

Giving Students a View of Buffer Overflow with Readily Available Tools

Ms. Cheryl Lynn Resch, University of Florida

BS, MS Mechanical Engineering University of MD MS Computer Science Johns Hopkins University
29 years at JHU Applied Physics Laboratory 12 years in cybersecurity. Cybersecurity architecture for US
government

Experience Report: Giving Students a View of Buffer Overflow with Readily Available Tools

Abstract

Buffer overflow is one of the most common vulnerabilities reported by the Common Vulnerabilities and Exposures (CVE) program. Giving students a mental model of how buffer overflow works and how dangerous these vulnerabilities are may instill in them a commitment to avoiding these vulnerabilities in the future. Buffer overflow and stack frames are known to be a difficult concept for students to understand. This experience report describes a buffer overflow assignment in which students use a free and open-source emulator and debugger to exploit a buffer overflow and view the effect on stack memory. The assignment steps students through assembling a C program vulnerable to buffer overflow in the emulator, running the program in the debugger, exploiting the vulnerable function and causing a buffer overflow, then examining the registers and the stack memory to see the effect of the buffer overflow. Students are then guided through overwriting a function return address saved in the stack with a buffer overflow and causing arbitrary code to run. We provide the assignment instructions and questions that students must answer. 591 students in a Computer Organization course in Fall 2022 were required to complete the assignment. We analyzed the submissions and found that over 90% of students could identify input data when viewing stack memory. 80% could identify that buffer overflow caused a saved register to be corrupted and cause the program to crash. 64% of students could identify saved registers in stack memory and 42% of students could correctly interpret the boundaries of stack frames. These results give insight into the effectiveness of the assignment and which parts are most difficult for students to understand. Students also responded to the reflection prompt “What was the most surprising or interesting part of this activity.” The responses were analyzed for common themes, which were the usefulness of visualizing memory in understanding the concepts of stack frames and buffer overflow, the prevalence of buffer overflow vulnerabilities in publicly available code, and how easy it is to exploit a buffer overflow vulnerability. Thus, this assignment shows promise in helping students to understand a difficult concept, and in emphasizing the importance of avoiding buffer overflow vulnerabilities.

Introduction

Software vulnerabilities in commercial products are an issue of national security, financial and economic stability, and consumer confidence. Data breaches caused by these vulnerabilities can lead to interruptions in public services, monetary loss, and loss of privacy. The 2021 Verizon Data Breach Investigation Report [1] indicates that there were 3,950 data breaches in 2021 in the United States. Software vulnerabilities continue to increase as tracked by National Institute of Standards and Technology (NIST) National Vulnerability Database [2] and MITRE Common Vulnerabilities and Exposures (CVEs) [3]. A 2021 report from Tenable, a leading IT vulnerability assessment and management solution company, indicates that there were 21,957 vulnerabilities reported in 2021, a slight increase from the 18,358 reported in 2020 [4]. Despite increased tracking and abatement of software vulnerabilities, Gueye and Mell [5] report that the most prevalent software errors have not changed much since vulnerabilities were first cataloged. Indeed, MITRE [6] lists the top three software vulnerabilities as:

1. Improper Neutralization of Input on Webpage Generation (cross-site scripting)
2. Out-of-bounds write (buffer overflow)

3. Improper input validation

The prevalence of software vulnerabilities can be reduced when developers use principles of secure programming. It is vital that future developers are taught principles of cybersecurity and secure programming, and that students understand the consequences of buffer overflow attacks.

