

CS 499/ISA 564: Lab 1 – Buffer Overflows and Shellcode

Lab Submission Guidelines

Submissions must be in Microsoft Word or PDF format. Be sure to clearly label what question you're answering. Where possible, screenshots should be embedded directly in the document. Screenshots should be cropped to only include what is necessary to answer the question. If you need to save them as a separate file, save them in compressed format (gif, jpg, png) and name them after the question they pertain to. If your submission contains multiple files, archive them (zip, 7z, tar.gz) and submit the file via Blackboard.

Lab Requirements

- Kali Linux VM
- Files in the Lab1 Resources folder
- (Optional) GDB Reference.pdf in the Readings folder

Kali VM Setup

- Go to <http://www.kali.org/downloads> and download either the Kali 32 bit ISO or the Kali Linux 32 bit VMware VM PAE Image
- Setup your VM if you downloaded the ISO or open the VMX file if you downloaded the VM image
- When running any of the *apt-get* commands, if you get an error like “could not get lock” or “unable to lock”, reboot your VM
- Update package repositories – *apt-get update*
- Upgrade installed packages – *apt-get dist-upgrade -y*
- Install Open VM Tools – *apt-get install open-vm-tools-desktop -y*
- Disable ASLR – *echo "kernel.randomize_va_space = 0" > /etc/sysctl.d/01-disable-aslr.conf*
- Reboot your VM – *reboot*

Lab 1: Task 1 – Understanding the Stack

For this exercise we will be using Lab1.1.c and Lab1.2.c. The purpose of this exercise is to gain an understanding of the Stack including memory layouts, stack frames, and to learn the value of controlling EIP.

Question 1.1 – Compile Lab1.1.c into binary form using GCC then execute the resulting binary. What were your results? Why did you get that result?

Question 1.2 – Debug your compiled binary in GDB. Set a breakpoint on the call to function(). Describe how the stack frame changes when function() is called (registers, memory addresses). Use the *info frame* command to get basic information about the current stack frame. Do *echo "set disassembly-flavor intel" >> ~/.gdbinit* if you prefer Intel over AT&T syntax.

Compile Lab1.2.c into binary form using GCC. The purpose of this exercise is to learn how to exploit control of a stack pointer to manipulate a function's return address and thereby modify program behavior. The objective is to modify function()'s return address to skip over the assignment *x=1* in *main()*. Successful output should print '0' and not produce a segmentation fault. You will need to edit two lines of code to make this exploit work:

- Line 8 needs to be edited such that *ret* points to function()'s return address. To determine the correct offset, find the value of *ret* after the assignment and compare it to the saved EIP register
- Line 9 needs to be edited such that **ret* increments the return address to point to the instruction after the *x=1* assignment. To determine the correct offset, disassemble *main*, find the instruction that sets *x=1*, and get the byte count of that instruction

Question 1.3 – Explain how lines 8 and 9 are able to change the execution flow of the program.

Question 1.4 – Provide screenshots showing your code modifications and successful execution of the exploit.

Lab 1: Task 2 – Shellcode

For this exercise we will be using Lab1.3.c and Lab1.4.c. The purpose of this exercise is to gain an understanding of assembly/disassembly, shellcode, and the challenges that arise from attempting to derive shellcode from higher level C code.

Question 2.1 – Compile Lab1.3.c into binary form using GCC. Run `objdump -d -j.text` on your compiled binary. Add the `-M intel` switch if you prefer Intel syntax and optionally pipe the output to a file. This command will produce assembly output including all opcodes and mnemonics for all functions within the binary. Annotate the assembly code for the main function in Lab1.3.c describing what each line does. Your annotation should include the contents of the EAX register before the first system call.

Question 2.2 – Describe what issues you would run into if you tried to use the assembly output from Lab1.3.c directly as shellcode. Remember that shellcode has character limitations and has to be position independent. How would you overcome these issues?

Lab1.4.c contains three shellcode arrays of increasing complexity. Our objective is to gain an understanding of what the shellcode does and what value it might have for an attacker. The first step is to disassemble the shellcode. There are many free and commercial disassemblers in existence. We will cover disassemblers in painfully exquisite detail later in the course, for now you can use a free, online disassembler for this exercise. <https://defuse.ca/online-x86-assembler.htm>

Question 2.3 – Disassemble and annotate the shellcode in the `shellcode1[]` array in Lab1.4.c. Describe how this shellcode overcomes the obstacles you documented in Question 2.2.

