UNIVERSIDAD DE MÁLAGA

PROGRAMACIÓN SEGURA

Práctica 2: buffer overflows y strings

CIBERSEGURIDAD E INTELIGENCIA ARTIFICIAL

Pablo Jesús Delgado Muñoz

Octubre de 2024

ÍNDICE GENERAL

CAPÍTULO I: BUFFER OVERFLOW

1.1 Definición y explicación

El **buffer overflow** o desbordamiento de búfer es un fenómeno que ocurre cuando un programa intenta almacenar más datos en un bloque de memoria (*buffer*) del que este puede contener. Los búferes son segmentos de memoria reservados específicamente para almacenar datos temporalmente. Estos datos pueden variar ampliamente, desde entradas del usuario, datos recibidos de un archivo, hasta paquetes de información que forman parte de comunicaciones en red. Sin embargo, cuando los datos que se intentan almacenar sobrepasan la capacidad asignada al búfer, se produce un desbordamiento. En esta situación, los datos excedentes se expanden hacia otras zonas de la memoria, sobrescribiendo potencialmente información crítica que el programa no tenía intención de modificar, lo cual puede alterar su funcionamiento.

Este tipo de vulnerabilidad es particularmente común en lenguajes de programación como `C` o `C++`, donde la gestión de la memoria no es controlada de manera automática por el lenguaje. En estos lenguajes, la memoria asignada a cada variable es administrada manualmente por el programador o, de manera predeterminada, sin verificación de límites. Cuando se intenta escribir más información en un búfer del que este puede albergar, los datos adicionales pueden sobrescribir variables adyacentes, estructuras de control del programa, o incluso direcciones de retorno en la pila de memoria. En el peor de los casos, esto puede hacer que el programa falle de manera catastrófica, se comporte de manera errática, o, en los casos más graves, ejecute código malicioso que haya sido inyectado por un atacante. Esto último es particularmente peligroso cuando el desbordamiento afecta regiones de memoria sensibles o protegidas del sistema, que el programa no tiene autorización para modificar.

Los buffer overflows no siempre son fáciles de detectar, ya que pueden no causar problemas visibles durante la ejecución normal del software, y es posible que pasen desapercibidos por largos periodos de tiempo. Sin embargo, la ausencia de síntomas evidentes no implica que el problema sea menor o inofensivo. Un desbordamiento puede generar fallos importantes en el programa y en la integridad de los datos, especialmente en aplicaciones que manejan grandes volúmenes de datos o que se ejecutan durante largos periodos de tiempo. Estos fallos pueden comprometer la fiabilidad, seguridad y usabilidad del software.

1.1.1 Por qué es un problema grave

El **desbordamiento de búfer** es considerado una vulnerabilidad grave en el ámbito de la seguridad informática, y no necesariamente debe ser explotado por un atacante para que represente un riesgo significativo. A menudo, los desbordamientos de búfer pueden estar presentes en un sistema durante mucho tiempo sin ser detectados, y sin provocar consecuencias inmediatas. Esto no significa que su presencia sea inofensiva; de hecho, un desbordamiento de búfer puede causar problemas muy serios en el funcionamiento de un programa en diversas formas.

Un aspecto importante a tener en cuenta es que un desbordamiento de búfer puede corromper los datos almacenados en áreas de la memoria que son esenciales para el funcionamiento adecuado del software. Este tipo de corrupción de datos puede llevar a **fallos aleatorios** en el programa, los cuales son notoriamente difíciles de depurar. Estos fallos suelen manifestarse solo en circunstancias específicas, como cuando se introducen valores de entrada que exceden el tamaño permitido por el búfer. Esto da lugar a lo que se conoce como **errores intermitentes**, que son complicados de diagnosticar debido a que no se producen de forma constante, y no siempre es sencillo reproducir las condiciones que los desencadenan.

