Vulnerability, Discovery and Exploitation (VDE)

Coursework

REPORT

Student No.: UP2009045
Word Count: 2099

Word Count

Section	Word Count
Introduction	141
Vulnerability Analysis	77
User Modification Code (File 1)	163
Unused Function	77
Format String	161
Delete User	166
Heap Overflow	146
Buffer Overflow	120
Database Code (File 2)	54
Heap Overflow	251
Vulnerability Analysis of Vulnerable Code	33
Symmetric Algorithm & Chosen Vulnerabilities	100
Buffer Overflow	157
Heap Overflow	107
Format String	91
Accessible Unencrypted Text	106
Conclusion	149
Total	2099

Contents

Introduction 4

Vulnerability Analysis 4

User Modification Code (File 1) 4

Unused function 6

Format String 6

Delete User Function 7

Heap Overflow 9

Buffer Overflow 10

Database Code (File 2) 10

Heap Overflow 11

Vulnerability Analysis of Vulnerable Code 14

Symmetric Algorithm & Chosen vulnerabilities 14

Analysis of code 14

Buffer Overflow 14

Heap Overflow 15

Format String 16

Accessible Unencrypted text 16

Conclusion 17

Appendix 18

C Code 18

Introduction

Throughout this report, we cover analysis of the two vulnerable files provided along with a symmetric encryption algorithm with implemented vulnerabilities. Examination of the vulnerable code, vulnerabilities and potential exploitation where possible will be carried out in each section, this will include an overarching view of how the initial code works in its current state, how each vulnerability works finally with suggestions, where applicable, how the attack scope can be minimised.

The tools used to perform this analysis were as follows:

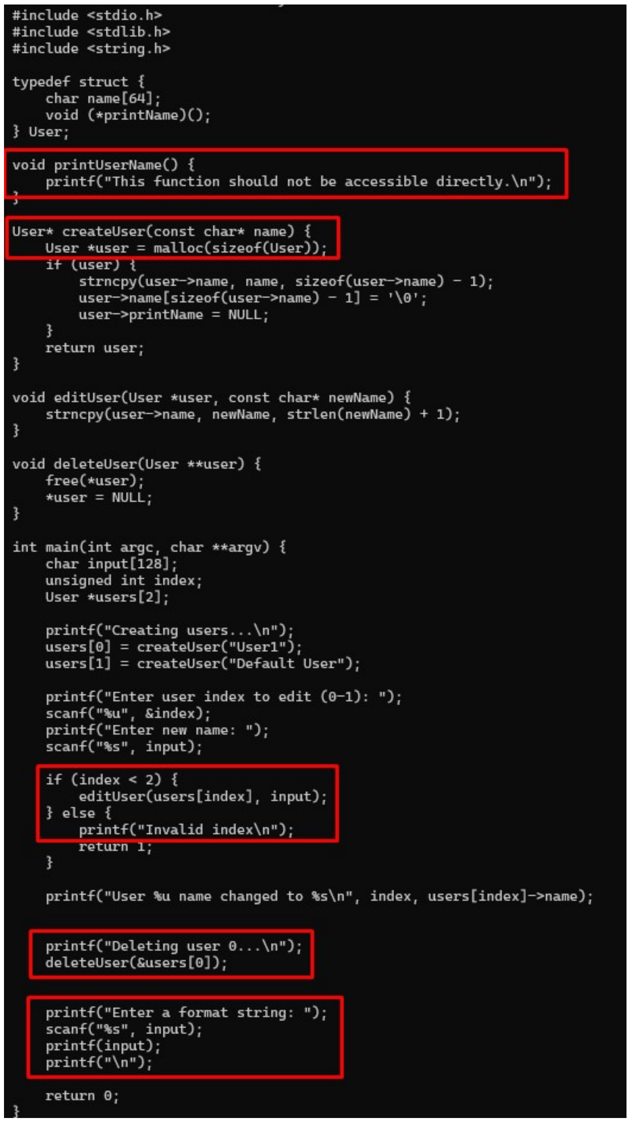
- GNU C Compiler – Used to build and compile code into an executable format
- GNU De-Bugger – Used for analysing and exploiting aspects of the code
- C – Programming language used for creating vulnerable code
- Linux (Ubuntu 22.04.4) – Testing code in Linux environment
- Msys2 Mingw – Installation of GCC + GDB on Windows
- Visual studio Code – Development Environment

Vulnerability Analysis

In this section we will cover both files provides, labelled file1.exe and file2.exe. The first file consists of code for user modification program and the second a simple database program. Both of these executables will be explored using GNU De-Bugger, reviewing the code behind the executable, what it does, potential vulnerabilities and exploitation of these vulnerabilities where possible. Should exploitation not be possible, a simple explanation of what could be possible with the vulnerability will be provided.

User Modification Code (File 1)

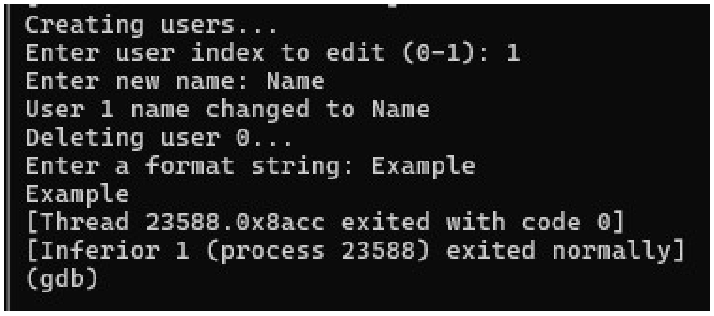
Beginning with file1.exe, a quick look at the code reveals some obvious vulnerabilities which are able to be exploited immediately to reveal information that should not otherwise be accessible. There are some other vulnerabilities in that require further analysis of the code to find.



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

typedef struct {
    char name[64];
    void (*printName)();
} User;

void printUserName() {
    printf("This function should not be accessible directly.\n");
}

User* createUser(const char* name) {
    User *user = malloc(sizeof(User));
    if (user) {
        strncpy(user->name, name, sizeof(user->name) - 1);
        user->name[sizeof(user->name) - 1] = '\0';
        user->printName = NULL;
    }
    return user;
}

void editUser(User *user, const char* newName) {
    strncpy(user->name, newName, strlen(newName) + 1);
}

void deleteUser(User **user) {
    free(*user);
    *user = NULL;
}

int main(int argc, char **argv) {
    char input[128];
    unsigned int index;
    User *users[2];

    printf("Creating users...\n");
    users[0] = createUser("User1");
    users[1] = createUser("Default User");

    printf("Enter user index to edit (0-1): ");
    scanf("%u", &index);
    printf("Enter new name: ");
    scanf("%s", input);

    if (index < 2) {
        editUser(users[index], input);
    } else {
        printf("Invalid index\n");
        return 1;
    }

    printf("User %u name changed to %s\n", index, users[index]->name);

    printf("Deleting user 0...\n");
    deleteUser(&users[0]);

    printf("Enter a format string: ");
    scanf("%s", input);
    printf(input);
    printf("\n");

    return 0;
}
```

Figure 1 - 1.c code with highlighted vulnerabilities

Seen in Figure 1 is a full overview of the code with highlighted sections where present vulnerabilities, using 'list 1' displays code from the very beginning of the file so we can review it as a whole. The code itself has multiple functions, createUser, editUser, deleteUser and one which is not accessed printUserName with each of these containing at least one vulnerability including one in the body of the main function itself.

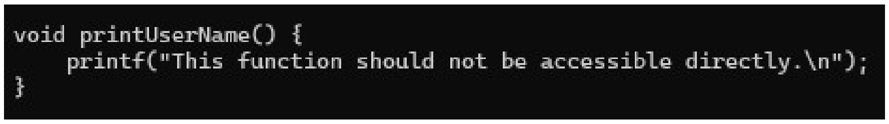


