Vulnerability Discovery And Exploitation

Author: Jean-Michel Batista

Supervisor: Dr Nick Savage

Table of Contents

1. Analysis: Program 1

    1.1. Static code analysis

    1.2. Dynamic code analysis

2. Analysis: Program 2

    1.1. Static code analysis

    1.2. Dynamic code analysis

3. Analysis: Program 3

    1.1. Assembly code analysis

    1.2. Dynamic binary analysis

4. Conclusion

5. References

Analysis: Program 1

Part 1: Static code analysis

Performing a static code analysis will help us address weaknesses in the source code that could potentially lead to vulnerabilities. In order to examine the code, we will use Atom. Upon opening the file.c source code and reviewing it, we can see that the program contains several errors that are exploitable.

We shall attempt to highlight these errors and discuss why these weaknesses make the program vulnerable.

1.
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

static char array[20];
char* fileOpen(char *filename, int select);
int read(char *argv);
```

The global buffer created by the static char array[20] will be stored in the DATA segment of the memory (.BSS Data Segment in this case). Here the array is given a limited size of 20 characters. There is no protection against the program copying data beyond the end of the array and corrupting memory outside its reserved space. If the program copies an input past the end of the array, it will be able to overwrite adjacent memory addresses. This error could be critical, depending on whether or not it is possible to overflow the buffer[1].

2.

```
int read(char *argv)
{
int word;
printf("Which number word do you want selected? ");
scanf("%d",&word);
printf("word is %d\n",word);
strcpy(array,argv);
return word;
}
```

Here the first problem is that scanf will store whatever the user's input is. It will then use the format string %d to take that input and store it as an int value in the "word" variable. There is no limit for huge numbers nor a check for the input to see if it is a number at all. We have the possibility of an integer overflow or wrap-around error as the user input stored in "word" could very well exceed the maximum value for an integer[2]. This weakness will generally lead to undefined behavior, but if a wrap-around error results in other conditions such as a buffer overflow, the weakness could be used to execute arbitrary code.

The second problem we run into is that bytes from argv are then copied into the array by strcpy without checking whether or not the array has the space to accommodate these bytes (or checking to see if argv is NULL-terminated). The big problem with this coding error is that the command line parameter could be longer than the static array, potentially leading to a global buffer overflow. An attacker could use this weakness as an attack vector to achieve arbitrary code execution by overwriting sensitive data such as function pointers in global memory or overwriting objects on the heap[3].

3.

```
char* fileOpen(char *filename, int select)
{
int x;
FILE *f0;
char *fileScan;
fileScan=malloc(30);
if ((f0=fopen(filename,"r"))==NULL) {printf("Error\n");exit(0);}