Un desbordamiento de búfer también puede **interrumpir el flujo de ejecución del programa**, provocando bloqueos, pérdida de datos o respuestas inesperadas. Esto es problemático no solo porque afecta la experiencia del usuario, sino porque compromete la estabilidad general del sistema en el que el programa se está ejecutando. Un programa que falla ocasionalmente en ciertas circunstancias puede afectar a otros procesos y servicios que dependan de él, generando una **cascada de fallos** en otros componentes del sistema. Esto puede llevar a un colapso total en aplicaciones de misión crítica, donde la confiabilidad y la estabilidad son esenciales para el funcionamiento del sistema en su conjunto.

Más allá de los problemas de integridad de datos y estabilidad, el desbordamiento de búfer presenta un **riesgo significativo de seguridad**. En situaciones donde un atacante puede controlar los datos que se ingresan en un búfer, puede aprovechar el desbordamiento para modificar la memoria del sistema de forma deliberada metiendo shellcode. En algunos casos, el atacante podría alterar el flujo de ejecución del programa para ejecutar código arbitrario(shellcode), dándole la posibilidad de tomar el control total del sistema en el que el programa vulnerable se está ejecutando. Por esta razón, los desbordamientos de búfer son uno de los vectores de ataque más comunes en exploits y ataques de escalación de privilegios.

CAPÍTULO II: DETECCIÓN DEL PROBLEMA

En este capítulo vamos a adentrarnos en el análisis, detección y busca de soluciones para el buffer overflow. Primero veremos una serie de herramientas muy útiles, los procesos realizados para detectar estas vulnerabilidades, impacto de estas mismas.

2.1 Herramientas utlilizadas

Para abrir, analizar y leer código, podemos utilizar una gran variedad de editores de texto y combinar estos con diferentes herramientas que facilitan el trabajo de desarrollo y depuración. Destacamos **Vim** , **SublimeText** o **Kate** como editores de texto.

Para facilitar la depuración, emplearemos **Valgrind**, un conjunto de herramientas muy útil para encontrar errores de administración de memoria y subprocesamiento en código.

Cuando recibimos código para revisar en busca de errores, muchas veces podemos encontrar que este no se encuentra correctamente indentado u organizado. Por lo tanto, para evitar cometer errores organizando el código por nosotros mismos, emplearemos **Indent**.

2.2 Análisis código original

Primero que nada vamos a abrir el código proporcionado por primera vez, usaremos Vim. Nos movemos a la carpeta donde se encuentra en código y lo abrimos.

```
ps@programacionsegura:~/Downloads/Practica2BufferOverflosyStrings$ vim practica1_00_original.c
```

```
#include <stdio.h>
#include <string.h>
int main(){char qfoo[9];char q0[9];int qbar=0;char qbaz;printf(
"Introduce el DNI caracter a caracter y presiona Enter para terminar):\n");
while((qbaz=getchar())!='\n'){qfoo[qbar++]=qbaz;}qfoo[qbar]='\0';strcpy(q0,
qfoo);printf("El DNI introducido es: %s\n",q0);return 0;}
~
```

Podemos observer que el código no está totalmente ordenado, por lo tanto emplearemos el comando Indent para evitar errores y hacer de forma automática el indentado del código.

```
ps@programacionsegura:~/Downloads/Practica2BufferOverflosyStrings$ indent practica1_00_original.c
ps@programacionsegura:~/Downloads/Practica2BufferOverflosyStrings$ ls
'apuntes practica2'   practica   practica1_00.c   practica1_00.c~   _practica1_00_original.c   practica1_00_original.c~   practica1_copia.c   practica_gdb
```

