Introduction

A two-stage assignment, we began by dividing our resources between the two tasks, with each member of our pair focusing on one of the tasks to complete. By splitting our efforts in two, we found that although better overall progress was made, it made it difficult to accurately judge the exact stage we were at on the tasks.

We also made use of GitLab with a private repo dedicated solely for this assignment, to keep track of what each of us had done, with the accompanying commits becoming an effective way to identify bugs or crashes we created along the way. Discord was also heavily used to share screenshots of our issues as well as communicate with each other over the holiday break.

Task 1: Create an IA32 assembly program that includes a pseudo random number generator

C#

Writing out the first draft in C# proved to be extremely useful when it came to the later C code. C# generally being an easier language to write and understand than C or IA32 Assembly, the code proved useful in building a mental image on what we would need to implement on Ubuntu.

Creating an C# version also helped us in identifying what C libraries and functions would likely be needed. As a result, it was effective in providing a scope of commands to research (scanf instead of ReadLine() or srand instead of Random() for example).

Figure 1: C# code, our first step

```
namespace VulnStudent
{
    class Program
    {
        static void Main(string[] args)
        {
            //Take name
            Console.WriteLine("What is your name?");
            String getName = Console.ReadLine();

            //Start Generator
            Random rng = new Random();

            //Generate Numbers
            int score1 = rng.Next(1, 100);
            int score2 = rng.Next(1, 100);

            //Calc Average
            Console.WriteLine(getName + ", your score is ");
            Console.Write((score1 + score2) / 2);
            Console.ReadLine();
        }
    }
}
```

C

Although not the primary focus of the assignment, the two biggest challenges we faced when creating the C program were how to obtain the user input and how to generate the two random numbers. Creating and testing this program was beneficial for us, as it showed us that a buffer overflow was an exploit we could work into our assembly code.

Our first attempt involved passing the argv parameter (user input string) into a strcpy, so the input can be larger than the 10 bytes of space allocated to name. We also added a warning

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
int Score(char *input)
{
    //Variables
    srand(time(NULL));
    int average;
    int score1;
    int score2;
    char name[10];

    //Copy the user input in name
    strcpy(name, input);
    printf("Hello %s \n", &name);

    //Get random scores
    printf("Your two scores are \n");
    score1 = rand() % 101;
    score2 = rand() % 101;
    printf("%d and %d \n", score1, score2);

    //Calculate average
    average = (score1 + score2) / 2;
    printf("Your average score is %d \n", average);
    return 0;
}

int main(int argc, char *argv[])
{
    //Check there is a name input
    if(argc < 2){
        printf("Usage: <yourname>\n");
        exit(0);
    }
    else{
        Score(argv[1]);
```

Figure 2: C code using argv[] for user input

message if you do not pass a name parameter to the program. Regarding vulnerabilities, it should be possible to buffer overflow the name variable since the buffer is declared higher up on the stack to the score variables (score2 will flow into score1 and then to the average variable). Although we couldn't find a way to overflow into the score variables and progress further down the stack, we knew it was possible to do this due to the proof of concept on strcpy vulnerabilities in lectures and the segmentation faults we were receiving. A more vulnerable user input function was required that doesn't check how much data it is reading in; this function was gets().

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
int main(int argc, char **argv)
{
    //Variables
    srand(time(NULL));
    int average = 0;
    int score1 = 0;
    int score2 = 0;
    char name[10];

    //Get random scores
    score1 = rand() % 101;
    score2 = rand() % 101;

    //Calculate average
    average = (score1 + score2) / 2;

    //Copy user input into name
    gets(name);

    //Print
    printf("Your average score is %d \n", average);
    return 0;
```

Figure 3: C code using gets() for user input

This was partly due to our existing knowledge of the command and because of the statement in figure 6 taken from the gets Linux man page, confirming its vulnerability to buffer overflows.

After further reading on common C vulnerabilities we were able to realise how buffer overflows were a commonplace and easily created anomaly. With this knowledge, we set out to create a vulnerable c program to test how difficult it would be to cause a buffer overflow, in a similar layout as would be present in our final x86 version.

```c
tabs  1 test.c
 1  int main(void){
 1  int average = 0;
 2  char buffer[4];
 3
--  4  average = getRand();
--  5  printf("Enter name : \n");
--  6  scanf("%s", buffer);
 7  printf("Congratulations %s ! \n", buffer);
 8  printf("You scored %d \n", average);
 9  return 0;
10  }
11
12  int getRand(){
13      int rand1 = 10;
14      int rand2 = 30;
15      return (rand1 + rand2)/2;
16  }
```

Figure 4: C code used to test buffer overflow

The random number generation was removed in the testing c code in order to simplify the application and to help with debugging, as we knew with a fixed value we would notice if the score was changed, rather than there being an error with the generation of the number.