```
Creating users...
Enter user index to edit (0-1): 1
Enter new name: Name
User 1 name changed to Name
Deleting user 0...
Enter a format string: Example
Example
[Thread 23588.0x8acc exited with code 0]
[Inferior 1 (process 23588) exited normally]
(gdb)
```

Figure 2 - Running file1.exe in GDB

Running the file in GDB Figure 2 displays a user representation of the code where it populates users > asks user for a user index > a name > the wrong user is then deleted > asks user to enter a format string > program then ends.

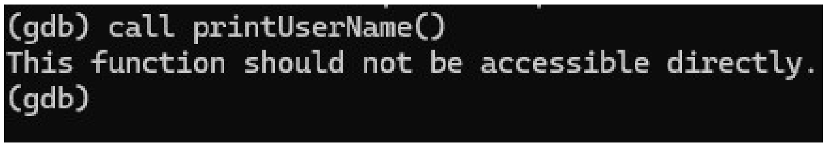
Unused function



```
void printUserName() {
    printf("This function should not be accessible directly.\n");
}
```

Figure 3 – Unused function in code

Beginning with the first vulnerability within the code and most clearly visible, the printUserName function, Figure 3, is one which is not accessed throughout the code by other functions or directly accessed.

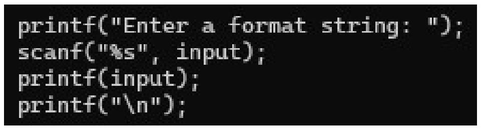


```
(gdb) call printUserName()
This function should not be accessible directly.
(gdb)
```

Figure 4 – Calling function in GDB

However, using GDB on and inserting a breakpoint at this function creates allows us to interact with the contents. Seen in Figure 4 by calling the function in GDB after stepping through each line of code, this prints the output of printUserName in our console.

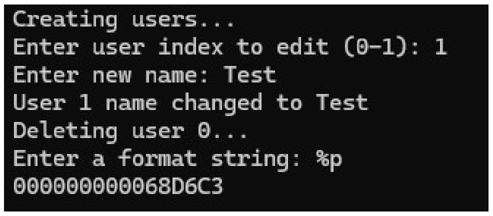
Format String



```
printf("Enter a format string: ");
scanf("%s", input);
printf(input);
printf("\n");
```

Figure 5 – Format string in code

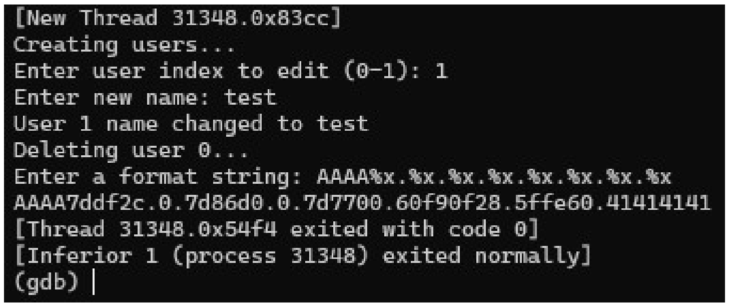
Moving onto the format string vulnerability, it can be seen in the code, Figure 5, the user is asked to input a format string, this is done using the 'scanf' function. The 'printf' function is then used to read user input simply accepting any string, then outputs the result of this directly after.



```
Creating users...
Enter user index to edit (0-1): 1
Enter new name: Test
User 1 name changed to Test
Deleting user 0...
Enter a format string: %p
000000000068D6C3
```

Figure 6 – Testing for format string vulnerability

After entering a value in to the format string field, for example %p in Figure 6, instead of printing the exact user input as printf usually would, i.e. if user were to enter "example" the printf function would output "example", this prints out a memory address '000000000068D6C3' confirming there is infact a format string vulnerability here.

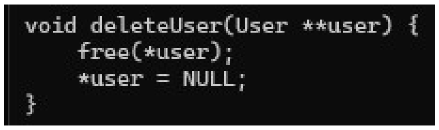


```
[New Thread 31348.0x83cc]
Creating users...
Enter user index to edit (0-1): 1
Enter new name: test
User 1 name changed to test
Deleting user 0...
Enter a format string: AAAA%x.%x.%x.%x.%x.%x.%x.%x
AAAA7ddf2c.0.7d86d0.0.7d7700.60f90f28.5ffe60.41414141
[Thread 31348.0x54f4 exited with code 0]
[Inferior 1 (process 31348) exited normally]
(gdb) |
```

Figure 7 – Reading from memory using format string

Looking further into this, by using 'AAAA%x.%x.%x.%x.%x.%x.%x.%x', Figure 7, 'AAAA' can be seen printed in hexadecimal format, '41414141' as part of the output from the printf function. This corresponds to the memory address '252e7825'. With this information an attacker should be able to write arbitrary code to this location running malicious commands.

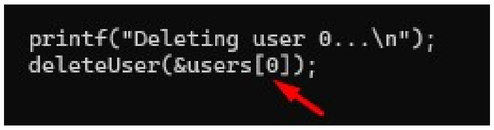
Delete User Function



```
void deleteUser(User **user) {
    free(*user);
    *user = NULL;
}
```

Figure 8 – Delete user function in code

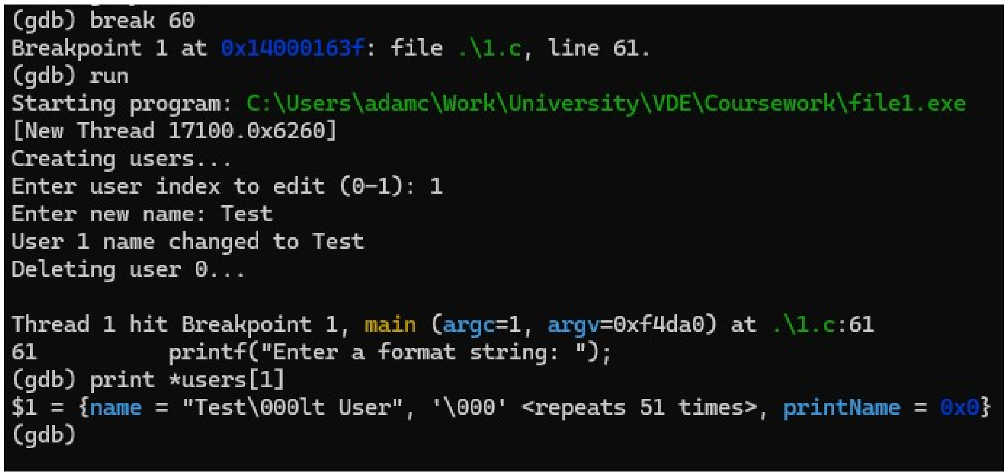
Looking over the delete user function, we can see in the code the function to delete a user has been setup correctly, freeing user in memory and setting the user variable as null in preparation to accept its new data, shown in Figure 8.