Abrimos el código de nuevo y podemos ver que ya está correctamente formateado.
Podemos ahora observar un fragmento de código en C que al parecer lee un DNI caracter a caracter, almacenándolo, y tras leer lo vuelve a imprimir.

```
#include <stdio.h>
#include <string.h>
int
main ()
{
  char qfoo[9]; //reservamos 9 caracteres en qfoo
  char q0[9]; //reservamos 9 caracteres en q0
  int qbar = 0; //inicializamos qbar a 0
  char qbaz;  //creamos variable qbaz de tipo caracter
  printf
    ("Introduce el DNI caracter a caracter y presiona
    Enter para terminar):\n");
  while ((qbaz = getchar ()) != '\n')
    //en qbaz guardamos lo que leemos y entramos al bucle
    {
      qfoo[qbar++] = qbaz;
      //guardamos en cada iteración del while (donde qbar
    }
  qfoo[qbar] = '\0'; //ponemos el caracter nulo en la
  //ultima posicion de qfoo (qbar)
  strcpy (q0, qfoo);  //hacemos strcpy de qfoo a q0, para
  printf ("El DNI introducido es: %s\n", q0);
  return 0;
}
```

Primero crea un array qfoo de 9 caracteres, q0[9], que es similar a qfoo y se crea un entero qbar inicializado en 0.
Ahora tras incializar los datos necesarios, el código imprime el mensaje que pide que se introduzca el DNI, y dentro del while se accederá a qfoo usando qbar como indice aumentando este uno a uno para así moverse dentro de qfoo. Asignará en cada posición el valor de qbaz, que es el caracter introducido en cada interación del bucle.
Ya por último se introduce el carácter nulo en qfoo y se usa strcpy para copiar el string de qfoo a q0, y se imprime q0 por pantalla.

2.3 Proceso de detección del problema

Para detectar el problema vamos a ver si el problema ”parece”hacer lo que debe:
Compilamos y ejecutamos:

```
Introduce el DNI caracter a caracter y presiona Enter para terminar):
33333333P
El DNI introducido es: 33333333P
```

A simple vista, parece que el programa funciona correctamente, pero debemos tener en cuenta que cuando un programa entra para buscar problemas con él, ha pasado ciertas pruebas, por lo tanto, es normal que .ªparentemente”funcione correctamemte. Vamos entonces a adentrarnos en materia y analizar el programa usando **Valgrind**.

Primero compilamos el programa de nuevo pero con el flag -g para obtener trazas de debugging y poder usar valgrind.

```
ps@programacionsegura:~/Documents/Prac2PS$ gcc -g practica1_00.c -o practica1_debugging
```

Ejecutamos ahora el programa con valgrind...

El DNI introducido es: 888888888

==5378== HEAP SUMMARY:
==5378== in use at exit: 0 bytes in 0 blocks
==5378== total heap usage: 2 allocs, 2 frees, 2,048 bytes allocated
==5378== All heap blocks were freed – no leaks are possible
==5378== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
==5378== 1 errors in context 1 of 1:
==5378== Source and destination overlap in strcpy(0x1ffefffdd9, 0x1ffefffde2)
==5378== at 0x484696F: strcpy (vg_replace_strmem.c:553)
==5378== by 0x1091C6: main (practica1_00.c:18)
==5378== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)

Podemos observar como al introducir 9 caracteres el programa da error, mientras que si introducimos 8 caracteres no lo dará, por lo tanto se ha producido un buffer overflow.

El DNI introducido es: 8888888V
==5382== HEAP SUMMARY:
==5382== in use at exit: 0 bytes in 0 blocks
==5382== total heap usage: 2 allocs, 2 frees, 2,048 bytes allocated
==5382== All heap blocks were freed – no leaks are possible
==5382== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

Los array qfoo y `q0` están diseñados para almacenar un máximo de 9 caracteres. Sin embargo, cuando introduces un DNI de 9 dígitos, no queda espacio para el carácter nulo (`\0`) que indica el final de una cadena en C. Al intentar copiar el contenido de `qfoo` a `q0` usando `strcpy`, se produce un desbordamiento porque se intenta copiar un carácter más del que cabe en `q0`.