Additionally, the buffer size chosen for the testing c code was selected to be extremely small, only 4 bytes allocated, so much less would have to be written to cause an overflow, this helped to further speed up testing.

As seen in figure 3, we use srand to generate a pseudo random number from a distinctive runtime value (usually implementing the time() function). This is our seed value for the rand() operation. The two numbers are created this way and then they're modularly divided by 101 to cut them down to the 0 – 100 range. The average is then found and output with a closing message. Although this program was functional, we could have added a welcome message parameter asking for a name instead of a blank response for easier usability. However, we couldn't find an exploit for this which we suspected was due to an absence of stack manipulation. This resulted in our exploits overflowing into system critical instructions rather than the score2 variable, causing a

```
[d9@debian Downloads]$ vi test.c
[d9@debian Downloads]$ ./a
bash: ./a: No such file or directory
[d9@debian Downloads]$ ./a.out
Enter name :
Tom
Congratulations Tom !
You scored 20
[d9@debian Downloads]$ ./a.out
Enter name :
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Congratulations AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA !
You scored 1094795585
Segmentation fault
```

Figure 5: Exploiting the compiled C code by overflowing the 4 byte buffer and writing over the average variable.

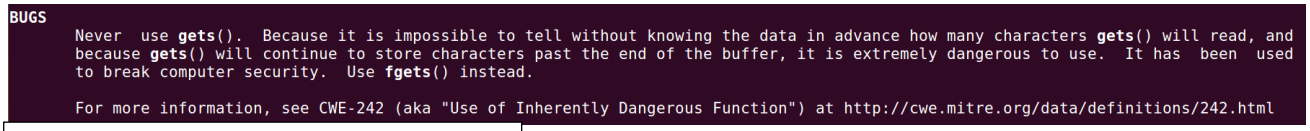

Figure 6: gets() man page bug notice

segmentation fault and system crash. More precise instructions on where to store each command was needed which brings us along to our assembly implementation.

IA32 Assembly

Making use of our understanding from the C exploitation, we sought to implement a similar buffer overflow vulnerability in the IA32 code. Initially this proved a much more difficult challenge, as when writing in an assembly language you need to fully understand what is happening at each step in the code. We found that keeping track of the stack and the order the instructions were placed on it particularly difficult in attempting to create an overflow. Furthermore, we also realised that we would have to use some esp and ebp manipulation in order to align the pointer to cause an overflow. Again, random number generation proved to be a challenge as div will not execute a division if the edx register contains data, a feature we were not aware of at first.

After many defective iterations using rdtsc and call time to create a solution, we finally decided on srand and created a working version of the program. This resulted in our IA32 code developing through numerous iterations, even after initially completing the aims, as we discovered that the original code was not structured in a way that was conducive to exploitation.

Initially going into assembly, we looked back over the sec204 lecture slides, using the exercises to regain that familiarity with writing assembly. The book Hacking: The Art of Exploitation proved extremely useful when figuring out where we should start designing exploits and what the tell-tale signs were that we were breaking the program. We then began implementing our random number solution we created earlier in C into assembly. Creating a function for this allowed us to call it whenever we needed a new random number generated and it would output this to the edx.

In some of the initial versions of the x86 code we were hesitant to use make full use of the stack, primarily due to its intimidating nature. This resulted in us opting to store the randomly generated numbers in the ebp, which despite not being it's intended purpose, worked fine as if we weren't using to point to the base of the stack. There was nothing preventing us from treating it like any other register.

This implementation had some key flaws however, while it was able to complete all the requirements of the task, it was very difficult to exploit in any meaningful way.

While initially hesitant to the idea, eventually we decided re-writing the entire application to make correct usage of the stack. The aim of this was to improve the programs readability

and vulnerability to a format or buffer exploit. Gaining this experience of how the stack operates proved essential in working the buffer overflow exploit.

This proved to be a much better idea, as not only did it greatly improve our understanding of the stack, it also allowed us to know where we had to reach to overwrite them. We could now calculate exactly much input would be required to overflow into the location of the average score.

```
pushl   %eax
call    srand           #requires a pushed value to seed properly
pushl   %ebp            #pushes base pointer
movl    %esp, %ebp      #sets stack pointer to base
subl    $32, %esp       #frees up 32 bytes to put stuff in
movl    $0, -4(%ebp)    #empties where the random number is going
movl    $0, %eax        #clears eax for getRand to write to it
call    getRand
movl    %edx, -4(%ebp)  #stores the random number on the stack
movl    $0, -8(%ebp)    #clears  where the 2nd random number is going
movl    $0, %eax        #clears eax for getRand to write to it
call    getRand
movl    %edx, -8(%ebp)  #stores the random number on the stack
```

Figure 7: snapshot of our task1 program

Instead of simply storing the generated numbers in a register we decided to store them at set locations on the stack, -4(%ebp) and –8(%ebp) respectively. This was done so we would always know where to expect them.