for (x=0;x<select;x++)
  fscanf(f0,"%s",fileScan);
fscanf(f0,"%s",fileScan);
fclose(f0);
return fileScan;
}
```

Here the first problem is that fileScan=malloc(30); is a significant error. This piece of code will allocate a limited amount of memory in heap space at run time (possibly to prevent a stack-based buffer overflow). The reason why this is a significant error is that just like we can have stack-based buffer overflows, the memory the program reserves in heap space can also be overflowed. It is relevant to mention that when comparing a heap-based buffer overflow to a stack-based buffer overflow, an attacker would have to use other means of exploitation, such as overwriting a heap-stored function pointer or overwriting a heap-allocated object's virtual function pointer[4] as no return addresses are stored on the heap.

The second problem is fscanf(f0,"%s",fileScan); is used with no field width limit. Having no field width limit means that the function fscanf() will be responsible for crashing the program in a huge input data scenario. The program will attempt to store the chosen "number word" the user has selected from the input.txt inside fileScan=malloc(30); even if that word happened to have thousands of characters, this could lead to a heap-based buffer overflow which can result in arbitrary code being executed.

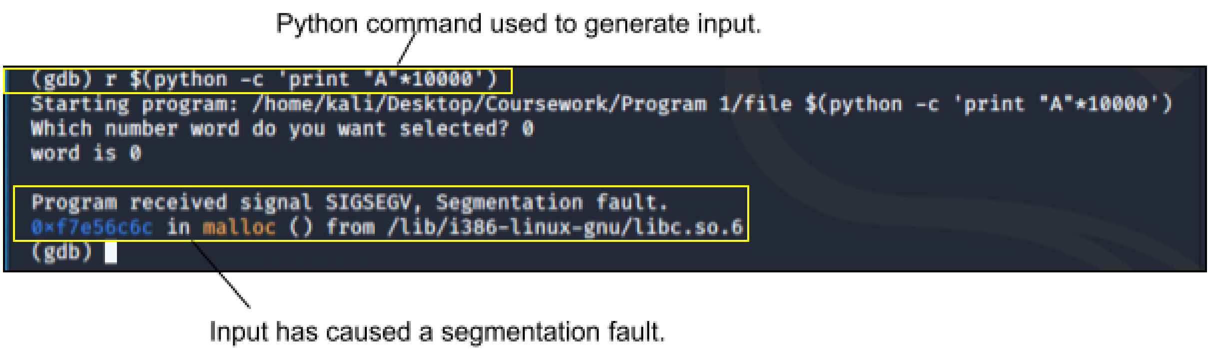
Part 2: Dynamic code analysis

Performing a dynamic code analysis will help us address the weaknesses found in the source code. We shall attempt to demonstrate these weaknesses and discuss the vulnerabilities found in our results. First, we compile the program. Then, we load it up inside the GNU debugger's environment.

1.GLOBAL BUFFER OVERFLOW(CRITICAL)

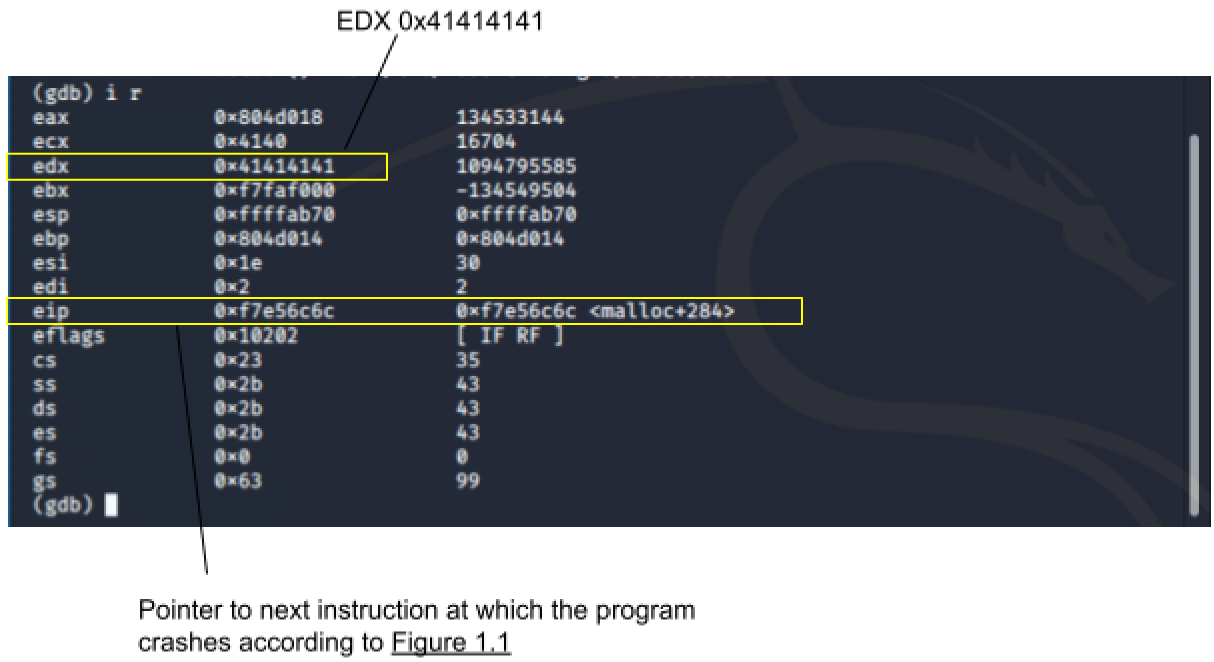
Demonstrating strcpy(array,argv); can overflow static char array[20];'s buffer.

Figure 1.1



As seen in Figure 1.1, we have caused a segmentation fault by giving a command line parameter bigger than static char array[20];'s buffer. Segmentation faults occur when we are overwriting memory we are not allowed to overwrite. The notice "in malloc ()" tells us we most likely have standard memory corruption through a buffer-overflow.

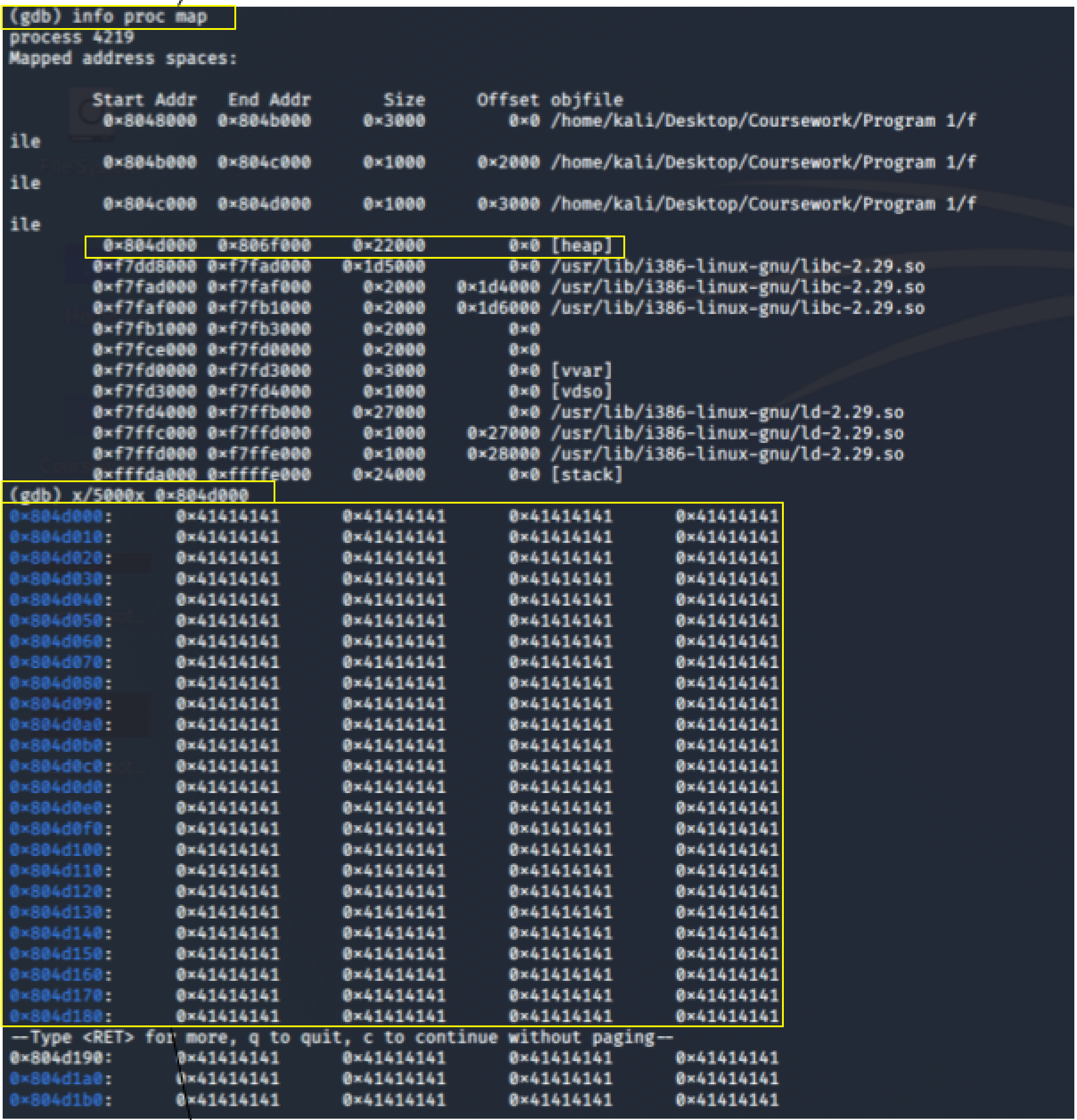
Figure 1.2



As seen in Figure 1.2 the EDX register will show the values 0x41414141. It is not possible to overwrite data registers with a buffer overflow as first, they are in the CPU, and second, do not have addresses. It is the values overwritten in RAM that will eventually be loaded into registers.

Figure 1.3

Report on the address ranges accessible in the program.



```
(gdb) info proc map
process 4219
Mapped address spaces:

        Start Addr   End Addr       Size     Offset objfile
         0x8048000  0x804b000     0x3000        0x0 /home/kali/Desktop/Coursework/Program 1/f
ile
         0x804b000  0x804c000     0x1000     0x2000 /home/kali/Desktop/Coursework/Program 1/f
ile
         0x804c000  0x804d000     0x1000     0x3000 /home/kali/Desktop/Coursework/Program 1/f
ile
         0x804d000  0x806f000    0x22000        0x0 [heap]
        0xf7dd8000 0xf7fad000   0x1d5000        0x0 /usr/lib/i386-linux-gnu/libc-2.29.so
        0xf7fad000 0xf7faf000     0x2000   0x1d4000 /usr/lib/i386-linux-gnu/libc-2.29.so
        0xf7faf000 0xf7fb1000     0x2000   0x1d6000 /usr/lib/i386-linux-gnu/libc-2.29.so
        0xf7fb1000 0xf7fb3000     0x2000        0x0
        0xf7fce000 0xf7fd0000     0x2000        0x0
        0xf7fd0000 0xf7fd3000     0x3000        0x0 [vvar]
        0xf7fd3000 0xf7fd4000     0x1000        0x0 [vdso]
        0xf7fd4000 0xf7ffb000    0x27000        0x0 /usr/lib/i386-linux-gnu/ld-2.29.so
        0xf7ffc000 0xf7ffd000     0x1000    0x27000 /usr/lib/i386-linux-gnu/ld-2.29.so
        0xf7ffd000 0xf7ffe000     0x1000    0x28000 /usr/lib/i386-linux-gnu/ld-2.29.so
        0xfffda000 0xffffe000    0x24000        0x0 [stack]
(gdb) x/5000x 0x804d000
0x804d000:      0x41414141      0x41414141      0x41414141      0x41414141
0x804d010:      0x41414141      0x41414141      0x41414141      0x41414141
0x804d020:      0x41414141      0x41414141      0x41414141      0x41414141
0x804d030:      0x41414141      0x41414141      0x41414141      0x41414141
0x804d040:      0x41414141      0x41414141      0x41414141      0x41414141
0x804d050:      0x41414141      0x41414141      0x41414141      0x41414141
0x804d060:      0x41414141      0x41414141      0x41414141      0x41414141
0x804d070:      0x41414141      0x41414141      0x41414141      0x41414141
0x804d080:      0x41414141      0x41414141      0x41414141      0x41414141
0x804d090:      0x41414141      0x41414141      0x41414141      0x41414141
0x804d0a0:      0x41414141      0x41414141      0x41414141      0x41414141
0x804d0b0:      0x41414141      0x41414141      0x41414141      0x41414141
0x804d0c0:      0x41414141      0x41414141      0x41414141      0x41414141
0x804d0d0:      0x41414141      0x41414141      0x41414141      0x41414141
0x804d0e0:      0x41414141      0x41414141      0x41414141      0x41414141
0x804d0f0:      0x41414141      0x41414141      0x41414141      0x41414141
0x804d100:      0x41414141      0x41414141      0x41414141      0x41414141
0x804d110:      0x41414141      0x41414141      0x41414141      0x41414141
0x804d120:      0x41414141      0x41414141      0x41414141      0x41414141
0x804d130:      0x41414141      0x41414141      0x41414141      0x41414141
0x804d140:      0x41414141      0x41414141      0x41414141      0x41414141
0x804d150:      0x41414141      0x41414141      0x41414141      0x41414141
0x804d160:      0x41414141      0x41414141      0x41414141      0x41414141
0x804d170:      0x41414141      0x41414141      0x41414141      0x41414141
0x804d180:      0x41414141      0x41414141      0x41414141      0x41414141
--Type <RET> for more, q to quit, c to continue without paging--
0x804d190:      0x41414141      0x41414141      0x41414141      0x41414141
0x804d1a0:      0x41414141      0x41414141      0x41414141      0x41414141
0x804d1b0:      0x41414141      0x41414141      0x41414141      0x41414141
```

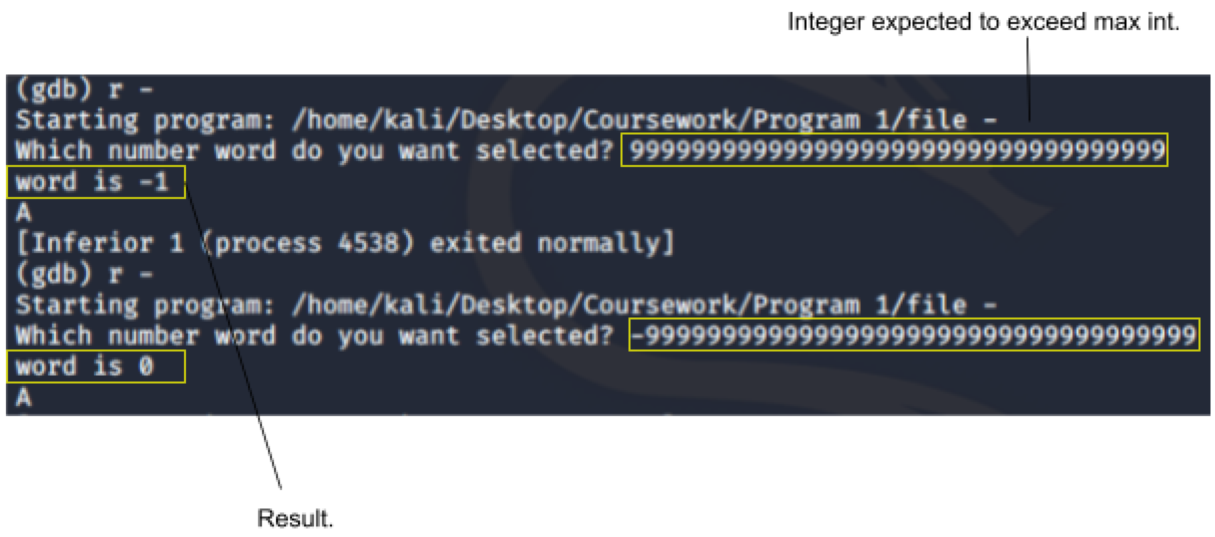
Memory addresses in which our input has overwritten bytes on the heap.

To follow up on the first notice of "in malloc ()" in Figure 1.1 we inspect the start address of the heap. We can see in Figure 1.3 corrupted memory addresses where our command-line argument has overwritten all bytes. This confirms it will be possible to overwrite objects on the heap through the caused global buffer overflow.

2.INTEGER OVERFLOW OR WRAP-AROUND ERROR

Demonstrating scanf("%d",&word); can exceed its maximum integer value.

Figure 1.4



Here in Figure 1.4, we can see that the integer the program stores in the word variable reaches its max integer value, represented by -1 or 0 in 64bit. In this case, the program will only store the number of bits that can be stored, the remaining bits that can’t be stored will be lost.

<u>3. HEAP BUFFER OVERFLOW(CRITICAL)</u>

Demonstrating fscanf(f0,"%s",fileScan); can overflow fileScan=malloc(30);'s buffer.

In our input, we will have a string composed of 700 "C" characters. We will use this 700 characters long string as input and try to store it inside fileScan=malloc(30);.

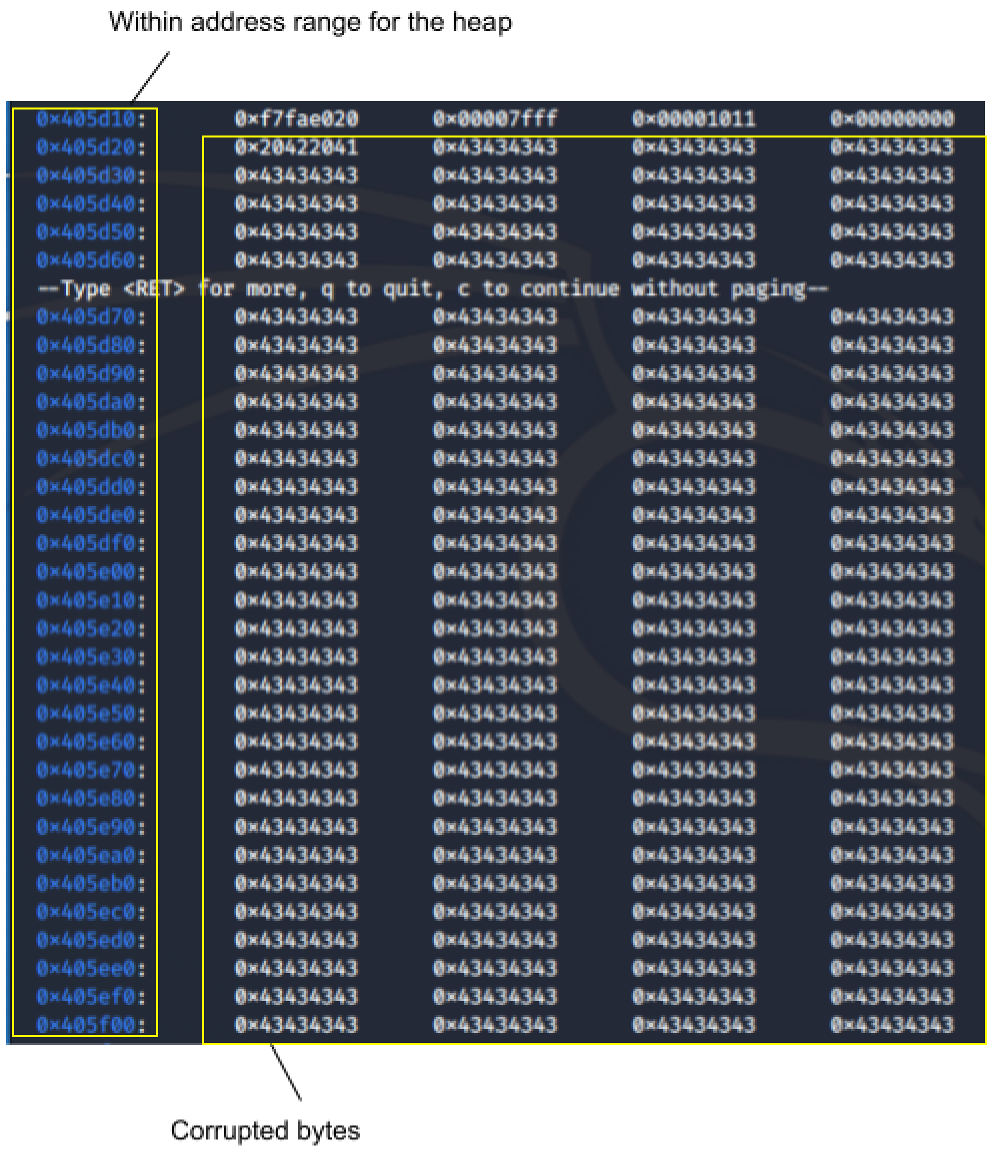
<u>Figure 1.5</u>

As we can see in Figure 1.5, the program crashes with a segmentation fault error caused by a heap-based buffer overflow.

Upon inspecting heap memory addresses, we can find the character C overwritten past the size of **fileScan=malloc(30);'s** buffer as shown by Figure 1.6 and Figure 1.7. This confirms the vulnerability could be used as an attack vector for arbitrary code execution by overwriting a heap-stored function pointer.

Figure 1.8

Figure 1.9


Within address range for the heap
0x405d10: 0xf7fae020 0x00007fff 0x00001011 0x00000000
0x405d20: 0x20422041 0x43434343 0x43434343 0x43434343
0x405d30: 0x43434343 0x43434343 0x43434343 0x43434343
0x405d40: 0x43434343 0x43434343 0x43434343 0x43434343
0x405d50: 0x43434343 0x43434343 0x43434343 0x43434343
0x405d60: 0x43434343 0x43434343 0x43434343 0x43434343
--Type <RET> for more, q to quit, c to continue without paging--
0x405d70: 0x43434343 0x43434343 0x43434343 0x43434343
0x405d80: 0x43434343 0x43434343 0x43434343 0x43434343
0x405d90: 0x43434343 0x43434343 0x43434343 0x43434343
0x405da0: 0x43434343 0x43434343 0x43434343 0x43434343
0x405db0: 0x43434343 0x43434343 0x43434343 0x43434343
0x405dc0: 0x43434343 0x43434343 0x43434343 0x43434343
0x405dd0: 0x43434343 0x43434343 0x43434343 0x43434343
0x405de0: 0x43434343 0x43434343 0x43434343 0x43434343
0x405df0: 0x43434343 0x43434343 0x43434343 0x43434343
0x405e00: 0x43434343 0x43434343 0x43434343 0x43434343
0x405e10: 0x43434343 0x43434343 0x43434343 0x43434343
0x405e20: 0x43434343 0x43434343 0x43434343 0x43434343
0x405e30: 0x43434343 0x43434343 0x43434343 0x43434343
0x405e40: 0x43434343 0x43434343 0x43434343 0x43434343
0x405e50: 0x43434343 0x43434343 0x43434343 0x43434343
0x405e60: 0x43434343 0x43434343 0x43434343 0x43434343
0x405e70: 0x43434343 0x43434343 0x43434343 0x43434343
0x405e80: 0x43434343 0x43434343 0x43434343 0x43434343
0x405e90: 0x43434343 0x43434343 0x43434343 0x43434343
0x405ea0: 0x43434343 0x43434343 0x43434343 0x43434343
0x405eb0: 0x43434343 0x43434343 0x43434343 0x43434343
0x405ec0: 0x43434343 0x43434343 0x43434343 0x43434343
0x405ed0: 0x43434343 0x43434343 0x43434343 0x43434343
0x405ee0: 0x43434343 0x43434343 0x43434343 0x43434343
0x405ef0: 0x43434343 0x43434343 0x43434343 0x43434343
0x405f00: 0x43434343 0x43434343 0x43434343 0x43434343
Corrupted bytes

Analysis: Program 2

Part 1: Static code analysis

Just like with program 1, we are performing a static code analysis that will help us address weaknesses in the source code that could potentially lead to vulnerabilities. Following the same methodology, we open up 2.c with Atom. Upon opening the 2.c source code and reviewing it, we can see that the program contains several errors that are exploitable.

We shall attempt to highlight these errors and discuss why these weaknesses make the program vulnerable.

We will break down the readfile function in two parts for better readability.

1.