¿Por qué no ocurre el error con 8 caracteres?
Cuando introduces 8 caracteres, el carácter nulo se almacena en la novena posición del arreglo, dejando espacio suficiente para ambos. Al utilizar `strcpy`, se copia correctamente la cadena a `q0` sin causar un desbordamiento.

Ahora vamos a hacer uso de la herramienta gdb para ver cómo se produce este buffer overflow:

```
ps@programacionsegura:$ gdb ./practica1\_debugging
(gdb) b 10
Breakpoint 1 at 0x11b4: file practica1\_00\_original.c, line 17.
(gdb) b 18
Breakpoint 2 at 0x11c7: file practica1\_00\_original.c, line 18.
```

Antes de causar el buffer overflow, vamos a ver dónde se escribe en memoria:

```
(gdb) x/64wx \$rsp
0x7fffffffde40: 0x00000000 0x00000000 0x00000000 0x00000000
0x7fffffffde50: 0x00000000 0x00000000 0x00000000 0x00000000
0x7fffffffde60: 0x00000001 0x00000000 0xf7de424a 0x00007fff
```

Ahora cuando introducimos datos

```
Breakpoint 1, main () at practica1\_00\_original.c:17
17        \textbf{strcpy} (q0, qfoo);
(gdb) x/10wx \$rsp
0x7fffffffde40: 0x00000000 0x00000000 0x00000000 0x00000000
0x7fffffffde50: 0x39390000 0x39393939 0x0a003939 0x00000008
```

Saltamos a la linea 18 donde se hace strcpy...

```
Breakpoint 3, main () at practica1\_00\_original.c:18
18   \textbf{printf} ("El DNI introducido es:, q0);

(gdb) print qfoo
\$2 = "99999999" OBSERVAMOS EL VALOR EN QFOO TRAS EL STRCPY

(gdb) x/64wx \$rsp

0x7fffffffde40: 0x00000000 0x00000000 0x39393900 0x39393939
0x7fffffffde50: 0x39390039 0x39393939 0x0a003939 0x00000008
```

```
0x7fffffffde60: 0x00000001 0x00000000 0xf7de424a 0x00007fff
```

Si comparamos, podemos ver las posiciones de memoria en qfoo, pero ahora vamos a hacerlo pasándonos de datos a ver que ocurre:

```
Introduce el DNI caracter a caracter y presiona Enter para terminar):
99999999999999999999999
99999999999999999
999999999999999999999999
```

```
Breakpoint 1, main () at practica1\_00\_original.c:17
```

```
17  \textbf{strcpy} (q0, qfoo);
```

```
(gdb) x/10wx \$rsp
```

```
0x7fffffffde40: 0x00000000 0x00000000 0x00000000 0x00000000
0x7fffffffde50: 0x39390000 0x39393939 0x0a393939 0x00000074
0x7fffffffde60: 0x00000001      0x00000000
```

Aqui lo podemos observar antes del copy...

```
(gdb) x/10wx \$rsp
```

```
0x7fffffffde40: 0x00000000 0x00000000 0x39393900 0x39393939
0x7fffffffde50: 0x740a3939 0x39393900  0x0a393939 0x00000074
0x7fffffffde60: 0x00000001 0x00000000
```

```
(gdb) print qfoo
```

```
\$1 = "\nt\n000999999"
```

Podemos observar como lo que se encuentra en memoria ya comienza a fluctuar respecto de lo que deberíamos de ver si no hubiera problemas.

2.4 Ubicación y explicación del problema

Ahora vamos a analizar el código para ver dónde se produce este buffer overflow. Como hemos visto al inicio del programa, se han reservado 9 posiciones en memoria para almacenar el string, pero claro, un DNI ocupa 9 posiciones, por lo tanto no se ha tenido en cuenta el carácter nulo, que `strcpy` usa para determinar el final de la cadena.