To verify their locations, we made a modified version of our x86 code to simply print out the stack and show the location of the stored average, in this modified version the random generation was removed so we knew exactly what value to search for. 20 in this instance.

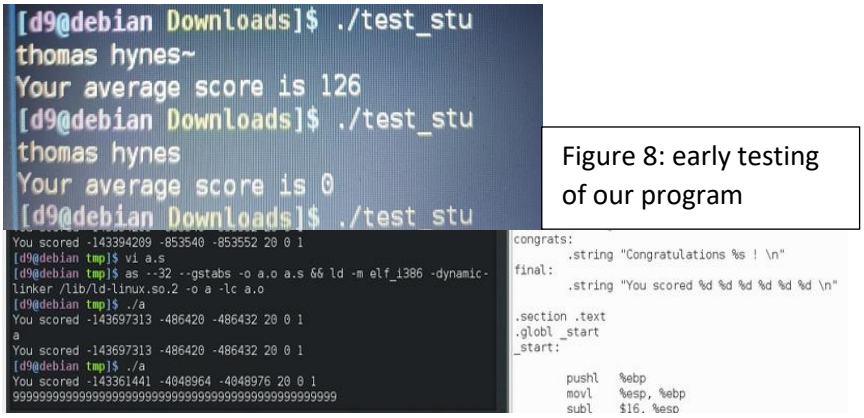

Figure 8: early testing of our program

The program highlighted that the method of storing input we were using at this time was not storing the input on the stack directly, instead storing the address of the actual memory location of the input values. We knew this meant that causing a buffer overflow with this method of storing input would not realistically be possible. In response to this we knew we would want to store our values on directly on the stack, so we set up our reading function to point directly to the address of the stack instead. Additionally, we also switched from using scanf to gets, a function infamous for its lack of a buffer as explained above, this was to prevent any future problems and simplify the design.

For this new redesign we freed up a 32 bytes buffer to store our values. Initially this was only 16 bytes, but we discovered that with such a small buffer, accidental overflows were possible resulting in a switch to 32 bytes. As an example, entering the name “Jamie Sanderson” would cause an overflow as while it would not reach the stored value itself, the string terminating character would be appended over the average, scoring you a 0.

After modifying the size of the buffer, the input now requires at least 27 characters, in order to overflow to the stored average, which we decided was enough for even long names. This is shown in figure 9.

```
secure@secure-virtual-machine:~/Documents/Sec204/vulnerable-student/FINAL TASK1$ ./task1
Please enter your name:
12345678901234567890123456
Your average score is 57
secure@secure-virtual-machine:~/Documents/Sec204/vulnerable-student/FINAL TASK1$
secure@secure-virtual-machine:~/Documents/Sec204/vulnerable-student/FINAL TASK1$ ./task1
Please enter your name:
12345678901234567890123456
Your average score is 36
secure@secure-virtual-machine:~/Documents/Sec204/vulnerable-student/FINAL TASK1$ ./task1
Please enter your name:
123456789012345678901234567
Your average score is 0
secure@secure-virtual-machine:~/Documents/Sec204/vulnerable-student/FINAL TASK1$ ./task1
Please enter your name:
123456789012345678901234567
Your average score is 57
secure@secure-virtual-machine:~/Documents/Sec204/vulnerable-student/FINAL TASK1$ ./task1
Please enter your name:
123456789012345678901234567d
Your average score is 100
secure@secure-virtual-machine:~/Documents/Sec204/vulnerable-student/FINAL TASK1$ ./task1
Please enter your name:
123456789012345678901234567d
Your average score is 100
secure@secure-virtual-machine:~/Documents/Sec204/vulnerable-student/FINAL TASK1$
```

Figure 9: Working buffer overflow exploit within task 1

At first, two 27-character strings are entered, and two random numbers are output by the program. Since the strings are staying within the bounds of the buffer, this will not overflow. However, when another character is appended, the program will always return 0 as the zero terminating character gets appends to the input overwriting the average. Finally, the 29th character will overwrite the average value stored at -4(%ebp). A lower-case d was used here because its ASCII value is 100 and is the highest naturally achievable score.

