes pero no están completos (entre otras cosas, el compilador reserva espacios de memoria para cuestiones de alineación).

Vulnerabilidades:

process_client() (zookd.c)**:** Es posible desbordar char reqpath[2048]. Esta función llama a http_request_line() (http.c) y esta llama a url_decode() (http.c) que copia una cadena en reqpath. Al no controlar la cantidad de caracteres que se copian, process_client() permite desbordar reqpath.

offset (desde reqpath)	Corresponde a
2067	return address
2063	ebp
0	reqpath[2048]

http_request_headers() (http.c)**:** Es posible desbordar char value[512] y char envvar[512]. Esta función separa los headers en 2 partes: valor y nombre. Al utilizar url_decode() (http.c) para copiar el valor del header en value no se controla la cantidad de caracteres copiados.

El arreglo char envvar también es vulnerable pues se puede copiar una cantidad de caracteres mayor a su tamaño a través de la función sprintf() siempre y cuando se cumpla la condición impuesta en el if ser distinto a la cadena CONTENT_TYPE o CONTENT_LENGHT.
De esta forma ambos se pueden desbordar.

offset (desde value)	Corresponde a
536	return address
532	ebp
520	int i
0	value[512]
-512	envvar[512]

http_serve() (http.c)**:** Es posible desbordar char pn[1024]. Esta función utiliza strcat() y no chequea el tamaño de pn. Por lo tanto es posible desbordar pn, concatenando con una cadena lo suficientemente larga.

offset (desde pn)	Corresponde a
1040	return address
1036	ebp
1024	*handler
0	pn[1024]

http_serve_directory() (http.c): Es posible desbordar name[1024]. Esta función recibe como segundo argumento char* pn y llama en algún momento del cógido a dir_join. Una de las instrucciones de dir_join es strcpy(dst,dirname) donde dst = name y dirname = pn. Como pn puede tener tamaño mayor a name, esta instrucción resulta en una falla de seguridad.

Utilizamos el este artículo como fuente de estudio sobre las vulnerabilidades relacionadas al desbordamiento de buffer y sus implementaciones, que nos demandó una gran investigación. El artículo data del 1996, y si bien tiene información más que suficiente y clara sobre las vulnerabilidades relacionadas al desbordamiento de buffer y sus implementaciones, el compilador de C a cambiado desde que fue escrito. Esto implicó que los ejemplos que tiene no funcionen sin requerirnos un claro entendimiento de lo que estábamos haciendo, ya que debíamos hacerlos funcionar para la forma en que ahora el compilador hace sus alineamientos, ordena sus variables y demás. Esto si bien nos requirió un gran trabajo, profundizó nuestros conocimientos sobre el manejo que hacen los programas escritos C de la memoria.

Exploits (stack ejecutable):

exploit-shell-client.py: Se aprovecha de la vulnerabilidad de process_client(), reemplazando la dirección de retorno de la función por la dirección donde se encuentra nuestro shellcode. Abre una shell del lado del servidor y el servidor se cae al cerrarla.

exploit-unlink-client.py: Se aprovecha de la vulnerabilidad de process_client(), reemplazando la dirección de retorno de la función por la dirección donde se encuentra nuestro shellcode. Elimina el archivo que se encuentra en "*/home/httpd/grades.txt*" y luego se cae el servidor.

exploit-shell-header.py: Se aprovecha de la vulnerabilidad de http_request_headers(), reemplazando la dirección de retorno de la función por la dirección donde se encuentra nuestro shellcode. Abre una shell del lado del servidor y el servidor no se cae.

exploit-unlink-header.py: Se aprovecha de la vulnerabilidad de http_request_headers(), reemplazando la dirección de retorno de la función por la dirección donde se encuentra nuestro shellcode. Elimina el archivo que se encuentra en "*/home/httpd/grades.txt*" y luego no se cae el servidor.

exploit-shell-serve.py: Se aprovecha de la vulnerabilidad de http_server(), cambiando el valor de un puntero a función por la dirección donde se encuentra nuestro shellcode. Abre una shell del lado del servidor y el servidor no se cae.

exploit-unlink-serve.py: Se aprovecha de la vulnerabilidad de http_server(), cambiando el valor de un puntero función por la dirección donde se encuentra nuestro shellcode. Elimina el archivo que se encuentra en "*/home/httpd/grades.txt*" y luego no se cae el servidor.

Exploits (Return-to-libc)

Cuando no es posible ejecutar código de pila, es decir, cuando esta no es ejecutable, una alternativa para tomar el control del programa es mediante return-to-libc. Este método consta de cambiar la dirección de retorno de alguna función, desbordando algún buffer, por la dirección de alguna función de libc y sus argumentos.