Review of Stack Buffer Overflow

A buffer overflow vulnerability occurs when input is allowed to be larger than what was allocated for it. This causes neighboring memory to be overwritten. Stack buffer overflow overwrites stack memory. Stack memory contains data pertaining to currently running procedures. Figure 1 shows C code with a 'main' function that calls the function 'get_input'. On the return statement from 'get_input', program control jumps to the instruction in 'main' following the one that called the procedure. When the C code is translated to assembly, there is a register that holds the address of that instruction, called the return address. When procedures are called in a nested fashion, return address values are saved to stack memory. Figure 2 shows a schematic of stack memory when 'main' has called 'get_input'. The memory address of the top of the stack is the stack pointer. The value of the stack pointer also has its own dedicated register. The boundary of a stack frame is the frame pointer, also with its own dedicated register. When a procedure calls another procedure, the current values of the return address, frame pointer, and any local variables are saved into a stack frame. Upon return from the called procedure, the registers for the return address and frame pointer are restored. For example, when 'main' calls 'get_input', the current frame pointer and return address are saved to a stack frame. After 'get_input' completes execution and control returns to 'main', the values of return address and frame pointer are read from the stack and put in the registers. In a buffer overflow, a local variable, e.g. 'buffer' in Figures 1 and 2, is allowed to use more memory than was allocated for it. The overflowed 'buffer' variable will overwrite stack memory. The C language 'gets' function, called in the program in Figure 1, is vulnerable to buffer overflow. The buffer variable will be allowed to use more memory than the 8 bytes allocated for it. If it is large enough, the overflowed data will reach the 'main' stack frame and overwrite the saved return address and stack frame. The corrupted data will be written back to the return address register, causing the program to crash, or, worse, possibly jump to another location in memory to execute code. This is the essence of a buffer overflow.

```
void get_input()
{
    char buffer[8];
    gets(buffer);
    puts(buffer);
    return;
}
int main()
{
    int a = 3;
    get_input();
    return 0;
}
```

Figure 1 – C code with Vulnerable Function

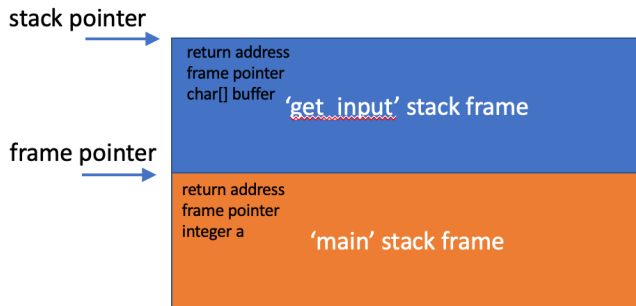


Figure 2 -Schematic of Stack Memory

Understanding stack buffer overflow means understanding stack frames and how local variables are stored in memory. Stack frames are a fundamental concept in Computer Science, but difficult for students to understand. Understanding stack frames requires students to have a correct mental model of the computer running a program [7]. Stack frames are one of the topics that require students to go beyond understanding how to write code, toward understanding how to relate program code to the dynamics of program execution [7].

Previous Work

To help students understand stack frames, Schweitzer and Bolen [8] developed an interactive simulator tool that allows students to step through a C-like program and watch how memory changes. The authors wrote their own language, and had students write programs, run them on the simulator, and view the contents of memory as the simulator runs. They used this tool to teach students about buffer overflow. The basic concepts of buffer overflow, why it is a security threat, and approaches to detect or eliminate them were presented in lecture format, followed by a one-class lab using the simulator. The activity was “highly rated on student critiques as an enjoyable and useful experience” [8, p. 5].

Akeyson et al. [9] also created a visualization tool to help students understand stack buffer overflow. With their tool, students are led in creating a C program, creating payload for a stack buffer overflow attack. They then run the program and are given a simulated view of stack memory and watch their payload overwrite stack memory. The tool supports a very limited version of C.

Sasano [10] describes a tool that is a graphical user interface (GUI) connected to the GNU compiler collection (gcc) debugger (gdb) that visualizes stack memory while any C program is running. The user can run the program in gdb from the GUI. The GUI contains a visualization of memory addresses, with yellow for return addresses, and red when a return address is being overwritten.

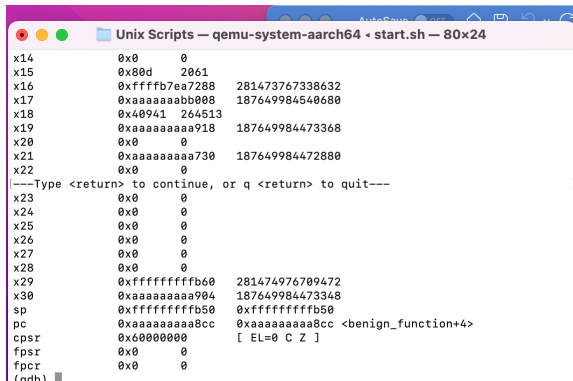
These three tools all attest to the usefulness of visualizing memory in order to understand buffer overflow. However, all three rely on custom made tools that are not publicly available. This paper presents an assignment that teaches students about buffer overflow without specialized visualization tools, but by looking directly at the contents of memory using the gcc debugger (gdb).