```
printf("Deleting user 0...\n");
deleteUser(&users[0]);
```

Figure 9 – User 0 permanently selected in delete user function

The function has been misused in main section of the code, identified in Figure 9, creating two issues: firstly, complete removal of user 0 regardless of which user is edited as user 0 is permanently selected and the delete user function is not using the a variable.



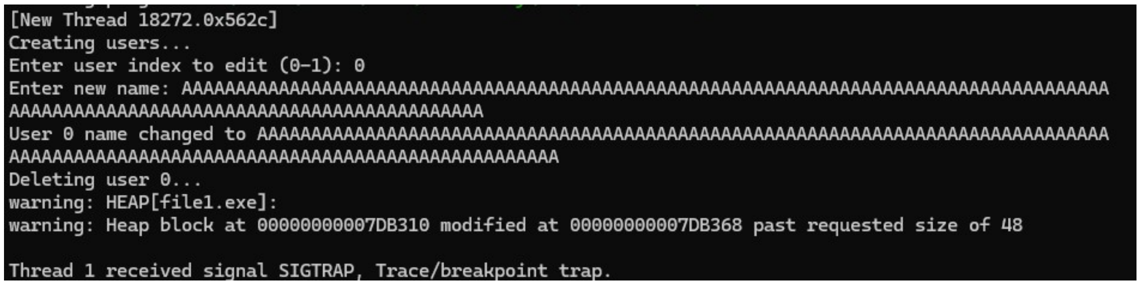
```
(gdb) break 60
Breakpoint 1 at 0x14000163f: file .\1.c, line 61.
(gdb) run
Starting program: C:\Users\adamc\Work\University\VDE\Coursework\file1.exe
[New Thread 17100.0x6260]
Creating users...
Enter user index to edit (0-1): 1
Enter new name: Test
User 1 name changed to Test
Deleting user 0...

Thread 1 hit Breakpoint 1, main (argc=1, argv=0xf4da0) at .\1.c:61
61          printf("Enter a format string: ");
(gdb) print *users[1]
$1 = {name = "Test\000lt User", '\000' <repeats 51 times>, printName = 0x0}
(gdb)
```

Figure 10 – Truncated old username with new username

Secondly, if a user other than user 0 is being modified, since operations in the code other than delete user modify the other user, the name chosen for the new user e.g., user 1, is simply amended to the name of user 0. This is shown after “print *users[1]” in Figure 10 where the entered name “Test” has been added onto “Default User” which has been truncated to “Test\000lt User”; this should only display “Test”.

Heap Overflow

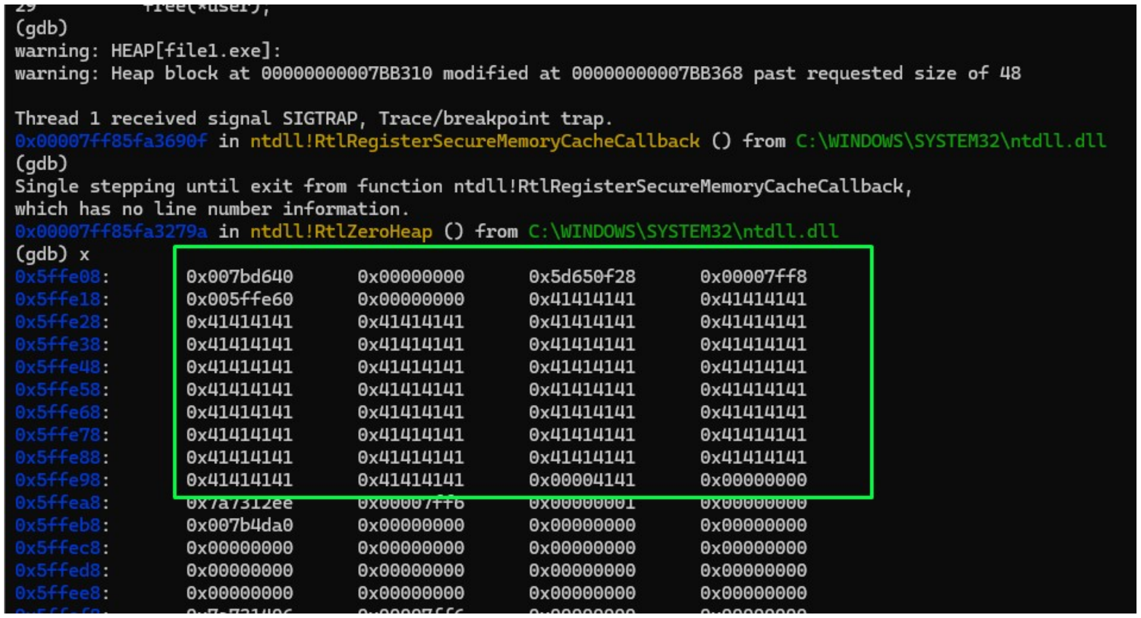
To create a user the code allocates memory at the size of the User structure we first see at the beginning of the file. Within the User structure it firstly sets a 64 byte character array “name” and secondly a pointer printName. As described in (GeeksforGeeks, 2024) pointers are dependent on the system architecture the code is being run on and since this code was run on a 64-bit system the size of the pointer is 8 bytes providing a total memory allocation of 72 bytes.



```
[New Thread 18272.0x562c]
Creating users...
Enter user index to edit (0-1): 0
Enter new name: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
User 0 name changed to AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Deleting user 0...
warning: HEAP[file1.exe]:
warning: Heap block at 00000000007DB310 modified at 00000000007DB368 past requested size of 48

Thread 1 received signal SIGTRAP, Trace/breakpoint trap.
```

Figure 11 – input which exceeds character limit

Shown in **Figure 11** by inserting a character length which extends beyond the allocation, we can see the terminal displays a warning related to the heap block due to it exceeding memory allocation size.

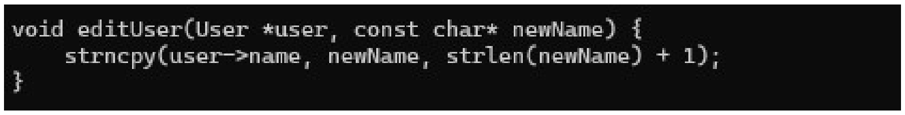


```
(gdb)
warning: HEAP[file1.exe]:
warning: Heap block at 00000000007BB310 modified at 00000000007BB368 past requested size of 48

Thread 1 received signal SIGTRAP, Trace/breakpoint trap.
0x00007ff85fa3690f in ntdll!RtlRegisterSecureMemoryCacheCallback () from C:\WINDOWS\SYSTEM32\ntdll.dll
(gdb)
Single stepping until exit from function ntdll!RtlRegisterSecureMemoryCacheCallback,
which has no line number information.
0x00007ff85fa3279a in ntdll!RtlZeroHeap () from C:\WINDOWS\SYSTEM32\ntdll.dll
(gdb) x
0x5ffe08:       0x007bd640      0x00000000      0x5d650f28      0x00007ff8
0x5ffe18:       0x005ffe60      0x00000000      0x41414141      0x41414141
0x5ffe28:       0x41414141      0x41414141      0x41414141      0x41414141
0x5ffe38:       0x41414141      0x41414141      0x41414141      0x41414141
0x5ffe48:       0x41414141      0x41414141      0x41414141      0x41414141
0x5ffe58:       0x41414141      0x41414141      0x41414141      0x41414141
0x5ffe68:       0x41414141      0x41414141      0x41414141      0x41414141
0x5ffe78:       0x41414141      0x41414141      0x41414141      0x41414141
0x5ffe88:       0x41414141      0x41414141      0x41414141      0x41414141
0x5ffe98:       0x41414141      0x41414141      0x00004141      0x00000000
0x5ffea8:       0x7a7312ee      0x00007ff8      0x00000001      0x00000000
0x5ffeb8:       0x007b4da0      0x00000000      0x00000000      0x00000000
0x5ffec8:       0x00000000      0x00000000      0x00000000      0x00000000
0x5ffed8:       0x00000000      0x00000000      0x00000000      0x00000000
0x5ffee8:       0x00000000      0x00000000      0x00000000      0x00000000
```

Figure 12 – Memory overflow seen in memory

Further investigation with GDB using “x”, a command to examine memory, displays the text entered in memory 0x41414141 being hexadecimal for ‘AAAA’ shown in **Figure 12**

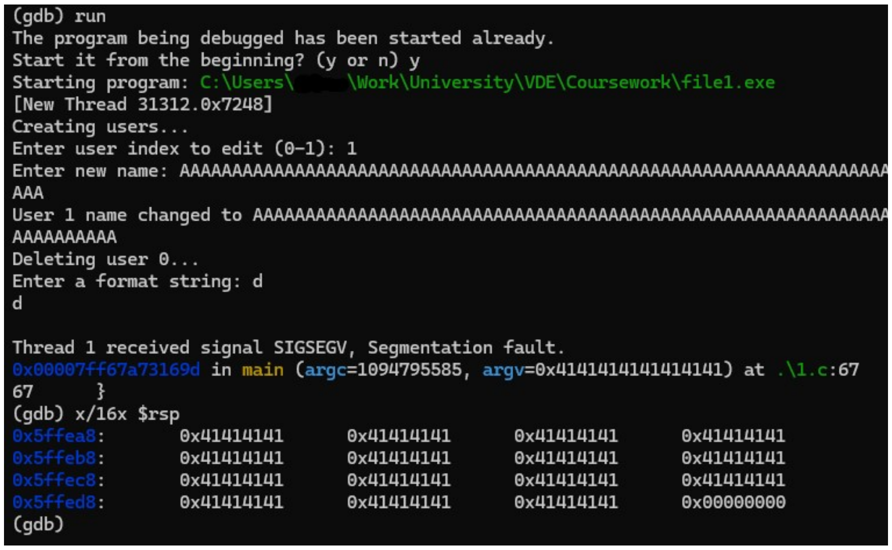
Buffer Overflow