Conclusion

A student wishing to score a result of their choice will need to insert 28 random characters and then a final ASCII character that corresponds to the score that they want. To strengthen this program, the most immediate way would be to collect and store the input first before generating the random numbers, preventing any input from being able to overwrite them as they wouldn't be stored yet when the user is trying to overwrite them. Then next best idea would be to stop storing the input directly on the stack, as this is how the overflow is caused, if instead the input was stored elsewhere with a pointer placed on the stack in its place it would protect the other elements on the stack from being overwritten as the size of the input would be fixed to an address instead of the variable length raw input. Finally replacing the gets function, returning to scanf and providing a buffer for the function to limit the input size would be an effective way to secure this program.

Task 2: Reverse engineer a binary code file and analyse its execution flow, potential vulnerabilities and bug fixes

Firstly, we identified the file as a 32 bit LSB executable for Intel 80386 using the file command. Being dynamically linked to C libraries, it likely contains C functions including printf and scanf. We initially received the file in read and write only permissions, so we changed them to 777 to allow execution and so there wouldn't be an issue with these in the future.

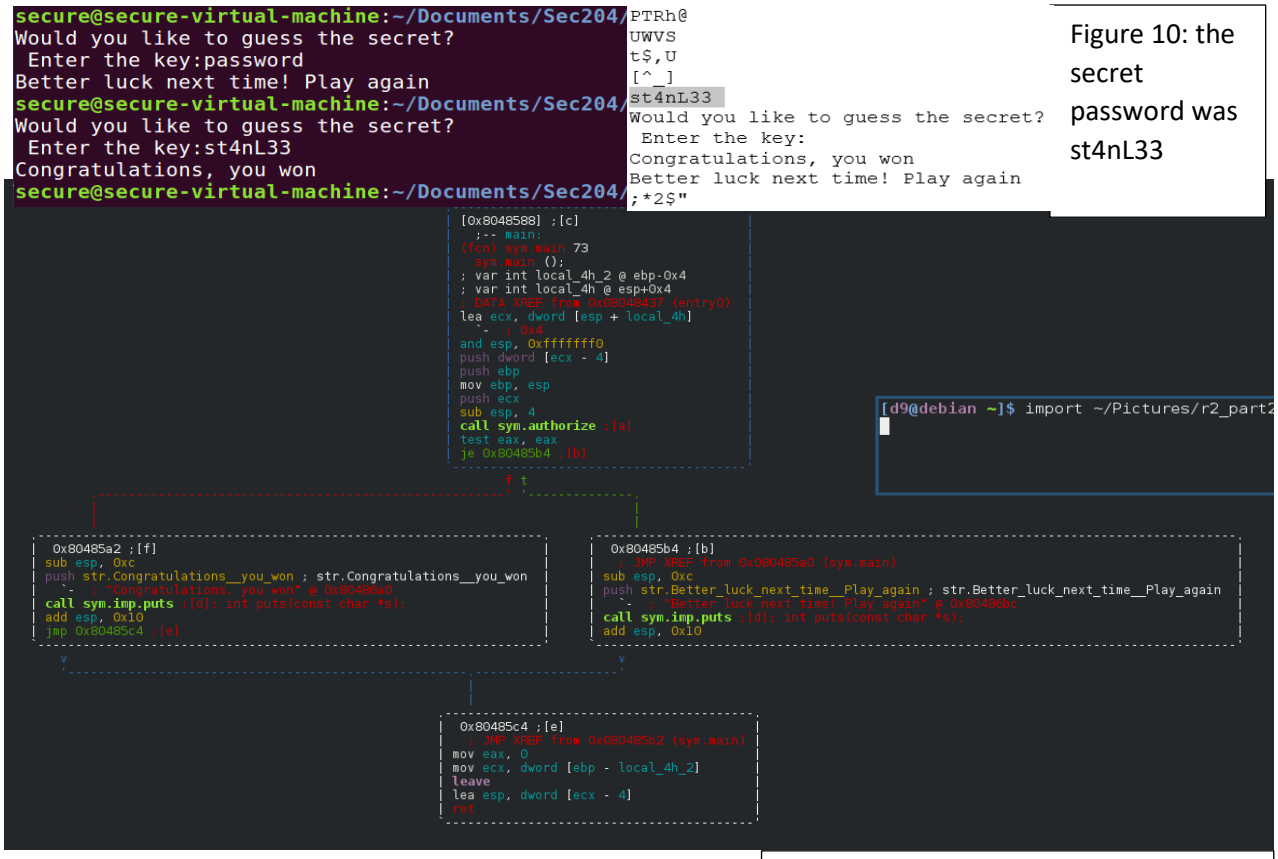

Figure 10: the secret password was st4nL33

Figure 11: Execution path of the part2

Next, we ran the program
which at first glance appeared to be a simple secret key program. The user inputs a key and the program follows an execution path of either correct or incorrect, with the accompanying output attached to this with printf (see figure 10).