```
int readfile(char *argv)
{
FILE *inFile;
char c;
int x;
int filesize=0;
int ChunkID;
int ChunkSize;
char Format[5];
int bitsPerSample;
char *file;
char ID[5],tmp;
if ((inFile=fopen(argv,"r"))==NULL) {printf("Error\n");exit(0);}

while((c=fgetc(inFile))!=EOF)
    if (c!='\n')
          filesize++;

printf("File size = %d\n",filesize);

file=malloc(filesize*sizeof(char));

rewind(inFile)
```

Here the first evident problems we run across are both char Format[5]; and char ID[5],tmp;. These pieces of code will only allocate enough space for five characters, each into their destination buffer on the stack. By this point, we understand the programs will read different inputs from an input file, the order and format string for each input is read will be determined by fscanf() function calls later on in the program. If fscanf() copies an input bigger than five characters inside the variables Format and ID, we can have bytes that will overwrite memory addresses outside of the intended bounds. These errors could be critical, depending on whether or not it is possible to overflow the buffers. If an attacker managed to control the extended instruction pointer (EIP) or overwrite a stack-stored function pointer, these weaknesses could be used to execute arbitrary code.

2.

```
fscanf(inFile,"%d",&ChunkID);
fscanf(inFile,"%d",&ChunkSize);
fscanf(inFile,"%s",Format);
fscanf(inFile,"%d",&bitsPerSample);
fscanf(inFile,"%c",&tmp);
for (x=0;x<filesize-10;x++)
  fscanf(inFile,"%c",&file[x]);
fscanf(inFile,"%s",ID);
fclose(inFile);

printf("Chunk ID = %d\n",ChunkID);
printf("Chunk Size = %d\n",ChunkSize);
printf("Format = %s\n",Format);
printf("Bits per sample = %d\n",bitsPerSample);
printf("ID = %s\n",ID);

processing(file,filesize);

}
```

Here the problem relies upon the way multiple function calls of fscanf() will store inputs from an input file in memory. Both fscanf(inFile,"%s",Format); and fscanf(inFile,"%s",ID); have no field width limit, meaning that the function fscanf() will be responsible for crashing the program in a huge input data scenario. The program will attempt to store the corresponding inputs for char Format[5]; and char ID[5],tmp; even if those inputs happened to have thousands of characters. For both cases, these could lead to stack-based buffer overflows.

In addition, fscanf(inFile,"%d",&ChunkID);, fscanf(inFile,"%d",&ChunkSize); and fscanf(inFile,"%d",&bitsPerSample); will store whatever the user's input is, it will then use the format string %d to take that input and store it as an int value in their corresponding variables. There is no limit for huge numbers nor a check to see if the input is a number at all. We have the possibility of three integer overflows or wrap-around errors as the user input stored in these variables could very well exceed the maximum value for an integer. These weaknesses will generally lead to undefined behavior, but if a wrap-around error results in other conditions such as a buffer overflow, the weakness could be used to execute arbitrary code.

3.