Details of the Assignment

This section summarizes the assignment. Students are instructed on how to download the free and open source emulator QEMU [11] and are provided with a disk image that uses the ARMv8 instruction set architecture. Students are given instructions on installing and starting up the emulator. Students then carry out the detailed instructions of the assignment, found in Appendix A.1.

Students are led through compiling and running a vulnerable C program, found in Appendix A.2. The program has a main function, which calls 'benign_function', then 'get_input'. The 'get_input' function calls the C function 'gets' that is vulnerable to buffer overflow.

Students are given instructions on how to set a break point at the start of execution of 'benign_function' then view registers (Figure 3) and note the values of the frame pointer (X29) return address (X30) and stack pointer (SP). Next students are instructed to view stack memory (Figure 4). Students are required to use the stack pointer and frame pointer values to find the boundaries of the stack frame in memory.



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

Register	Value
x29	
x30	
SP	

Table 2

Figure 3 – Viewing Registers

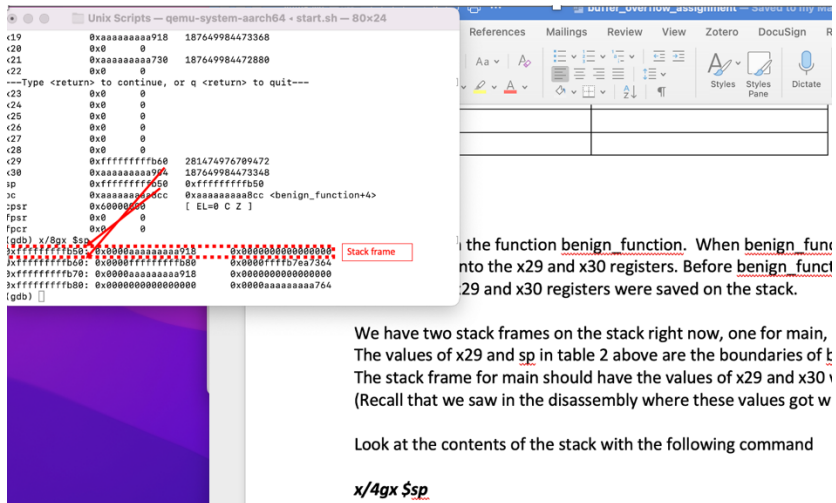


Figure 4 – Viewing the Stack

Next, students step through the program to the vulnerable ‘gets’ function, provide input longer than the 8 bytes allocated for it, and view the stack (Figure 5). As shown in Figure 5, a large number of B’s was input. The hexadecimal ASCII representation of ‘B’ is 42. Figure 5 shows the B’s overflowing the space allocated for it, and overwriting stack memory below it, where the frame pointer and return address for ‘main’ were stored.

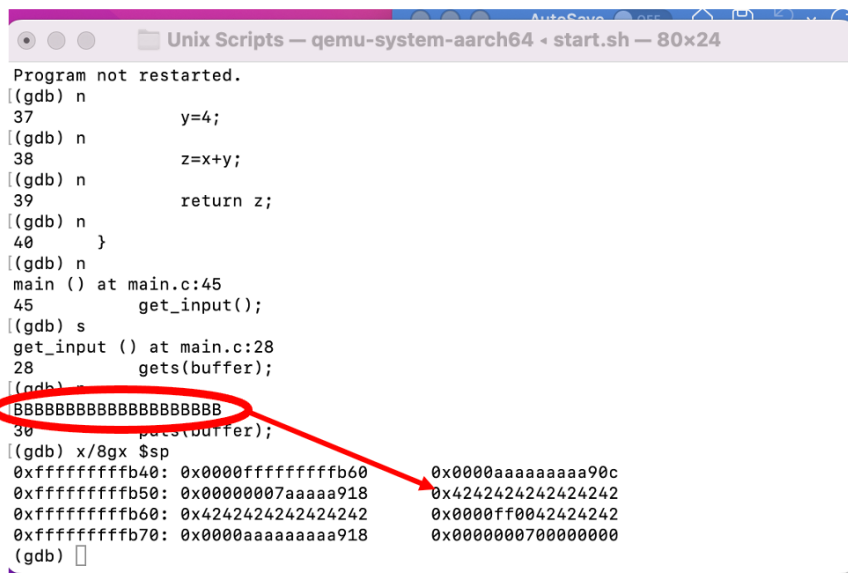


Figure 5 – Viewing a Buffer Overflow

Next students are guided through exploiting the buffer overflow vulnerability by running the vulnerable code from the command line. Students are then introduced to Address Space Layout Randomization (ASLR), which is a mitigation for buffer overflow vulnerabilities. Students are instructed to turn off ASLR, then guided through crafting an input that causes the program to execute a procedure that was not called (Figure 6).