Running strings on the program produces a list of all the identifiable variables the program is using, while examining it we spotted where the question was stored and above it on the stack was the answer, st4nL33. This took the program down the success route and displayed the congratulations message.

The program was also disassembled using objdump with the -d (disassemble), -EL (little endian architecture) and --demangle arguments to ensure a readable output. After running the disassembled code through radare2, a visual graph of the execution process was created, shown in figure 11.

As mentioned before, we knew it what the program appreared to be but also suspected that it could have another purpose that we didn't spot. The radare graph debunked these suspicions since it only appeared to have two execution paths, so it was very useful in establishing exactly what functionality exists on the surface. Futhermore, this also identified that both paths are dependant on an authorize function. This is confirmed in the objdump assembly.

The function appears to just be a simple string compare, with it jumping to a specific memory address dependent on whether the strcmp passes or fails. It first prints out "Would you like to guess the secret?". As this is the first printf or puts to be called within the program, this must be true. After reading the user input with scanf and cleaning up the stack, it moves onto comparing it with the secret answer.

After strcmp has been called, the stack pointer is cleaned up and the program performs a bitwise AND (this checks the bits are the same) on the eax register, where the result of the strcmp lies. Based on the SF, if the two outputs are not the same it will clear the eax register, but if they are, it will move 1 into the eax and jump past the instruction that clears the eax register. This is how the program determines whether the user has entered st4nL33 or not, it sets an authentication flag to either 1 (true) or 0 (false) and then exits with that value saved in the eax.

Once it returns to the main function, the program tests the eax registers again and performs a je (jump if equal) function with eax. However, this time it is checking whether the eax is equal to 0 (remembering that if the answer was correct, the eax was set to 1). If it is 0, then the je passes and the program jumps to a puts call which outputs the "Better luck next time" message. If the eax does contain a value however, it will print a different string which is likely to be the "Congratulations, you won" message. The only other relevent actions the program takes is to clean the stack and move 0 into the eax, to prepare for the halt program instruction.

```
804855c:   50                      push   %eax
804855d:   e8 4e fe ff ff          call   80483b0 <strcmp@plt>
8048562:   83 c4 10                add    $0x10,%esp
8048565:   85 c0                   test   %eax,%eax
8048567:   75 07                   jne    8048570 <authorize+0x55>
8048569:   b8 01 00 00 00          mov    $0x1,%eax
804856e:   eb 05                   jmp    8048575 <authorize+0x5a>
8048570:   b8 00 00 00 00          mov    $0x0,%eax
8048575:   8b 55 f4                mov    -0xc(%ebp),%edx
8048578:   65 33 15 14 00 00 00    xor    %gs:0x14,%edx
804857f:   74 05                   je     8048586 <authorize+0x6b>
8048581:   e8 4a fe ff ff          call   80483d0 <__stack_chk_fail@plt>
8048586:   c9                      leave
8048587:   c3                      ret
```

Figure 12: extract from the gdb disassembly of authorize

As for vulnerabilities, an obvious flaw is the inclusion of the password string within the program code. Passwords should be stored externally or encrypted wherever possible to avoid exactly the sort of exploit we ran above. A better method would be to have the user utilise an SSH (not Telnet) connection or a secure website to authenticate themselves online, so the password is inaccessible on the localhost. In addition to this the developer could have included string length verification, code that checks the input is the same length as the password. This could have been done with strlen in C with both the password and the input and another strcmp to check they are the same. Verifying the letters ASCII values could also be done this way too, removing the need for an authentication flag in the eax. Using a 0 or 1 system is dangerous because the variables can be overwritten with a specially crafted buffer overflow or format string. Finally, the developers should consider using an

alternative method to obtain user input. Scanf can cause program crashes if the string input is too long as it only reads characters up to a whitespace character. Providing a buffer length (with malloc) will reserve the space in memory for the input and ensure the next value in the stack is not overwritten.

References

- **Figure 6:** T.Koenig (1993). GETS(3) man page, *Bugs* [online]. Available at man7.org/linux/man-pages/man3/gets.3.html [Accessed 09/01/2019]
- J.Erickson (2008). *Hacking: The Art of Exploitation* [book]. 2nd Edition, Pg 119-193 [Accessed 09/01/2019]