```
void editUser(User *user, const char* newName) {
    strncpy(user->name, newName, strlen(newName) + 1);
}
```

Figure 13 – Edit user function in code

Similar to the heap overflow, editUser also has an overflow vulnerability. This function copies the new name to the name variable created in the struct which has a size previously identified as 64. This overflow vulnerability is present due to the size being taken from the string length of the user input and not the size of the name variable, input in comparison has a input length of 128 bytes.



```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: C:\Users\[redacted]\Work\University\VDE\Coursework\file1.exe
[New Thread 31312.0x7248]
Creating users...
Enter user index to edit (0-1): 1
Enter new name: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAA
User 1 name changed to AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAA
Deleting user 0...
Enter a format string: d
d

Thread 1 received signal SIGSEGV, Segmentation fault.
0x00007ff67a73169d in main (argc=1094795585, argv=0x4141414141414141) at .\1.c:67
67      }
(gdb) x/16x $rsp
0x5ffea8:       0x41414141      0x41414141      0x41414141      0x41414141
0x5ffeb8:       0x41414141      0x41414141      0x41414141      0x41414141
0x5ffec8:       0x41414141      0x41414141      0x41414141      0x41414141
0x5ffed8:       0x41414141      0x41414141      0x41414141      0x00000000
(gdb)
```

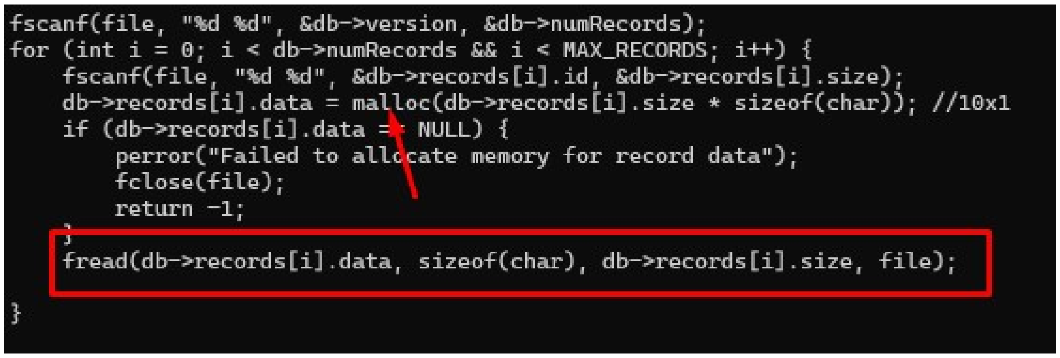
Figure 14 – Input overflow seen causing error

In this case anything above 64 bytes will cause an error as indicated by the segmentation fault in Figure 14. Also seen here is the output of the name we entered indicated in memory as "0x41414141" hexadecimal for "AAAA" displaying vulnerability within the code and that it can be exploited.

Database Code (File 2)

Next reviewing 2.c, again similar to the first file, looking at the code reveals some vulnerabilities though less obvious in this file and again these are able to be exploited to reveal information that should not otherwise be accessible. There are some other vulnerabilities in that require further analysis of the code to find.

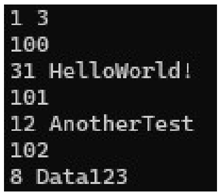
Heap Overflow



```
fscanf(file, "%d %d", &db->version, &db->numRecords);
for (int i = 0; i < db->numRecords && i < MAX_RECORDS; i++) {
    fscanf(file, "%d %d", &db->records[i].id, &db->records[i].size);
    db->records[i].data = malloc(db->records[i].size * sizeof(char)); //10x1
    if (db->records[i].data == NULL) {
        perror("Failed to allocate memory for record data");
        fclose(file);
        return -1;
    }
    fread(db->records[i].data, sizeof(char), db->records[i].size, file);

}
```

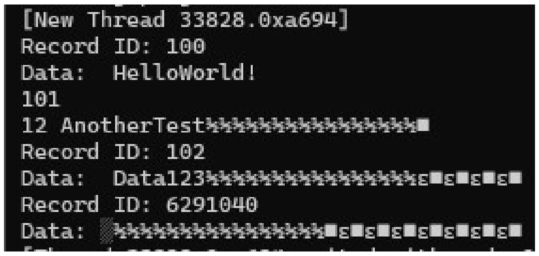
Figure 15 – for loop for loading records

Looking over 2.c’s code, we can see within the loadDatabase function records are read using the fread function, Figure 15, at a specified character length or ‘size’ from the input file. Within the for loop above, memory allocation is dependent on the record size multiplied by size of char, which as described in (GeeksforGeeks, 2023) the value of char is forever 1.



```
1 3
100
31 HelloWorld!
101
12 AnotherTest
102
8 Data123
```

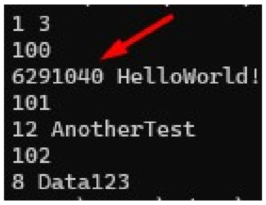
Figure 16 – Edited input file



```
 [New Thread 33828.0xa694]
 Record ID: 100
 Data:  HelloWorld!
 101
 12 AnotherTest½½½½½½½½½½½½½½½½■
 Record ID: 102
 Data:  Data123½½½½½½½½½½½½½½½½½ε■ε■ε■ε■
 Record ID: 6291040
 Data: ▒½½½½½½½½½½½½½½½½■ε■ε■ε■ε■ε■ε■ε■
```

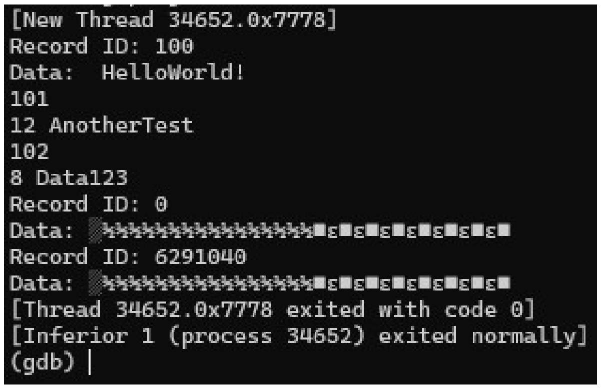
Figure 17 – Output showing ability to read next line in file

The issue here is there is no validation implemented for how long the string entered actually is. For example “ HelloWorld!” is 12 characters long but if the size defined in our input file is larger than this shown in Figure 16 it will overflow and the next record can easily be read Figure 17.