Comenzamos intentando reemplazar la return address para que apunte a la función de libc system() y así ejecutar /bin/sh. Tuvimos inconvenientes para encontrar la dirección de system(), ya que la buscábamos desde un programa corriendo fuera del servidor, es decir, en un entorno distinto al del proceso del servidor. Hasta que logramos encontrar la dirección correcta desde gdb adjuntado al proceso del servidor, y con el comando 'p system' fue trivial hallarla. Luego necesitábamos pasarle el argumento, es decir, una string "/bin/sh" y decidimos hacerlo mediante una variable de entorno. La función http_request_headers() (http.c) por cada header field '[fieldname]: [field-value]' crea una variable de entorno llamada HTTP_fieldname(convertido a mayúsculas) y con el valor field-value. Entonces encontramos la dirección de estas variables de entorno y las utilizamos para explotar mediante return-to-libc.

exploit-sh-libc.py: Se aprovecha de la vulnerabilidad de http_request_headers(), reemplazando la dirección de retorno de la función por la dirección de system() seguido de la dirección de retorno de system() y la dirección del argumento de system(). Abre una shell del lado del servidor y el servidor no se cae.

exploit-unlink-libc.py: Se aprovecha de la vulnerabilidad de http_request_headers(), reemplazando la dirección de retorno de la función por la dirección de system() seguido de la dirección de retorno de system() y la dirección del argumento de system(). Elimina el archivo que se encuentra en "*/home/httpd/grades.txt*" y luego no se cae el servidor.

Solución a las vulnerabilidades

Las vulnerabilidades encontradas son debido al uso de funciones de C que copian datos de un lugar a otro sin verificar cual es el espacio reservado para donde se está escribiendo. Naturalmente, la solución para esto fue asegurarnos de que se verifique el espacio reservado para donde se está escribiendo, para no escribir en un espacio que no esté reservado para tal escritura.

process_client(): Para solucionar esta vulnerabilidad, modificamos url_decode() agregando un tercer argumento que representa el tamaño del array de destino. También cambiamos la condición del for para que copie como límite hasta el máximo del array de destino o hasta el final de la cadena de origen.

http_request_headers(): Misma solución que process_client().

http_serve(): Cambiamos strcat() por strncat(), pasándole como tercer argumento la diferencia entre el tamaño máximo de pn y el tamaño actual de pn (strncat() concatena como máximo la cantidad de bytes pasadas en el tercer argumento).

Shellcode

Es código máquina representado en hexadecimal. Lo obtenemos escribiendo las instrucciones que deseamos en assembly, compilando y desde GDB copiando el valor hexadecimal de cada instrucción.

Ejemplo de shellcode para abrir una terminal:
\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68

Ejemplo de shellcode para eliminar el archivo /home/httpd/grades.txt:
/home/httpd/grades.txt:\xeb\x15\x5e\x31\xc0\x88\x46\x16\xb0\x0b\xfe\xc8\x89\xf3\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xe6\xff\xff\xff\x2f\x68\x6f\x6d\x65\x2f\x68\x74\x74\x70\x64\x2f\x67\x72\x61\x64\x65\x73\x2e\x74\x78\x74

ASLR

La aleatoriedad en la disposición del espacio de direcciones (conocida por las siglas en inglés ASLR) es una técnica de seguridad informática relacionada con la explotación de vulnerabilidades basadas en la corrupción de memoria. Con el fin de impedir que un atacante salte de forma fiable a, por ejemplo, una función explotable en concreto de la memoria, ASLR dispone de forma aleatoria las posiciones del espacio de direcciones de las áreas de datos clave de un proceso, incluyendo la base del ejecutable y las posiciones de la pila, el heap y las librerías.

Algunos métodos para vulnerar esta protección son:

- Return-to-plt
- Brute force
- GOT overwrite and GOT dereference

Conclusiones y extensiones

Una conclusión inmediata es que hay tomar precauciones al escribir código en C, al trabajar con funciones que escriben en arrays, en el sentido de que nunca debemos copiar una cantidad de datos mayor al tamaño del array. Hacer esto trae graves fallas de seguridad en nuestros programas.

Una extensión al trabajo podría ser la implementación de exploits similares pero sobre una arquitectura de 64bits, que demandaría una investigación del comportamiento de la memoria sobre dicha arquitectura.

Otra posible extensión sería ser capaces de manejar la shell abierta del lado del servidor desde el lado del cliente, lo que requeriría crear un canal de comunicación entre el servidor y el cliente.