```
int processing(char *input, int filesize)
{
char *output;
output=malloc(filesize*sizeof(char));
strcpy(output,input);
printf("File info = %s",output);
}
```

Here the bytes from input are copied into output by strcpy without checking whether or not the memory stored in the heap has the space to accommodate these bytes. Nonetheless, we can say that the use of =malloc(filesize*sizeof(char)); throughout the program attempts to safeguard the program as a countermeasure to an otherwise statically declared value for malloc().

Part 2: Dynamic code analysis

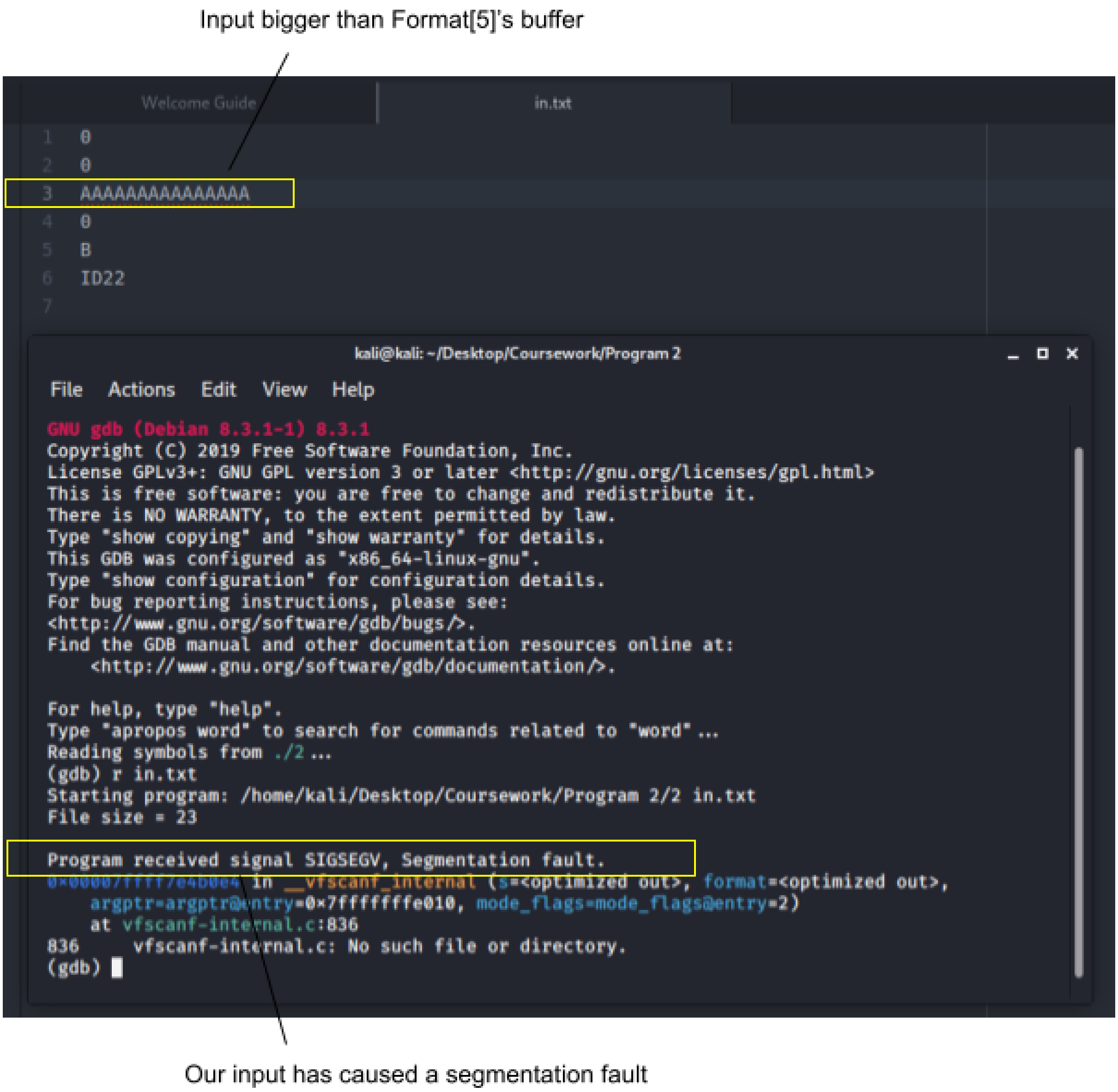
Performing a dynamic code analysis will help us address the weaknesses found in the source code. We shall attempt to demonstrate these weaknesses and discuss the vulnerabilities found in our results. First, we compile the program. Then, we load it up inside the GNU debugger's environment.

1. STACK BUFFER OVERFLOW(CRITICAL)

Demonstrating fscanf(inFile,"%s",Format); can overflow char Format[5];'s buffer.

In our input file, we will have a string composed of "A" characters at the position meant for Format. We will cause a big input scenario where our input stored will be larger than char Format[5];.

Figure 2.1



As we can see in Figure 2.1, the program crashes with a segmentation fault error caused by a stack-based buffer overflow. We confirm an attacker could manage to control the extended instruction pointer (EIP) or overwrite a stack-stored function pointer by overwriting illegal memory locations.

2. STACK BUFFER OVERFLOW

Demonstrating fscanf(inFile,"%s",ID); can overflow char ID[5],tmp;'s buffer.

In our input file, we will have a string composed of "B" characters at the position meant for . char ID[5],tmp;. We will cause a big input scenario where our input stored will be larger than char ID[5],tmp;.

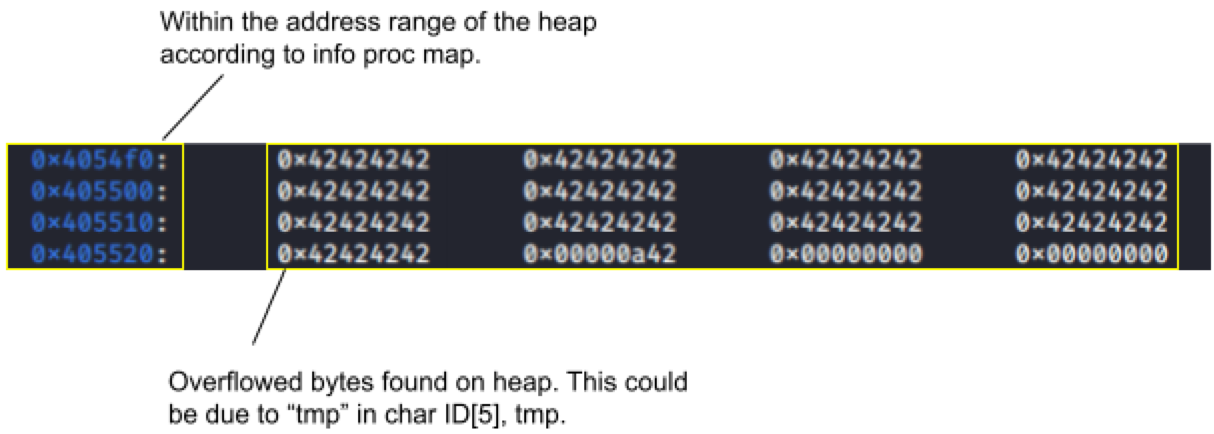
Figure 2.2

As we can see in Figure 2.4, the program will not crash no matter the size of the input. The first five characters are stored in char ID[5],tmp;, we can also see a sixth character as we overflow the buffer.

Figure 2.3

In Figure 2.3, by further inspecting the address in which the variable ID resides, we can see our five A characters plus a sixth character that is corrupting memory on the stack.

Figure 2.4

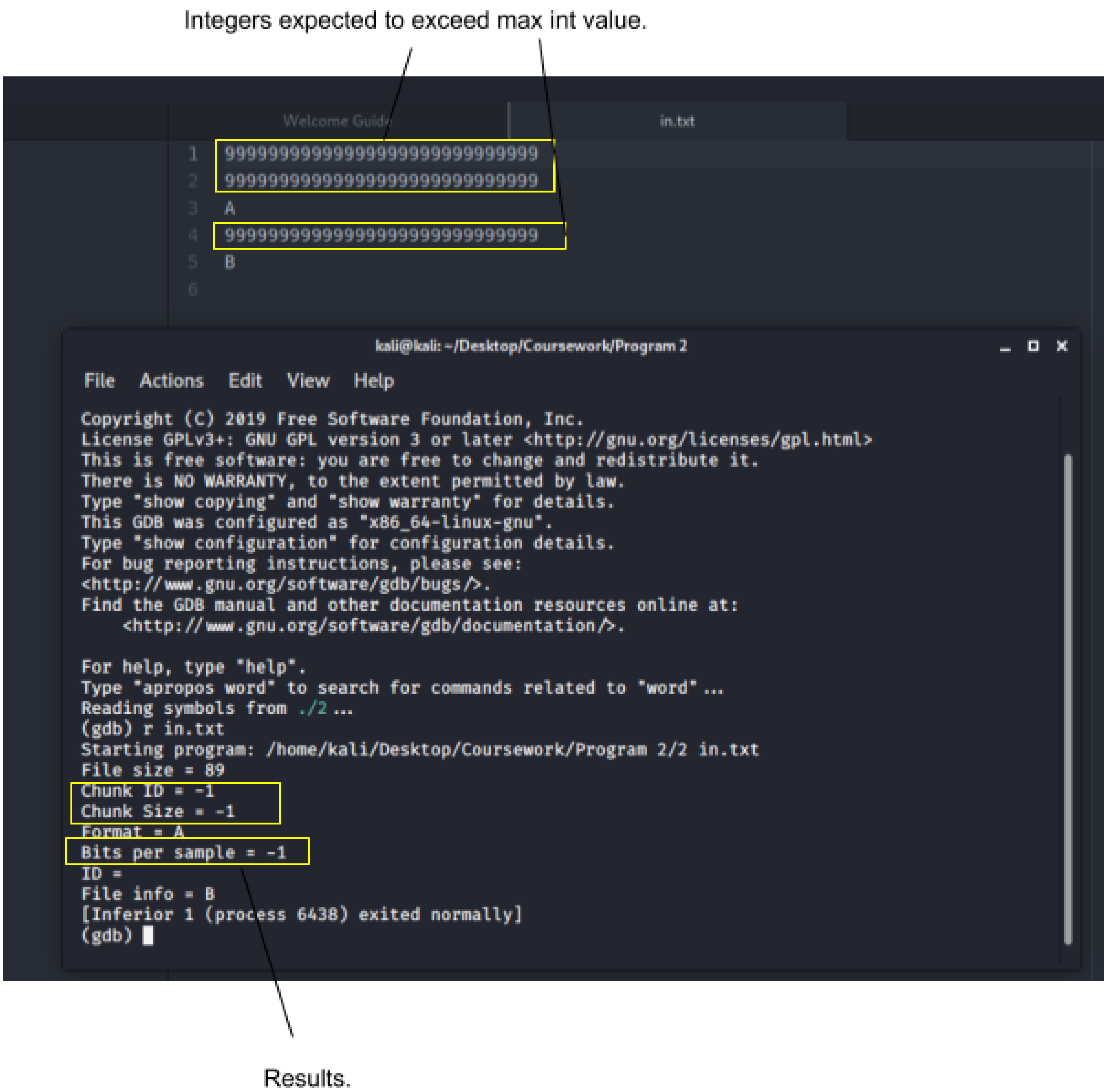


We can see in Figure 2.4 that any character past the first six will be printed by File info, which prints characters straight out of heap space. In this program the heap space will be based depending on the file size. Even if we overflow char ID[5],tmp;'s buffer we would not be able to overwrite any heap stored function pointer as heap space is allocated based on file size.