```
1 3
100
6291040 HelloWorld!
101
12 AnotherTest
102
8 Data123
```

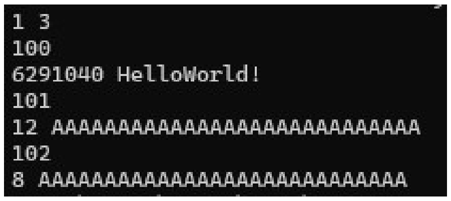
Figure 18 – Edited input with new record size



```
[New Thread 34652.0x7778]
Record ID: 100
Data:  HelloWorld!
101
12 AnotherTest
102
8 Data123
Record ID: 0
Data: ▒½½½½½½½½½½½½½½½½■ε■ε■ε■ε■ε■ε■ε■
Record ID: 6291040
Data: ▒½½½½½½½½½½½½½½½½■ε■ε■ε■ε■ε■ε■ε■
[Thread 34652.0x7778 exited with code 0]
[Inferior 1 (process 34652) exited normally]
(gdb) |
```

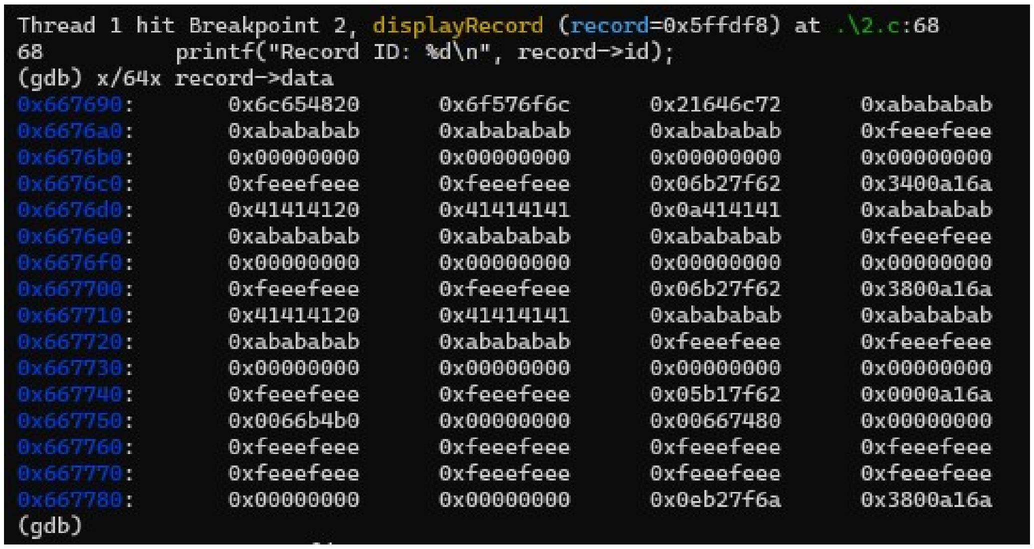
Figure 19 – Output displaying all records

Furthermore, the third record ID has changed to '6291040', using this information and increasing the size of the first record in the input file, Figure 18, we can read the whole of the next record, shown in Figure 19.



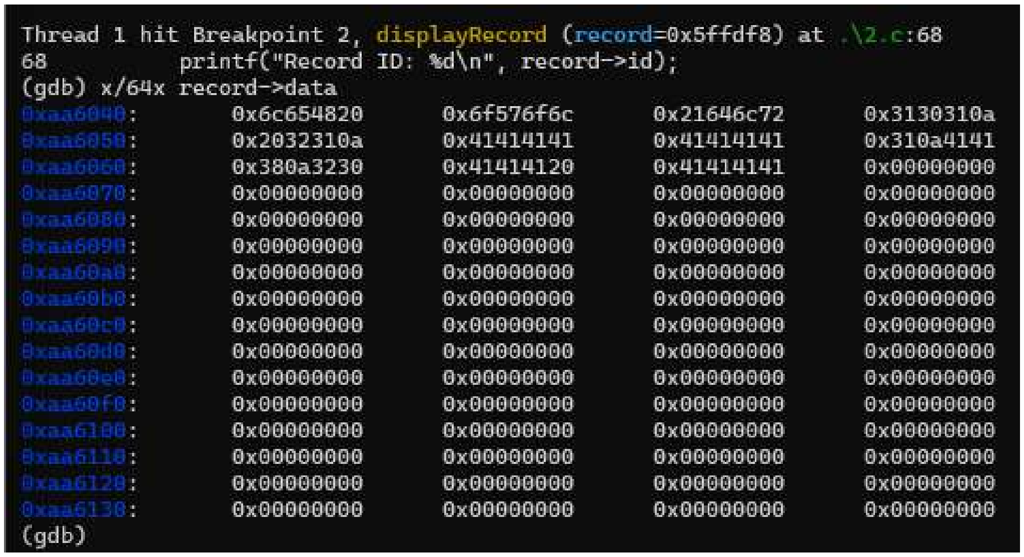
```
1 3
100
6291040 HelloWorld!
101
12 AAAAAAAAAAAAAAAAAAAAAAAAAAAA
102
8 AAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Figure 20 – Altered input file



```
Thread 1 hit Breakpoint 2, displayRecord (record=0x5ffdf8) at .\2.c:68
68          printf("Record ID: %d\n", record->id);
(gdb) x/64x record->data
0x667690:       0x6c654820      0x6f576f6c      0x21646c72      0xabababab
0x6676a0:       0xabababab      0xabababab      0xabababab      0xfeeefeee
0x6676b0:       0x00000000      0x00000000      0x00000000      0x00000000
0x6676c0:       0xfeeefeee      0xfeeefeee      0x06b27f62      0x3400a16a
0x6676d0:       0x41414120      0x41414141      0x0a414141      0xabababab
0x6676e0:       0xabababab      0xabababab      0xabababab      0xfeeefeee
0x6676f0:       0x00000000      0x00000000      0x00000000      0x00000000
0x667700:       0xfeeefeee      0xfeeefeee      0x06b27f62      0x3800a16a
0x667710:       0x41414120      0x41414141      0xabababab      0xabababab
0x667720:       0xabababab      0xabababab      0xfeeefeee      0xfeeefeee
0x667730:       0x00000000      0x00000000      0x00000000      0x00000000
0x667740:       0xfeeefeee      0xfeeefeee      0x05b17f62      0x0000a16a
0x667750:       0x0066b4b0      0x00000000      0x00667480      0x00000000
0x667760:       0xfeeefeee      0xfeeefeee      0xfeeefeee      0xfeeefeee
0x667770:       0xfeeefeee      0xfeeefeee      0xfeeefeee      0xfeeefeee
0x667780:       0x00000000      0x00000000      0x0eb27f6a      0x3800a16a
(gdb)
```

Figure 21 – Expected output without overflow



```
Thread 1 hit Breakpoint 2, displayRecord (record=0x5ffdf8) at .\2.c:68
68          printf("Record ID: %d\n", record->id);
(gdb) x/64x record->data
0xaa6040:       0x6c654820      0x6f576f6c      0x21646c72      0x3130310a
0xaa6050:       0x2032310a      0x41414141      0x41414141      0x310a4141
0xaa6060:       0x380a3230      0x41414120      0x41414141      0x00000000
0xaa6070:       0x00000000      0x00000000      0x00000000      0x00000000
0xaa6080:       0x00000000      0x00000000      0x00000000      0x00000000
0xaa6090:       0x00000000      0x00000000      0x00000000      0x00000000
0xaa60a0:       0x00000000      0x00000000      0x00000000      0x00000000
0xaa60b0:       0x00000000      0x00000000      0x00000000      0x00000000
0xaa60c0:       0x00000000      0x00000000      0x00000000      0x00000000
0xaa60d0:       0x00000000      0x00000000      0x00000000      0x00000000
0xaa60e0:       0x00000000      0x00000000      0x00000000      0x00000000
0xaa60f0:       0x00000000      0x00000000      0x00000000      0x00000000
0xaa6100:       0x00000000      0x00000000      0x00000000      0x00000000
0xaa6110:       0x00000000      0x00000000      0x00000000      0x00000000
0xaa6120:       0x00000000      0x00000000      0x00000000      0x00000000
0xaa6130:       0x00000000      0x00000000      0x00000000      0x00000000
(gdb)
```

Figure 22 – Output with overflow

Delving further into this by placing a breakpoint at the displayRecord function, we can take a look futher into memory, shown in Figure 21 - Figure 22, and for this the input file was altered, Figure 20, so the overflow would be more clear.