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.

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

```
echo -e "ABCDEFGHJKLMNOP\x78\xa8\xaa\xaa\xaa\xaa" | ./main
```

Figure 6 – Instructions for Crafting Input to Cause Arbitrary Code to Run

Finally, students are asked to fix the vulnerability, and look up a current buffer overflow vulnerability in the Common Vulnerabilities and Exposures (CVE) website [12].

The rubric for the assignment is found in Table 1. The total score is 100. The rows in italics are tasks that can be completed by simply following instructions. These consist of 62 points. The rows in bold require students to use information from the tasks or interpret what they see. These consist of 35 points. The final 3 points asks students to think about why there are still buffer overflow vulnerabilities.

Item	Points
<i>Task 1 – compile and run the program, provide a screen shot with your name</i>	2
<i>Task 2 – Table 1 – provide the address of branch and link instructions</i>	2
<i>Task 2 – Table 2 – fill in register values for frame pointer, stack pointer, return address</i>	5
<i>Task 2 – Table 3 – fill in table with addresses and contents of stack memory</i>	5
Task 2 – Use the information from tables 2 and 3 to determine the boundaries of the stack frames for the current function and the main function that called it	5
Task 2 – Previous values of X29 and X30 – use the contents of stack memory to determine the previous values of frame pointer and return address, when main called the current function	5
<i>Task 2 – Table 4 – fill in register values for frame pointer, stack pointer, return address (the vulnerable function has now been called)</i>	5
<i>Task 2 – Table 5 - fill in table with addresses and contents of stack memory</i>	5
Task 2 – What does the data in the top row of Table 5 represent? (It is the saved frame pointer and return address for the call to the vulnerable function)	5
Task 2 – Highlight the data in Table 5 that you typed in when asked for input	5

<i>Task 3 - Table 6 - fill in table with addresses and contents of stack memory – after a buffer overflow</i>	5
Task 3 – What has happened to the stack frame for main? (It has been overwritten with buffer data)	5
<i>Task 3 – Table 7 - fill in register values for frame pointer, stack pointer, return address (after corrupted data from the stack was written into them)</i>	5
Task 3 – What happened to X29 and X30? – they were corrupted by the buffer overflow	5
Task 3 – What happened and why? The program tried to branch to the corrupted X30 value and cause a segmentation fault	5
<i>Task 4 – Arbitrary code address – Write down the address of the first instruction in the “Arbitrary code” function</i>	2
<i>Task 5 – Run code from command line, overflow the buffer. What happens? Segmentation fault</i>	2
<i>Task 6 – What is ASLR – research and describe address space layout randomization</i>	10
<i>Task 6 – Turn off ASLR. Craft input that includes the address of the Arbitrary code function. What happens? It runs Arbitrary code</i>	2
<i>Task 7 – Turn off ASLR, run the code with the same input. What happens? Segmentation fault</i>	2
<i>Task 8 – Fix the vulnerability. What happens? Only 8 characters are read.</i>	2
<i>Task 9 – Look up a buffer overflow vulnerability in the CVE website and describe it.</i>	8
<u>Task 9 – Why are there still vulnerabilities like this?</u>	<u>3</u>
	100

Table 1 – Buffer Overflow Assignment Rubric

Data Collection and Analysis

591 students in a Computer Organization course at our large R1 university were given this assignment in Fall 2022. The course is required for Computer Science and Computer Engineering majors and Computer Science minors. 582 students completed the assignment and submitted it to our learning management system (LMS). The assignments were graded using a rubric in the LMS. The average score on the assignment was 92.6 out of 100. The average total score on questions in bold in Table 1 that require interpretation of the stack memory contents was 29.1 out of 35. The average score on the parts of the assignment that required simply following instructions was 63.5 out of 65. Students were able to follow instructions and complete the assignment.