3. INTEGER OVERFLOW OR WRAP-AROUND ERROR

Demonstrating fscanf(inFile,"%d",&ChunkID);, fscanf(inFile,"%d",&ChunkSize); and fscanf(inFile,"%d",&bitsPerSample); can exceed its maximum integer value.

Figure 2.5



Here in Figure 2.5, we can see that the integers the program stores in the int variables reach their max integer values, represented by -1 or 0 in 64bit. In this case, the program will only store the number of bits that can be stored, the remaining bits that can't be stored will be lost.

Analysis: Program 3

Part 1: Assembly code analysis

We will not be able to examine the source code of the program as we did with program one and program two. However, what we can do is analyze the assembly code with the help of GDB. After loading up program 3 in GDB, we can disassemble the main() function using the following command:

> disas main

This will show us the assembly code of the main() function of the program, allowing us to get a better understanding of the program flow. If needed, we will also inspect the assembly code for subsequent subroutines. We shall attempt to highlight the essential things we need to take into consideration and discuss how they will help us identify vulnerabilities in the program.

1.
```
<+51>:    callq  0x8a0 <socket@plt>
```

Here the call to socket() is being made to create a socket. A socket will consist of three things, an IP address, a transport protocol, and a port number. From this, we can expect that the program will communicate over the network.

2.
```
<+397>:   cmp    $0x54,%al
```

Here we have an essential piece of assembly code that will be crucial during our dynamic analysis. The assembly code tells us that the data the program receives is being compared (cmp) to 0x54, which is the ASCII code for "T".

3. <+399>: jne 0xc09 <main+477>

Right after the instruction to compare the data the program receives to the character "T" we have a JNE (Jump if not equal) instruction to <main+477>. If the compared value is not equal to "T" we will jump to somewhere else in the program. By following this path in the program flow we run into calls to sleep() (<+487>: callq 0x890 <sleep@plt>) and close() (<+505>: callq 0x830 <close@plt>) which means we're most likely causing the program to terminate.

4. <+430>: callq 0x7e0 <recv@plt>

If we follow the different program flow where JNE is equal to "T" we have a new call to recv(), possibly meaning that the program will be able to receive more data from the connected socket if the first character is "T"

.

5. <+472>: callq 0x9ca <cracked>

By following the program flow in which JNE is equal to "T" further down the instructions, we run into the cracked() function. To understand what this function does, we must further inspect its content. To do so, we do the following:

> disas cracked

Upon disassembling the cracked function, we can see there is a potential weakness in the program. Within the cracked function there is a function call to strcpy() (<+53>: callq 0x7f0 <strcpy@plt>). If the program is storing the data being sent over the network into a buffer using strcpy, there is a chance strcpy is storing that data without checking whether or not the space allocated in memory has the space to accommodate these bytes (or checking to see if argv is NULL-terminated). The big problem with this coding error is that the command line parameter could be longer than the memory allocated, potentially leading to a buffer overflow, which can result in arbitrary code being executed.

Within the cracked function <+4>: sub $0x120,%rsp will allocate space for our buffer. If we convert this to decimal we get the number 288 (Not the exact size of the buffer but close as the function could be reserving space for other things). Theoretically if the buffer can be overflowed, 300 characters should be enough to overflow the buffer and 1100 characters should be enough to crash the program (<+4>: sub $0x430,%rsp at the very start of the assembly of the main function (1072 in decimal)).

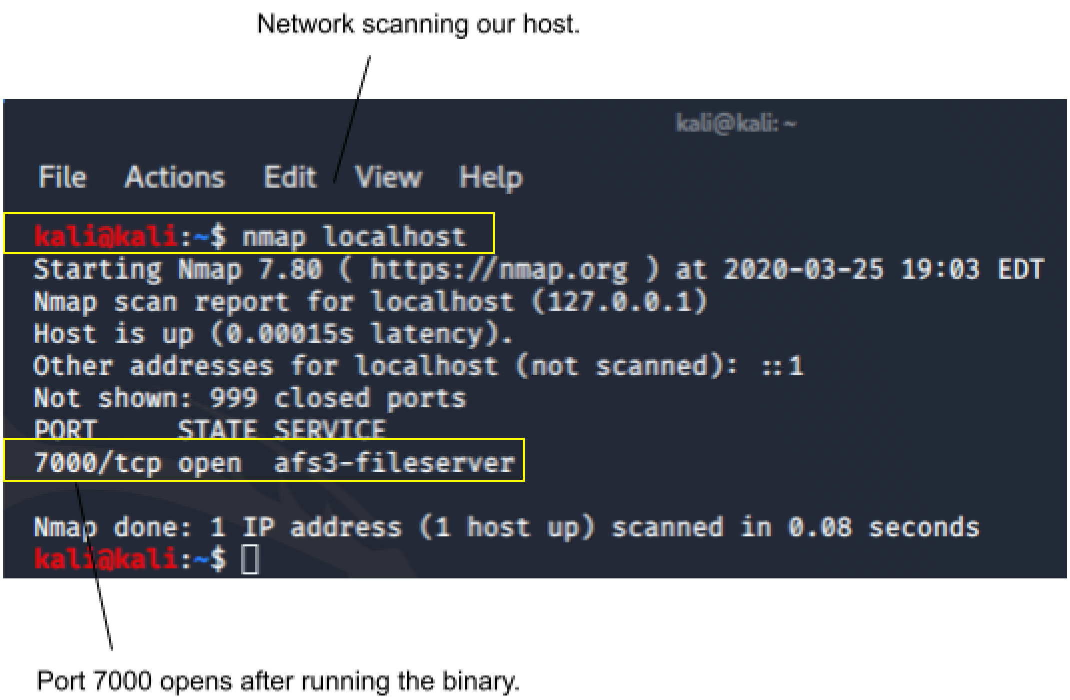
6.
   <+530>: callq 0x800 <__stack_chk_fail@plt>

Back in the main function our last function call will be made to <__stack_chk_fail@plt>. This is a clear indicator that the program will try to protect itself against stack-based buffer overflows. The function will be added by the compiler and act as a canary to achieve stack protection. If we can overflow the buffer we should expect stack smashing detection and not a segmentation fault.

Part 2: Dynamic binary analysis

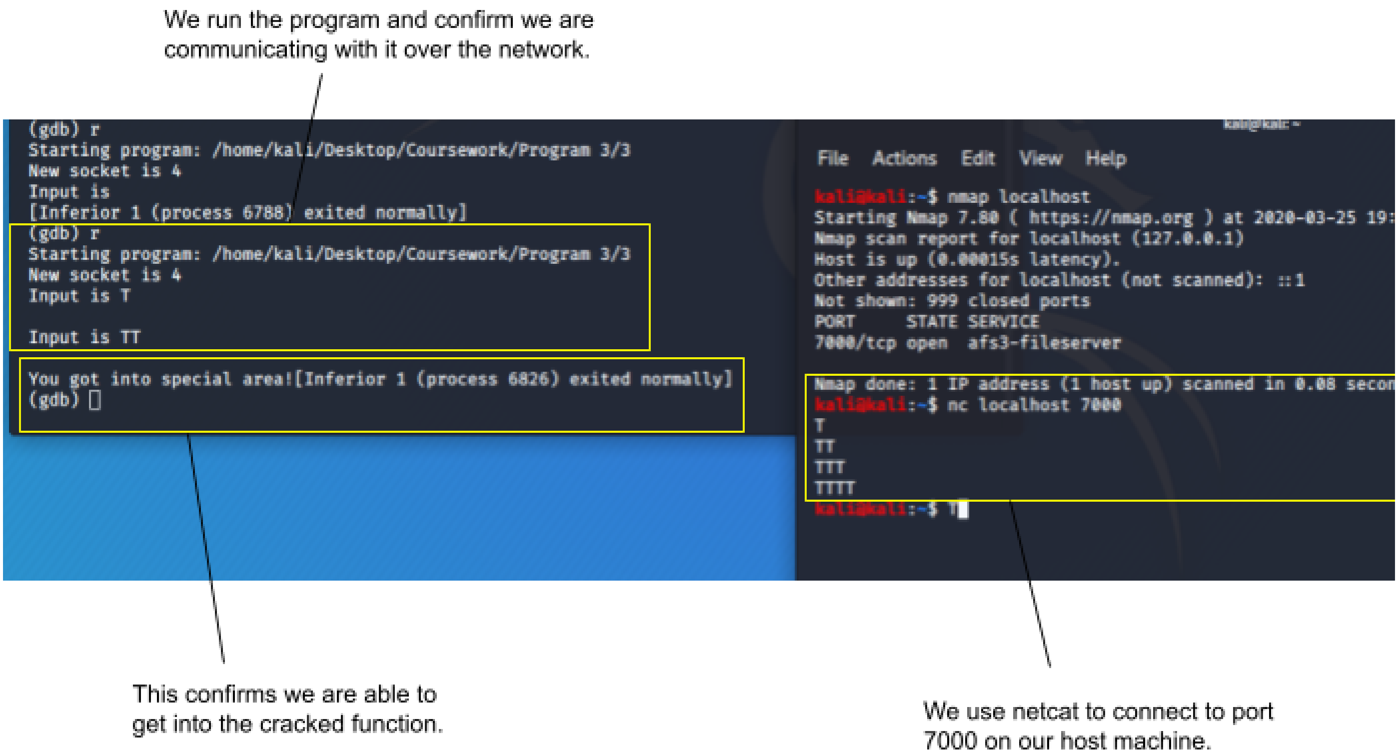
Performing a dynamic analysis of the binary will help us address the weakness found in the assembly code. We shall attempt to demonstrate this weakness and discuss the vulnerability found in our result. We load the binary up inside the GNU debugger's environment and execute the binary.

Figure 3.1



In order to confirm the binary allows communication over the network, as seen in Figure 3.1, we perform a nmap scan on our host computer.

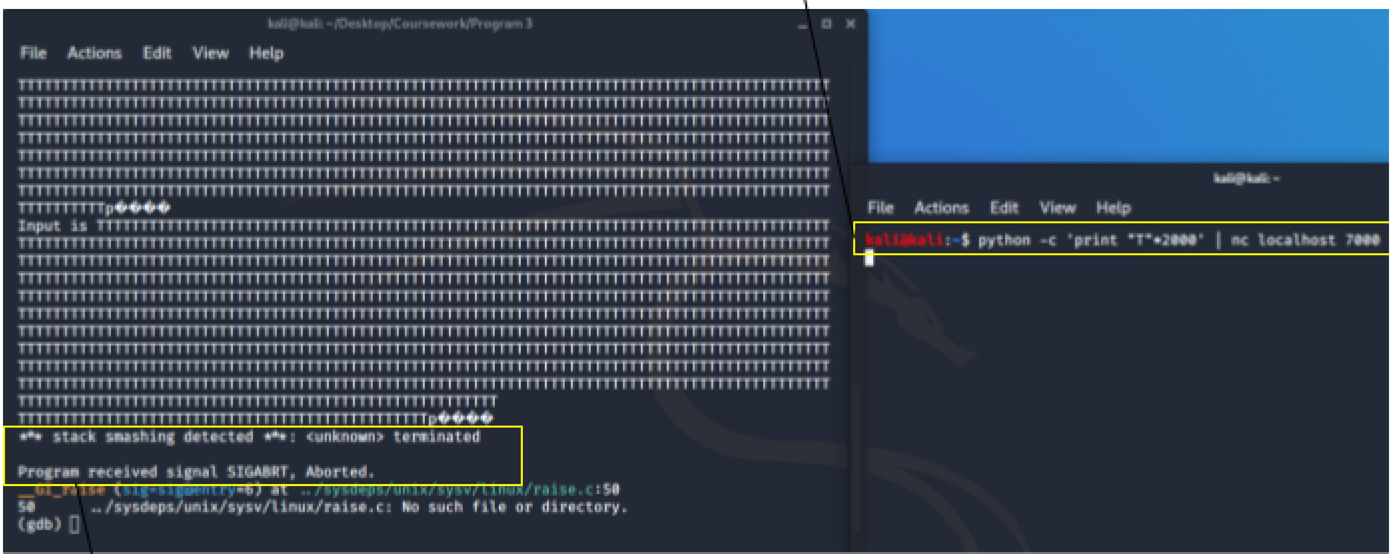
Figure 3.2



As seen in Figure 3.2, we connect to port 7000 using Netcat to send data over the network. Upon communicating multiple "T"'s, we can get into the special area of the program.

Figure 3.3

Python is used to generate our input, we pipe the output of (python -c ‘print “T”*2000’) into the program.

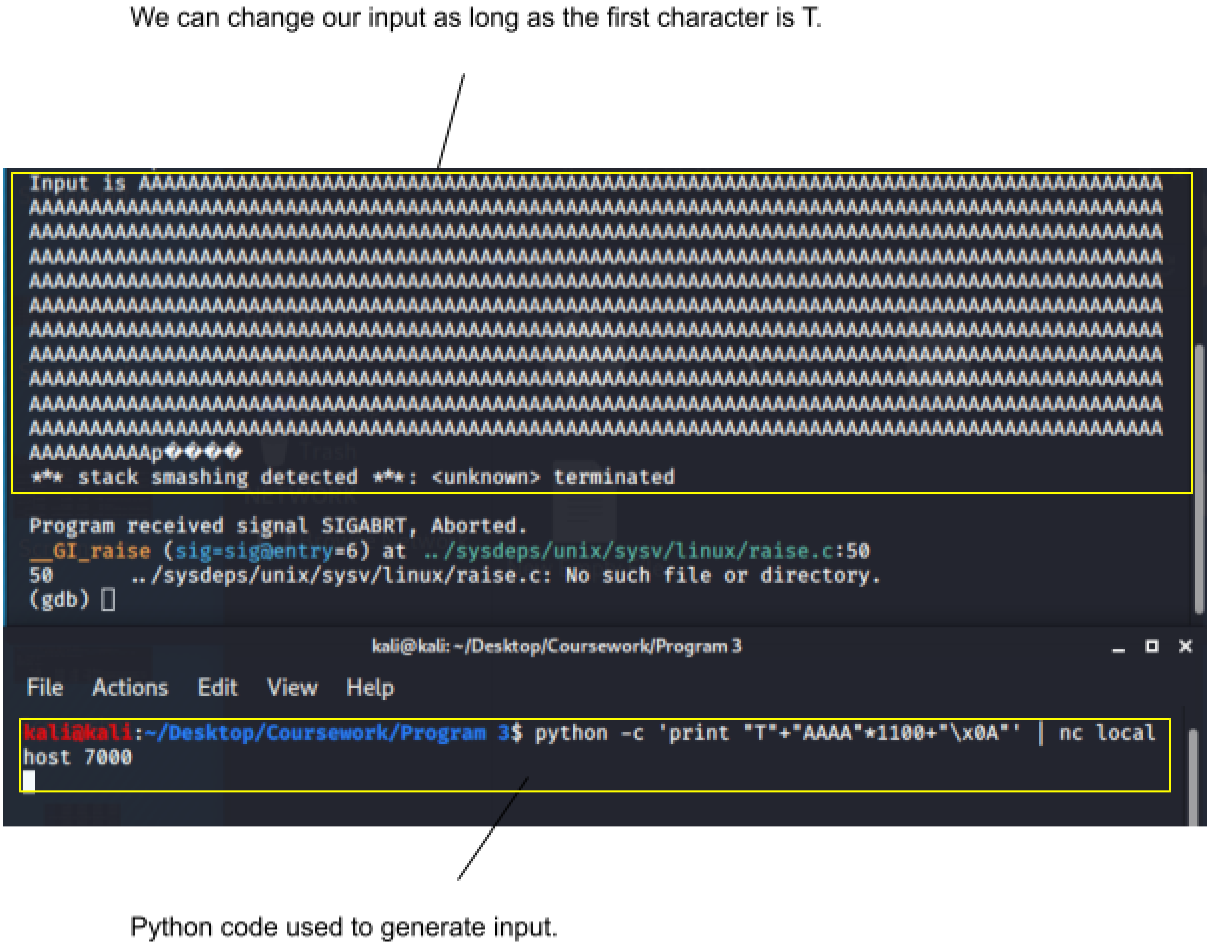


Stack smashing detected. The program sends a signal to abort.