Within the expected output, it shows surrounding activity around the records whereas comparing this to the output with the overflow, the difference is clear. In the overflow output, the A's from records 2 and 3 can be seen at the beginning of memory, 0xaa6050 – 0xaa6060, whereas after this memory is empty.

Vulnerability Analysis of Vulnerable Code

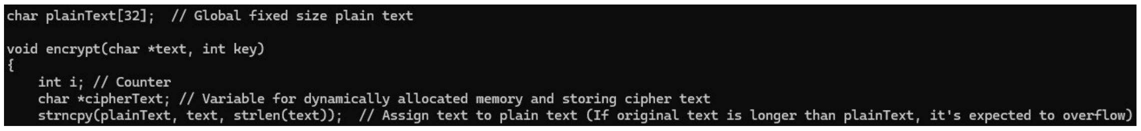
This section will run through the vulnerable code (C Code) creation using a pre-existing symmetric key algorithm, where vulnerabilities will be implemented only during the encryption phase of the code, tested and exploited.

Symmetric Algorithm & Chosen vulnerabilities

Beginning with the symmetric algorithm, a simple Caesar cypher was chosen for a number of reasons. Firstly, the type of key used in the cipher would be an integer and due to how the cypher is alphabet based, limited to a maximum of 26. Secondly, the process involved with encryption allows for the implementation of both a heap overflow and Buffer Overflow followed by a format string vulnerability post encryption. And finally keeping in line with vulnerabilities within the encryption code, with how this specific encryption code is built, a second unused function was created overall rendering the encryption useless.

Analysis of code

Buffer Overflow

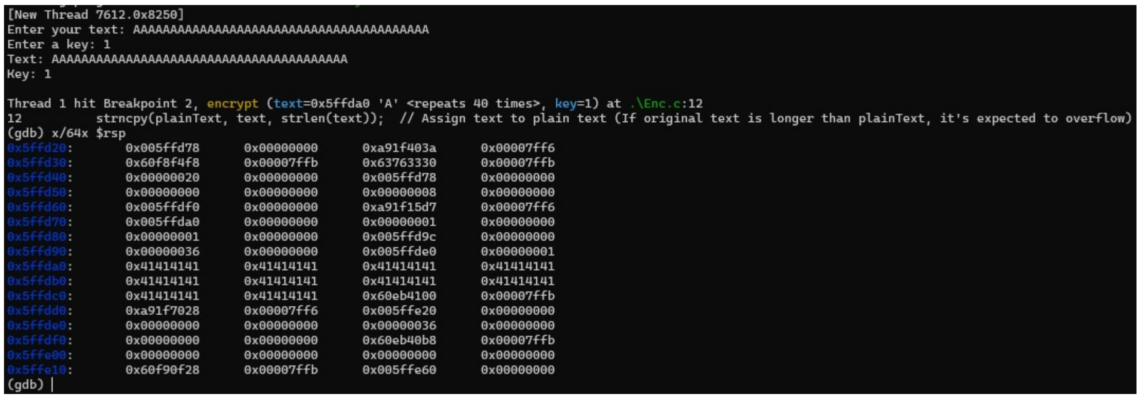


```
char plainText[32];  // Global fixed size plain text

void encrypt(char *text, int key)
{
    int i; // Counter
    char *cipherText; // Variable for dynamically allocated memory and storing cipher text
    strncpy(plainText, text, strlen(text));  // Assign text to plain text (If original text is longer than plainText, it's expected to overflow)
```

Figure 23 – user input copied into 32 byte plaintext variable

Beginning with the Buffer Overflow vulnerability a global variable was created named plainText, show in Figure 23, with a size of 32 bytes. This is done so it can easily be accessed from anywhere in the code with the size chosen for demonstration purposes.

plainText is then used within the encrypt function as part of strncpy() where the original text entered by the user is copied into the plaintext variably at the length of the original text entered during user input indicated by strlen(text). Should text exceed the plainText limit of 32, this is expected to produce a Buffer Overflow.

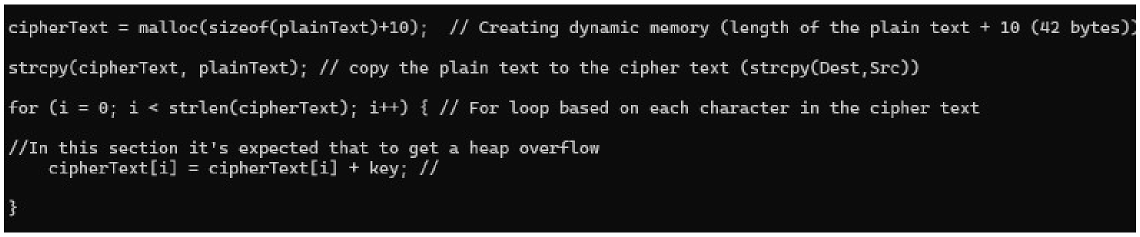


```
[New Thread 7612.0x8250]
Enter your text: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Enter a key: 1
Text: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Key: 1

Thread 1 hit Breakpoint 2, encrypt (text=0x5ffda0 'A' <repeats 40 times>, key=1) at .\Enc.c:12
12          strncpy(plainText, text, strlen(text));  // Assign text to plain text (If original text is longer than plainText, it's expected to overflow)
(gdb) x/64x $rsp
0x5ffd20:       0x005ffd78      0x00000000      0xa91f403a      0x00007ff6
0x5ffd30:       0x60f8f4f8      0x00007ffb      0x63763330      0x00007ffb
0x5ffd40:       0x00000020      0x00000000      0x005ffd78      0x00000000
0x5ffd50:       0x00000000      0x00000000      0x00000008      0x00000000
0x5ffd60:       0x005ffdf0      0x00000000      0xa91f15d7      0x00007ff6
0x5ffd70:       0x005ffda0      0x00000000      0x00000001      0x00000000
0x5ffd80:       0x00000001      0x00000000      0x005ffd9c      0x00000000
0x5ffd90:       0x00000036      0x00000000      0x005ffde0      0x00000001
0x5ffda0:       0x41414141      0x41414141      0x41414141      0x41414141
0x5ffdb0:       0x41414141      0x41414141      0x41414141      0x41414141
0x5ffdc0:       0x41414141      0x41414141      0x60eb4100      0x00007ffb
0x5ffdd0:       0xa91f7028      0x00007ff6      0x005ffe20      0x00000000
0x5ffde0:       0x00000000      0x00000000      0x00000036      0x00000000
0x5ffdf0:       0x00000000      0x00000000      0x60eb40b8      0x00007ffb
0x5ffe00:       0x00000000      0x00000000      0x00000000      0x00000000
0x5ffe10:       0x60f90f28      0x00007ffb      0x005ffe60      0x00000000
(gdb) |
```

Figure 24 – Overflow shown in code

The issue with this begins with the limited size of plaintext. User input can be much greater than 32 bytes and therefor opens an easily exploitable Buffer Overflow path as shown in Figure 24. The strncpy function is used specifically for this purpose as it allows for a fixed size allocation for implementation of the Buffer Overflow.