Item	Points	Average Score	Number with Full Credit (N=582)
Task 2 – Use the information from tables 2 and 3 to determine the boundaries of the stack frames for the current function and the main function that called it	5	3.38	247

Task 2 – Previous values of X29 and X30 – use the contents of stack memory to determine the previous values of frame pointer and return address, when main called the current function	5	3.51	370
Task 2 – What does the data in the top row of Table 5 represent? (It is the saved frame pointer and return address for the call to the vulnerable function)	5	4.48	490
Task 2 – Highlight the data in Table 5 that you typed in when asked for input	5	4.59	526
Task 3 – What has happened to the stack frame for main? (It has been overwritten with buffer data)	5	4.27	478
Task 3 – What happened to X29 and X30? – they were corrupted by the buffer overflow	5	4.36	482
Task 3 – What happened and why? The program tried to branch to the corrupted X30 value and cause a segmentation fault	5	4.54	499
	35	29.13	155

Table 2 – Scores on Questions Requiring Interpretation

Table 2 shows the average score and number of students with full credit on each question that required students to interpret what they see. 526 out of 582 could find their input data when viewing stack memory (see Figure 3). 478 could identify that the stack frame was overwritten by the input data, and 482 could identify that frame pointer and return address values saved to the stack were overwritten by input data. 499 identified that the return address was corrupted and caused a segmentation fault.



Figure 7 – Solution to Stack Frame Boundary Question

The lowest scores were for the activities that ask students to look at stack memory and identify the boundaries of two stack frames, and that asked students to find saved values of the frame pointer and return address on the stack. Figure 7 shows details on the correct solution to the question asking for boundaries of stack frames. The stack pointer is the upper boundary of the current stack frame, or 0xffffffffb50 in Figure 7. The frame pointer is the lower boundary of the current stack frame, or 0xffffffffb60 in Figure 7. The lower bound of the main stack frame is the previous frame pointer value that had been saved on the stack, or 0xffffffffb80 in Figure 7.

```

Unix Scripts — qemu-system-aarch64 - start.sh — 80x24
x19      0xaaaaaaaa918  187649984473368
x20      0x0          0
x21      0xaaaaaaaa730  187649984472880
x22      0x0          0
---Type <return> to continue, or q <return> to quit---
x23      0x0          0
x24      0x0          0
x25      0x0          0
x26      0x0          0
x27      0x0          0
x28      0x0          0
x29      0xffffffffb60  281474976709472
x30      0xaaaaaaaa904  187649984473348
sp       0xfffffffffb50  0xfffffffffb50
pc       0xaaaaaaaa8cc  0xaaaaaaaa8cc <benign_function+4>
cpsr    0x60000000  [ EL=0 C Z ]
fpsr    0x0          0
fpcr    0x0          0
(gdb) x/8gx $sp
0xfffffffffb50: 0x0000aaaaaaaa918  0x0000000000000000
0xfffffffffb60: 0x0000fffffffffb80  0x0000fffffb7ea7364
0xfffffffffb70: 0x0000aaaaaaaa918  0x0000000000000000
0xfffffffffb80: 0x0000000000000000  0x0000aaaaaaaa764
(gdb)

```

Figure 8 – Solution to Saved Frame Pointer and Return Address Question

Figure 8 shows the solution to the question that asks students to look at stack memory and identify frame pointer (X29) and return address (X30) values that had been saved to the stack.

After completing the assignment, students were given the opportunity to receive extra credit for answering a reflection prompt and consenting to have their responses used in our research. This protocol was approved by the Institutional Research Board (IRB) at our university. 463 students answered the following reflection prompt:

- What was the most interesting or surprising thing you learned from this activity?

The students answered the questions in the LMS. The responses were downloaded from the LMS and read by the author to find common themes. The most common themes in responses to what was interesting or surprising were:

- The continuing prevalence of buffer overflow vulnerabilities (n=92, 20%)
- How easy buffer overflow vulnerabilities are to exploit (n=76, 16%)
- How visualizing memory helped them understand buffer overflows and the stack (n=76, 16%)

The last theme listed indicates that many students found the hands-on visualization in the assignment very useful in understanding the how memory and stack frames work. Specific quotes include:

- “It was interesting to see how the register values get saved on stack frames and how the stack pointer moves.”
- “It helped me see what is actually in the stack. Not only that but it connected information I had previously known with actual results. Seeing where registers are placed on the stack and how they can be overwritten was very enlightening.”