To check if strcpy is storing data without checking whether or not the space allocated in memory has the space to accommodate these bytes, as seen in figure Figure 3.3, we use a python command to generate a large amount of character and pipe that output to our Netcat command. The program receives a signal to abort as stack smashing is detected. The stack canary (<__stack_chk_fail@plt>) comes into place and generates the stack smashing detected error in response to its defense mechanism against stack buffer overflows.

Stacking smashing detection confirms again that this is a stack-based buffer overflow and that the overflow will attempt to overwrite anything before it in the stack frame[4]. The warning could be bypassed by different means of exploitation. As long as the first byte is equal to “T”, any additional input will also be accepted. This can be seen in Figure 3.4 and Figure 3.5. Here overflowing the buffer could only be an attack vector for arbitrary code execution if an attacker managed to bypass the compiler’s stack smashing detection.

Figure 3.4



Conclusion: Most significant vulnerabilities

The most significant vulnerabilities in each program can be pinpointed to multiple instances of buffer overflows.

GLOBAL BUFFER OVERFLOW(CRITICAL)
HEAP BUFFER OVERFLOW(CRITICAL)

In program one, we have demonstrated both the buffers allocated in the .BSS data segment and heap could be potential attack vectors to achieve arbitrary code execution by an attacker.

STACK BUFFER OVERFLOW(CRITICAL)

In program two, the most significant vulnerability is the possibility to overflow char Format[5];'s buffer, which is present on the stack; this buffer could also be a potential attack vector to achieve arbitrary code execution by an attacker.

STACK BUFFER OVERFLOW(CRITICAL)

In program three, the most significant vulnerability is the possibility to overflow the stack-allocated buffer in which strcpy copies the user input into, this buffer could also be a potential attack vector to achieve arbitrary code execution by an attacker if the attacker manages to bypass stack smashing detection.

References:

[1] OWASP. (2020). Buffer Overflow. Retrieved from:
https://owasp.org/www-community/vulnerabilities/Buffer_Overflow

[2] CWE. (2019). CWE-190: Integer Overflow or Wraparound. Retrieved from:
https://cwe.mitre.org/data/definitions/190.html

[3] A. Smotrakov. (2020). Global Buffer Overflow. Retrieved from:
https://blog.gypsyengineer.com/en/security/global-buffer-overflows.html

[4] Y. Younan, W. Joosen, F. Piessens, (2004). Code Injection in C and C++: A Survey of Vulnerabilities and Countermeasures. Retrieved from:
http://www.cs.kuleuven.ac.be/publicaties/rapporten/cw/CW386.pdf