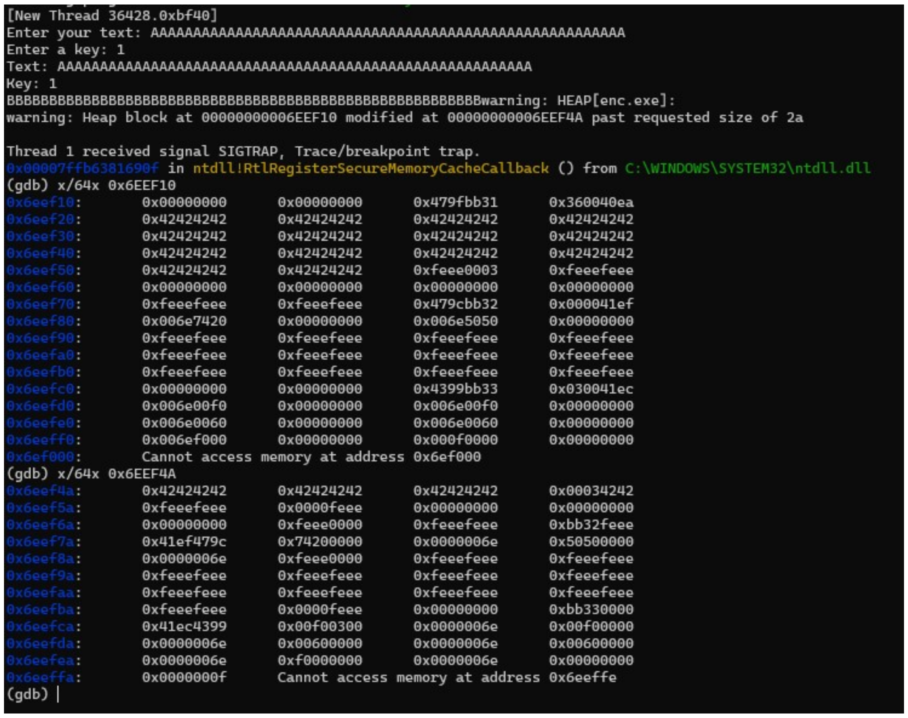
Heap Overflow



```
cipherText = malloc(sizeof(plainText)+10);  // Creating dynamic memory (length of the plain text + 10 (42 bytes))
strcpy(cipherText, plainText); // copy the plain text to the cipher text (strcpy(Dest,Src))
for (i = 0; i < strlen(cipherText); i++) { // For loop based on each character in the cipher text
//In this section it's expected that to get a heap overflow
    cipherText[i] = cipherText[i] + key; //
}
```

Figure 25 – Memory allocation assigned

Adding onto the previous overflow constructs, variable cipherText was created and assigned memory allocation at the length of our plainText variable plus ten (42 bytes), plaintext is then copied into our ciphertext variable using strcpy Figure 25.

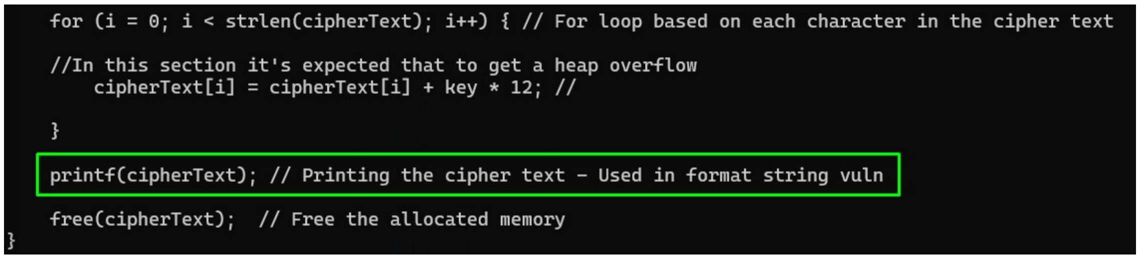


```
[New Thread 36428.0xbf40]
Enter your text: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Enter a key: 1
Text: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Key: 1
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBwarning: HEAP[enc.exe]:
warning: Heap block at 00000000006EEF10 modified at 00000000006EEF4A past requested size of 2a

Thread 1 received signal SIGTRAP, Trace/breakpoint trap.
0x00007ffb6381690f in ntdll!RtlRegisterSecureMemoryCacheCallback () from C:\WINDOWS\SYSTEM32\ntdll.dll
(gdb) x/64x 0x6EEF10
0x6eef10:       0x00000000      0x00000000      0x479fbb31      0x360040ea
0x6eef20:       0x42424242      0x42424242      0x42424242      0x42424242
0x6eef30:       0x42424242      0x42424242      0x42424242      0x42424242
0x6eef40:       0x42424242      0x42424242      0x42424242      0x42424242
0x6eef50:       0x42424242      0x42424242      0xfeee0003      0xfeeefeee
0x6eef60:       0x00000000      0x00000000      0x00000000      0x00000000
0x6eef70:       0xfeeefeee      0xfeeefeee      0x479cbb32      0x000041ef
0x6eef80:       0x006e7420      0x00000000      0x006e5050      0x00000000
0x6eef90:       0xfeeefeee      0xfeeefeee      0xfeeefeee      0xfeeefeee
0x6eefa0:       0xfeeefeee      0xfeeefeee      0xfeeefeee      0xfeeefeee
0x6eefb0:       0xfeeefeee      0xfeeefeee      0xfeeefeee      0xfeeefeee
0x6eefc0:       0x00000000      0x00000000      0x4399bb33      0x030041ec
0x6eefd0:       0x006e00f0      0x00000000      0x006e00f0      0x00000000
0x6eefe0:       0x006e0060      0x00000000      0x006e0060      0x00000000
0x6eeff0:       0x006ef000      0x00000000      0x000f0000      0x00000000
0x6ef000:       Cannot access memory at address 0x6ef000
(gdb) x/64x 0x6EEF4A
0x6eef4a:       0x42424242      0x42424242      0x42424242      0x00034242
0x6eef5a:       0xfeeefeee      0x0000feee      0x00000000      0x00000000
0x6eef6a:       0x00000000      0xfeee0000      0xfeeefeee      0xbb32feee
0x6eef7a:       0x41ef479c      0x74200000      0x0000006e      0x50500000
0x6eef8a:       0x0000006e      0xfeee0000      0xfeeefeee      0xfeeefeee
0x6eef9a:       0xfeeefeee      0xfeeefeee      0xfeeefeee      0xfeeefeee
0x6eefaa:       0xfeeefeee      0xfeeefeee      0xfeeefeee      0xfeeefeee
0x6eefba:       0xfeeefeee      0x0000feee      0x00000000      0xbb330000
0x6eefca:       0x41ec4399      0x00f00300      0x0000006e      0x00f00000
0x6eefda:       0x0000006e      0x00600000      0x0000006e      0x00600000
0x6eefea:       0x0000006e      0xf0000000      0x0000006e      0x00000000
0x6eeffa:       0x0000000f      Cannot access memory at address 0x6eeffe
(gdb) |
```

Figure 26 – Memory overflow displayed at 0x6EEF4A

After allocation of memory size and copying plaintext the encryption process begins and as seen in Figure 26 this involves iterating through characters in cipher text at the length of user input, causing the heap overflow. As user input is larger than cipherText it forces writing to space where memory was not allocated for, which can also be seen again by the display of 0x42424242 when looking further into memory.