- “I really liked being able to see the values in memory change as the buffer got overflowed. It was genuinely interesting to see what buffer overflow meant in action. It was also interesting to step through a c function and see the memory in general. It made the things we learn in class seem more useful and it was a good application.”

Discussion

Almost all students were able to run the program in the debugger, and view registers and stack memory. 90% of students were able to identify input data when viewing stack memory. This indicates that the assignment instructions effectively guide students to a view of what happens in a stack buffer overflow. However, they had more difficulty identifying saved frame pointer and return addresses and identifying the function that the saved frame pointer refers to. This is not surprising, as stack frames are known to be a difficult concept for students to understand. Over 80% of students correctly stated that the buffer overflow caused a return address to be corrupted and cause a segmentation fault.

One of the most common themes found in reflection prompts was the usefulness of the activities in the assignment in helping them to understand how stack memory works. Even though identifying data in the stack was the most difficult activity, many acknowledged that this view of memory was enlightening and helped them understand the stack frames and memory. Thus, this assignment that uses freely available resources is promising as a tool for giving students hands-on experience with a difficult concept that is vital for students to understand.

Another common theme in the reflection responses was the prevalence of buffer overflow vulnerabilities in publicly available software. Students noticed this because they were required to look up a current buffer overflow vulnerability in the MITRE CVE website. Thus the assignment was also effective in helping students understand the importance of avoiding buffer overflow vulnerabilities.

Conclusions

This paper presents an assignment that uses freely available tools to show students the effect of a buffer overflow vulnerability. The assignment features detailed instructions that step students through exploiting a buffer overflow vulnerability and viewing stack memory and how it changes. Over 90% of students could identify input data when viewing stack memory. 80% could identify that buffer overflow caused a saved register to be corrupted and cause the program to crash. The activities that the students had the most difficulty with were identifying saved registers in stack memory and interpreting the boundaries of stack frames. A common theme students reported in reflection responses was that being able to visualize memory helped them to better understand stack frames and that watching what a buffer overflow does to memory was very enlightening. This assignment is shown to be effective in helping students understand and appreciate buffer overflow vulnerabilities, using freely available resources.

References

- [1] Bassett, G., Hylender, D., Langlois, P., Pinto, A., & Widup, S., “Data Breach Investigations Report” Available:<https://www.verizon.com/business/resources/reports/dbir>, 2022.

[2] “NIST National Vulnerability Database” Available: <https://nvd.nist.gov/vuln/data-feeds>, 2023.

[3] MITRE Common Vulnerabilities and Exposures. Available: https://cve.mitre.org/cve/data_feeds.html, 2023.

[4] “Tenable’s 2021 Threat Landscape Retrospective” Available: <https://www.tenable.com/cyber-exposure/2021-threat-landscape-retrospective>, 2022.

[5] Gueye, A., & Mell, P. “A Historical and Statistical Study of the Software Vulnerability Landscape”, *arXiv preprint arXiv:2102.01722*, 2021.

[6] “MITRE 2020 CWE Top 25 Most Dangerous Software Errors, 2021”, Available: https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html, 2022.

[7] Sorva, J., Karavirta, V., and Malmi, L., “A review of generic program visualization systems for introductory programming education”. *ACM Trans. Comput. Educ.* 13, 4, Article 15, DOI:<http://dx.doi.org/10.1145/2490822>, 2013.

[8] D. Schweitzer and J. Boleng, “A simple machine simulator for teaching stack frames,” *Proceedings of the 41st ACM technical symposium on Computer science education*, pp. 361–365, 2010.

[9] Akeyson, E., Ramaprasad, H., & Sridhar, M. “DISSAV: A Dynamic, Interactive Stack-Smashing Attack Visualization Tool,” *Journal of The Colloquium for Information Systems Security Education* (Vol. 9, No. 1, pp. 8-8), 2022.

[10] Sasano, I., “A tool for visualizing buffer overflow with detecting return address overwriting”, *Proceedings of the 9th EAI International Conference on Bio-inspired Information and Communications Technologies (formerly BIONETICS)*, 2016.

[11] “QEMU – A generic and open source machine emulator and virtualizer”, Available: <https://www.qemu.org/>, 2023.

[12] “Mitre common vulnerabilities and exposures”, Available: https://cve.mitre.org/cve/data_feeds.html, 2023.