Vamos a explicar los pasos que han llevado a este desbordamiento:

1. **Almacenamiento del DNI en qfoo**
2. **Uso de strcpy / no controlar el límite de lo que se lee**
3. **Sobrescritura en memoria**

Vamos a observar el array antes y después de usar strcpy:

```
Antes de strcpy:
qfoo[9]:  |1|2|3|4|5|6|7|8|9|  (sin '\0')
q0[9]:    | | | | | | | | |  (vacío)

Después de strcpy (incorrecto):
qfoo[9]:  |1|2|3|4|5|6|7|8|9|  (sin '\0')
q0[9]:    |1|2|3|4|5|6|7|8|9|\0|  (desbordamiento)
```

2.5 Impacto del desbordamiento en la ejecución del programa

Aunque el programa pueda seguir ejecutándose después de un desbordamiento de búfer, es fundamental comprender que esto no significa que el programa esté funcionando correctamente. Los efectos del desbordamiento pueden ser impredecibles y difíciles de detectar, pero siempre representan una amenaza para la seguridad y la estabilidad del programa.

Esto se debe básicamente a que en este programa, como es sencillo, no llegamos a hacer más cosas que imprimir únicamente el dni que hemos introducido, por lo tanto no hay mucha posibilidad de que este buffer overflow afecte al funcionamiento simple del programa.

Introduce el DNI caracter a caracter y presiona Enter para terminar):
999999999

CAPÍTULO III: SOLUCIONES

3.1 Soluciones generales

Los desbordamientos de búfer representan una amenaza seria en el desarrollo de software, especialmente en lenguajes de bajo nivel como C, que no realiza verificaciones automáticas de los límites de la memoria.
A continuación, se presentan una serie de prácticas y herramientas que se utilizan ampliamente para prevenir y gestionar eficazmente los desbordamientos de búfer:

1. **Uso de Funciones de Manipulación de Cadenas Seguras**
   Las funciones de manipulación de cadenas en C, como `strcpy`, `sprintf`, y `strcat`, son conocidas por su vulnerabilidad a desbordamientos de búfer, ya que no tienen en cuenta los límites de la memoria asignada para las cadenas destino. Para minimizar el riesgo de desbordamiento de búfer y mejorar la seguridad del código, es fundamental adoptar funciones seguras que permitan controlar la cantidad de datos que se copian o concatenan en cada operación. Funciones como `strncpy` y `snprintf`, que permiten especificar la longitud máxima de los datos a copiar o formatear, son alternativas más seguras que deberían ser preferidas siempre que sea posible. También existen bibliotecas modernas y funciones especializadas en manipulación de cadenas seguras que están diseñadas específicamente para reducir este riesgo.

2. **Validación de la Longitud de las Entradas**
   Validar la longitud de las entradas antes de almacenarlas en búferes de tamaño fijo es una práctica esencial en el desarrollo seguro de software. Esto implica verificar que los datos ingresados por el usuario o provenientes de otras fuentes externas no excedan la capacidad del búfer, evitando así la sobrescritura de áreas de memoria adyacentes. Esta validación debe hacerse tanto en tiempo de compilación como en tiempo de ejecución para garantizar que el programa funcione correctamente bajo condiciones normales y ante entradas inesperadas. En algunos casos, es conveniente establecer límites de longitud fijos y claros en la especificación del programa para facilitar esta tarea.

3. **Asignación Dinámica de Memoria (cuando es posible)**
   La asignación dinámica de memoria mediante funciones como `malloc`, `calloc`, y `realloc` en C permite al programa ajustar el tamaño del búfer de acuerdo con la cantidad de datos que necesita manejar en tiempo de ejecución. Esto reduce el riesgo de desbordamientos de búfer, ya que el tamaño de la memoria puede aumentar según la demanda del programa, en lugar de depender de un tamaño estático. Sin embargo, es importante recordar que un uso incorrecto de la memoria dinámica puede introducir otros problemas de seguridad, como filtraciones de memoria (*memory leaks*) o uso de memoria no inicializada. Para evitar esto, los desarrolladores deben asegurarse de liberar correctamente la memoria asignada con `free` después de que ya no sea necesaria y de inicializar adecuadamente cualquier memoria asignada.