Question 2.4 – Disassemble and annotate the shellcode in the `shellcode2[]` array in Lab1.4.c. Compile Lab1.4.c into binary form using GCC (use `-z execstack`) and execute your compiled binary. Take a screenshot that demonstrates successful shellcode execution. This shellcode does not just pop a shell so your screenshot should show the actual system change. Describe what value this shellcode could have for an attacker.

Extra Credit (10 points) – The `shellcode3[]` array in Lab1.4.c is obfuscated. Disassemble and annotate the deobfuscation routine. There are two stages to the deobfuscation routine. Explain what value obfuscating shellcode would provide to an attacker. Note that fully deobfuscating this shellcode is very challenging and requires advanced debugging skills. Partial extra credit will be given for partial deobfuscation and a good explanation of the value of obfuscating shellcode.

Lab 1: Task 3 – Stack-Based Buffer Overflow

For this exercise we will be using Lab1.5.c. The purpose of this exercise is to learn how to perform classic stack-based buffer overflow exploits against vulnerable software as well as how operating system defenses such as DEP and stack cookies stymie our efforts. Finally we will look at the challenges of writing effective exploit shellcode and how to debug it.

Compile Lab1.5.c into binary form using GCC (use `-z execstack`). Some helpful tips for the following questions:

- Use the Python command `$(python -c 'print(""\x90"*xx + "\xDE\xAD\xBE\xEF")')` to construct and run your exploit string. 'xx' is the number of times to repeat `\x90`. `\xDE\xAD\xBE\xEF` can be any bytes
- GDB's *info frame* command will display the address and value of EIP
- Find the address of Locals within the stack frame and then do `x/40x <address>-20` to get a view of the stack after your attempted overwrite. This can help you determine the length of your exploit string
- You can use Lab1.4.c to get the length of the shellcode (as well as reassure yourself that it works)
- When debugging your shellcode, use the *watch* (watchpoint) command to break execution when the value at an address changes and use the *bt* (backtrace) command to see the last executed instruction after a segmentation fault

Question 3.1 – Construct a proper NOP-sled + shellcode + return address overwrite string and provide a screenshot showing the output of *info frame* and a memory view of the Stack showing successful overwrite of the return address with a value pointing into your NOP-sled. Use the Stack Shellcode byte array.

Question 3.2 – Provide a screenshot showing how despite all your hard work with correctly overwriting the return address, the program still had the temerity to throw a segmentation fault in your face. Also provide screenshots showing exactly how the segmentation fault happened. These screenshots should include the memory address both before and after it gets overwritten by the shellcode.

Question 3.3 – The problem encountered in Question 3.2 can be solved in one of two ways, either reconstruct your exploit string to be NOP-sled + shellcode + NOP-sled + return address so that the length is the same but the push to the Stack doesn't overwrite your shellcode (experiment with the before and after NOP lengths until it works), or you can use the Non-Stack Shellcode byte array. Choose an approach and provide a screenshot showing your overwrite string and successful execution of your strategy.

Extra Credit (10 points) – Use both approaches. Provide screenshots for both showing your exploit string and successful execution of the shellcode.

Question 3.4 – Recompile Lab1.5.c into binary form using GCC but this time omit `-z execstack`. Provide a screenshot showing your monumental (albeit self-imposed) failure to successfully execute your shellcode. Explain how/why this happened.

Question 3.5 – Recompile Lab1.5.c into binary form using GCC but this time compile with `-fstack-protector-strong`. Provide a screenshot showing your catastrophic (albeit contrived) failure to successfully execute your shellcode. Explain how/why this happened.

Lab 1: Task 4 – Format String Exploitation

For this exercise we will be using Lab1.6.c. The purpose of this exercise is to exploit a format string vulnerability to read and write arbitrary memory.

Compile Lab1.6.c into binary form using GCC. In order to successfully exploit the vulnerability you need to perform the following steps:

- Find the offset from EBP that stores the address of the variable you want to read/write
- Determine how many 4-byte chunks on the stack you need to traverse to get to your desired address
- Craft an appropriate format string to perform the stack traversal and read/write of your desired address
- Use the Python command `$(python -c 'print <address>' + "%08x-"*<number of 4-byte chunks> + "%s"/"%n"` to read/write respectively

Question 4.1 – Construct a Format String exploit to read the contents of `secret[]`. Provide a screenshot showing successful output including your exploit string.

Question 4.2 – Construct a Format String exploit to change the value of `magic` such that the program prints "GREAT SUCCESS!" Provide a screenshot showing successful output including your exploit string.

Grading

- Task 1 – 20 points (5 points per question)
- Task 2 – 30 points (7.5 points per question)
- Task 3 – 35 points (7 points per question)
- Task 4 – 15 points (7.5 points per question)
- Extra Credit – 20 points