Appendix

A.1 Assignment

Look at main.c. Describe what it does.

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.

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.

Instruction	Address
bl benign_function	
bl get_input	

Table 1

Note that in the first line of the disassembly code that the `stp` function is run. `stp` is store pair and

Memory Address	Value	Value
0xfffffffffaf0	0x0000aaaaaaaa918	0x0000000000000000
0xfffffffffb00	0x0000ffffffffffb20	0x0000ffffb7ea7364

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?

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.

ir

Fill in the following table with register values

Register	Value
<code>x29</code>	
<code>x30</code>	
<code>SP</code>	

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

Memory Address	Value	Value

Table 3

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.

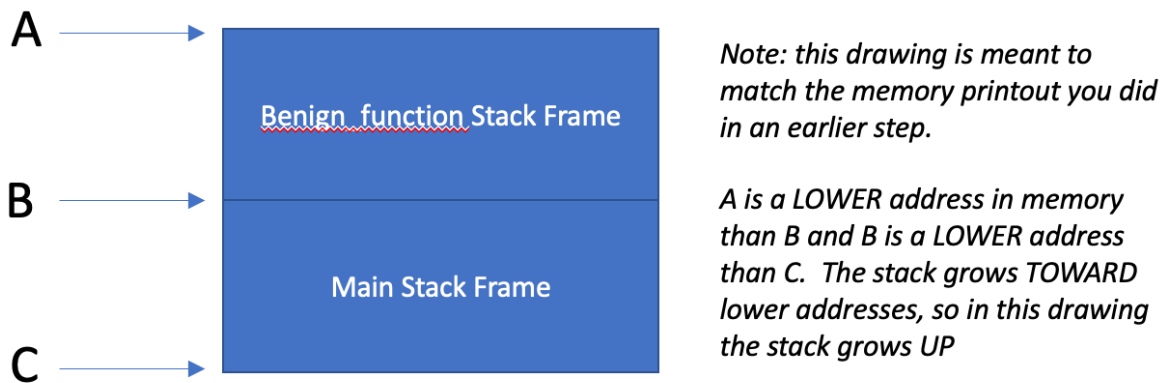


Figure 2

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?

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?

Type *n* repeatedly until you have entered get_input, main.c:45

Type

ir

Fill in the following table with hexadecimal register values

Register	Value
x29	
x30	
SP	

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/4gx \$sp*

Fill in the following table.

Memory Address	Value	Value

Table 5

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

Circle or highlight the 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 $x/20x \$sp$

Fill in the following table

Memory Address	Value	Value	Value	Value

Table 6

What has happened to our stack frame for main?

Type n repeatedly until you get some question marks

Display register values with the following command

ir

Fill in the following table with register values

Register	Value
x29	
x30	
SP	

Table 7

What happened to our x29 and x30 value?

Type n

What is the result? Why did this happen?

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?

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?

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

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.

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

echo -e "ABCDEFGHJKLMNOP\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?

(Control C to make execution stop)

Task 7 – Turn ASLR back on

Type *echo 1 > /proc/sys/kernel/randomize_va_space*

Type *echo -e "ABCDEFGHJKLMNOP\x78\x88\xaa\xaa\xaa\xaa" | ./main*

What happens?

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?

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?

A.2 Vulnerable Code

```
/* * bf vuln.c
 * This program is vulnerable to buffer overflow
 * and is solely for demonstrating purposes
 * Author: Nik Alleyne
 * blog: http://securitynik.blogspot.com
 * Date: 2017-01-01
 */
#include <stdio.h>
#include <unistd.h>
// This will be our arbitrary code
// Nothing malicious just fun

void arbitrary code()
{
    printf(" Now you know I should not be seen !");
    printf(" But I am. Don't believe me just watch ");
}
//Our vulnerable function
void get input()
{
    //declaring our buffer size of 8 bytes
    char buffer[8];
    //read our input. This 'gets' function is where our problem really lies
    gets(buffer);
    //print the output back to the screen.
    //Simply echo our input back to the screen
    puts(buffer);
}

int benign function()
{
    int x, y, z;
    x=3;
    y=4;
    z=x+y;
    return z;
}

int main()
{
    int a;
    a = benign function();
    get input();
    return 0;
}
/* Note: Nowhere in the above code did we call 'arbitrary code'
 * However, we will see shortly that we can execute this code
 * by using its memory location
 */
```