4. **Control y Seguimiento del Uso de Memoria**
   Monitorear y controlar el uso de la memoria del programa es otra estrategia efectiva para prevenir desbordamientos de búfer. Existen herramientas de análisis y depuración de memoria, como *Valgrind*, que permite detectar posibles errores de memoria, incluyendo accesos fuera de los límites del búfer y uso de memoria sin inicializar. Estas herramientas pueden integrarse en el flujo de trabajo de desarrollo y ejecutarse durante pruebas exhaustivas para detectar y corregir vulnerabilidades antes de la fase de producción. Además, muchas herramientas de análisis de memoria proporcionan información detallada sobre el estado del programa, lo que permite identificar fácilmente áreas del código propensas a desbordamientos de búfer.

5. **Desarrollo Seguro y Prácticas de Programación Defensiva**
   Implementar prácticas de programación defensiva y seguir principios de desarrollo seguro es fundamental para reducir el riesgo de desbordamiento de búfer y otros errores de seguridad. Esto incluye escribir código robusto que prevea posibles condiciones de error y situaciones de uso inesperadas, de modo que el programa responda de manera controlada y segura. La programación defensiva también implica una revisión constante del código para verificar que no existan accesos indebidos a la memoria y que los límites de los búferes se respeten en todas las operaciones de lectura y escritura. Además, el uso de revisiones de código y auditorías de seguridad puede ayudar a identificar problemas de memoria y prácticas inseguras en el código, lo cual es una excelente práctica para reducir errores en etapas tempranas del desarrollo.

3.2 Solución específica

Para solucionar el buffer overflow en este programa, tendremos en cuenta los fallos que se han cometido hasta ahora:

1. **No tener en cuenta el caracter nulo.**
2. **Uso de función de copia de string insegura, no se controla el máximo de lectura**

1. Vamos a modificar el código de tal forma que qfoo tenga espacio para 9 + 1 caracteres, incluyendo así el caracter nulo.

```
int
main ()
{
  char qfoo[9+1]; //Incluyendo caracter nulo
  char q0[9+1]; //Incluyendo caracter nulo
  int qbar = 0;
  char qbaz;
```

2. Ahora vamos a modificar el bucle while donde se lee de tal forma que no se pueda escribir más de lo que se debería en qfoo.

```
#include <stdio.h>
#include <string.h>
int
main ()
{
  char qfoo[9+1]; //Incluyendo caracter nulo
  char q0[9+1]; //Incluyendo caracter nulo
  int qbar = 0;
  char qbaz;
  printf
    ("Introduce el DNI caracter a caracter y presiona
    Enter para terminar):\n");
  while ((qbaz = getchar ()) != '\n' && qbar<9) //Para que no
  //se escriba más de 9 caracteres de DNI
    {
      qfoo[qbar++] = qbaz;
    }
  qfoo[qbar] = '\0';
  strcpy (q0, qfoo);
  printf ("El DNI introducido es: %s\n", q0);
  return 0;
}
```

Por úiltimo vamos a pasarle valgrind para ver si se encuentra algún error. Introduce el DNI caracter a caracter y presiona Enter para terminar):
999999999999999999999999999999999999999999999999
El DNI introducido es: 999999999
==4366== HEAP SUMMARY:
==4366== in use at exit: 0 bytes in 0 blocks
==4366== total heap usage: 2 allocs, 2 frees, 2,048 bytes allocated
==4366==
==4366== All heap blocks were freed – no leaks are possible
==4366==
==4366== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
Podemos observar que **no ocurre ningún error** ahora tras escribir de más.