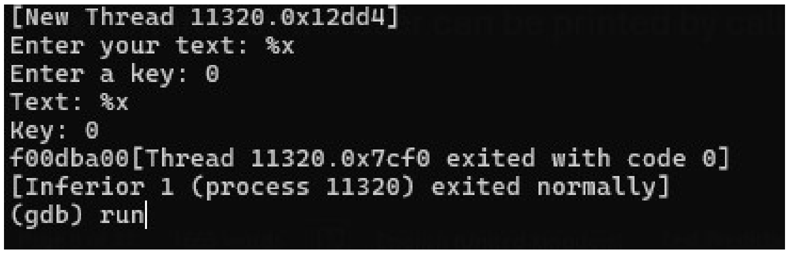
Format String



```
    for (i = 0; i < strlen(cipherText); i++) { // For loop based on each character in the cipher text

    //In this section it's expected that to get a heap overflow
        cipherText[i] = cipherText[i] + key * 12; //

    }

    printf(cipherText); // Printing the cipher text - Used in format string vuln

    free(cipherText);  // Free the allocated memory
}
```

Figure 27 – printf function with no format specification

Moving onto our format string vulnerability next, the section of code for this is included within the encrypt function and is prior to freeing the allocated memory, Figure 27. The printf function used here doesn't include any function such as %s which would otherwise help towards protect it against this type of attack.

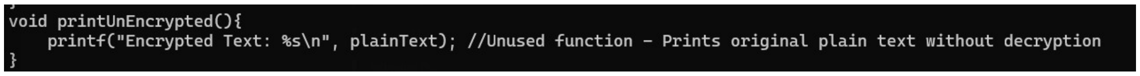


```
[New Thread 11320.0x12dd4]
Enter your text: %x
Enter a key: 0
Text: %x
Key: 0
f00dba00[Thread 11320.0x7cf0 exited with code 0]
[Inferior 1 (process 11320) exited normally]
(gdb) run
```

Figure 28 – Text input with no key used for format string

As Figure 28 displays, by entering a format string as the plaintext to be encrypted and using the key 0 so the text doesn't change, this function can then be manipulated into printing memory addresses using format strings.

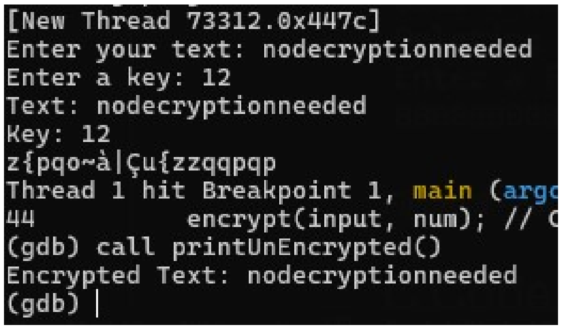
Accessible Unencrypted text



```
void printUnEncrypted(){
    printf("Encrypted Text: %s\n", plainText); //Unused function - Prints original plain text without decryption
}
```

Figure 29 - printUnEncrypted() function

Finally, included is a function separate to both encrypt and main functions. The intention of this is to print out the original unencrypted text that was entered by the user without the use of decryption and simulate leftover code for example from testing shown in Figure 29.



```
[New Thread 73312.0x447c]
Enter your text: nodecryptionneeded
Enter a key: 12
Text: nodecryptionneeded
Key: 12
z{pqo~à|Çu{zzqqpqp
Thread 1 hit Breakpoint 1, main (argc
44          encrypt(input, num); // C
(gdb) call printUnEncrypted()
Encrypted Text: nodecryptionneeded
(gdb) |
```

Figure 30 - calling printUnEncrypted()

Inserting a breakpoint at the end of main after the encrypt function has been called ensures the plainText variable has been populated. Once this is populated and as shown in Figure 30, this can be the printUnEncrypted() function can be called from within GNU Debugger to reveal the original text entered by the user without the need for decryption.

Conclusion

Identified throughout are multiple vulnerabilities across the two provided files and the symmetric algorithm requested. After finishing analysis of the first file the following vulnerabilities were found: Directly accessible information, format string, heap overflow, buffer overflow and Incorrect deletion of user. Heap overflow was the only vulnerability found in file2, however, during analysis there seemed to be the possibility for an integer overflow or a buffer overflow in the code. These vulnerabilities were found at various points in the code in each of the files and could be easily avoided by using more suitable functions or for example specifying %s in the printf function to help prevent a format string vulnerability.

Moving onto the custom c code section, all vulnerabilities have been implemented outside of the main user inputs as required with a total of four vulnerabilities implemented: Heap overflow, buffer overflow, format string and directly accessible plain text.

Appendix

Figure 1 - 1.c code with highlighted vulnerabilities 5
Figure 2 - Running file1.exe in GDB 5
Figure 3 – Unused function in code 6
Figure 4 – Calling function in GDB 6
Figure 5 – Format string in code 6
Figure 6 – Testing for format string vulnerability 6
Figure 7 – Reading from memory using format string 7
Figure 8 – Delete user function in code 7
Figure 9 – User 0 permanently selected in delete user function 7
Figure 10 – Truncated old username with new username 8
Figure 11 – input which exceeds character limit 9
Figure 12 – Memory overflow seen in memory 9
Figure 13 – Edit user function in code 10
Figure 14 – Input overflow seen causing error 10
Figure 15 – for loop for loading records 11
Figure 16 – Edited input file 11
Figure 17 – Output showing ability to read next line in file 11
Figure 18 – Edited input with new record size 11
Figure 19 – Output displaying all records 12
Figure 20 – Altered input file 12
Figure 21 – Expected output without overflow 12
Figure 22 – Output with overflow 13
Figure 23 – user input copied into 32 byte plaintext variable 14
Figure 24 – Overflow shown in code 14
Figure 25 – Memory allocation assigned 15
Figure 26 – Memory overflow displayed at 0x6EEF4A 15
Figure 27 – printf function with no format specification 16
Figure 28 – Text input with no key used for format string 16
Figure 29 - printUnEncrypted() function 16
Figure 30 - calling printUnEncrypted() 16

C Code

```
#include <stdio.h>

#include <string.h>


char plainText[32];  // Global fixed size plain text


void encrypt(char *text, int key)

{

   int i; // Counter
```

```
    char *cipherText; // Variable for dynamically allocated memory and storing cipher text

    strncpy(plainText, text, strlen(text));  // Assign text to plain text (If original text is longer than
plainText, it's expected to overflow)


    cipherText = malloc(sizeof(plainText));  // Creating dynamic memory (length of the plain text + 1 (33
bytes))


    strcpy(cipherText, plainText); // copy the plain text to the cipher text (strcpy(Dest,Src))


    for (i = 0; i < strlen(cipherText); i++) { // For loop based on each character in the cipher text


    //In this section it's expected that to get a heap overflow

        cipherText[i] = cipherText[i] + key; //


    }


    printf(cipherText); // Printing the cipher text - Used in format string vuln


    free(cipherText);  // Free the allocated memory

}

void printUnEncrypted(){

    printf("Encrypted Text: %s\n", plainText); //Unused function - Prints original plain text without
decryption

}

int main(int argc, char **argv)

{

    char input[256]; // create input variable (Higher length than plainText)

    int num; // create num variable (for use with ceaser cipher key)


    printf("Enter your text: "); //Asks user for the text to encrypt

    scanf("%s", input); // Read the input
    printf("Enter a key: "); // Asks user for the key
    scanf("%d", &num); // Read the key
    printf("Text: %s\nKey: %d\n", input, num); // Printing the text and key on the same line

    encrypt(input, num); // Calling the encrypt function passing the text and key

    return 0; // Exiting the program
}
```