CS 499/ISA 564: Lab 2 – Advanced Exploitation

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 Lab2 Resources folder
- PEDA and ROPgadget

PEDA Installation

- Copy the peda folder from Lab2 Resources into the /root/ folder in your Kali VM
- Integrate PEDA into GDB - *echo "source ~/peda/peda.py" >> ~/.gdbinit*

ROPGadget Installation

- *pip install ropgadget*

Lab 2: Task 1 – Return-Into-Libc

For this exercise we will be using Lab2.1.c. The purpose of this exercise is to learn how to leverage the return-into-libc technique to exploit a stack-based buffer overflow vulnerability in a binary that has been compiled with a non-executable stack.

Compile Lab2.1.c into binary form using GCC. To perform a successful return-into-libc attack, you will need to do four things, all of which can be accomplished with the GDB or PEDA *find* and *print* commands.

- Find the return address like with any other stack-based buffer overflow exploit
- Find the address of the system function
- Find the address of the exit function
- Find the location of a "/bin/sh" string in process memory

Note that if there are certain bytes in the function/string addresses like \x20, this will break Python's string parsing. You need to enclose the entire Python expression in double quotes

Question 1.1 – Create a return-into-libc exploit that spawns a shell and exits from the shell without causing a segmentation fault. Provide a screenshot of your exploit string. Also provide a screenshot of the successful execution of, and exit from, the shell without producing a segmentation fault.

Extra Credit (20 points) – Use Lab2.1-EC.c to perform the advanced return-into-libc technique known as "frame faking". This version of Lab2.1 prints the address of buf so you can use it as a reference for the fake frames you'll be creating. This technique allows you to chain together multiple libc function calls. String together a sequence of three libc function calls, starting with *puts()* to output a string of your choice to stdout followed by *system()* with the argument "/bin/sh" to pop a shell and finally *exit()* to gracefully terminate. Provide a screenshot showing your exploit string, the output of your chosen string, a shell prompt, and exiting from the shell without producing a segmentation fault. You'll want to consult "Advanced Return-Into-Libc.mht", specifically section 3.3, and "Chained Return-Into-Libc.htm" both located in the Readings folder for guidance.

Lab 2: Task 2 – Return-Oriented Programming Part 1

For this exercise we will continue to use Lab2.1.c. The purpose of this exercise is to learn how to leverage return-oriented programming to exploit a stack-based buffer overflow vulnerability in a binary that has been compiled with a non-executable stack.

Recompile Lab2.1.c into a different binary using GCC (use *–static*). At this point you should have two binary versions of Lab2.1.c, one that is dynamically linked (the default) and another that is statically linked. Run ROPgadget against both binaries using this command: *ROPgadget --ropchain --binary <filename>*. You may wish to consider redirecting output to a file.

Question 2.1 – For both the static and dynamic versions of the binary, answer the following questions:

- How many ROP gadgets were discovered?
- Was ROPgadget able to build a full ROP chain? Why or why not?

ROPgadget will generate a full ROP chain for one of the two binaries. Use the ROP chain to craft an exploit for that binary; to do so you'll need to add a line *p += "<random byte>"*<offset to return address>* to the start of the code as well as a line *print p* at the end. Save the code to a file, make it executable, and remove the leading whitespace. Then use command substitution to supply the output of the file to the vulnerable binary. (e.g. *./Lab2.1-Static $(./ropstat.py)*)

Question 2.2 – Provide a screenshot showing successful execution of your ROP chain leading to a shell being launched. Also provide the Python file you generated or a screenshot of it.

Lab 2: Task 3 – Return-Oriented Programming Part 2

For this exercise we will be using Lab2.2.c. The purpose of this ROP exercise is to perform a stack pivot to execute shellcode stored on the Heap.

Compile Lab2.2.c into binary form using GCC (use *–static –z* execstack). Use whatever shellcode you wish. Some careless programmer (spoiler alert: it was me) left you a stack pointer that you can manipulate to point to a ROP gadget to do the stack pivot. That same careless programmer also allowed you to input arbitrary data into a buffer on the heap.

Your modification of the stack pointer should be the delta between the return address and your chosen ROP gadget (subtract the return address from the ROP gadget address). Note that it has to be the return address from *function()*. Your chosen ROP gadget should transfer control to your shellcode on the heap. Hint, that assembly line is there for a reason. Use ROPgadget or PEDA's *ropsearch* command to search for applicable gadgets. A traditional stack pivot ("push GPR; pop esp") gadget might not be available. There are other ways to transfer flow control to an address stored in a register however.

Question 3.1 – Provide a screenshot showing your input string and successful execution of your shellcode.

Question 3.2 – Why, even though the shellcode is being executed from the heap, did we need to compile the binary with *–z execstack*? Hint, try compiling without it and check the memory maps using PEDA's *vmmap* command (you'll need to run the program first).

Lab 2: Task 4 – Heap Spraying

In this exercise you will be working with Lab2.3.c. The purpose of this exercise is to perform a heap spray leading to execution of shellcode on the heap. To do so you will need to do the following:

- Acquire working Linux shellcode that pops a shell and add it to the Shellcode[] array
- Properly set the NOPSize and ShellcodeSize variables
- Find the location of the memory map associated with HeapBuf and mark it executable using *mprotect()*

To acquire working shellcode, you can use PEDA's *shellcode search* command, use any example shellcode from class, or find your own somewhere in the great vastness of the Internetz. Use \x encoding (e.g. \x90).

In order to satisfy the requirements of the question, your NOP sled must be sufficiently large as to guarantee a 95% success rate for execution. Play around with the size of the sled until you meet this requirement.

mprotect() is a Linux syscall and it is well documented online. Learn how to construct a proper call to set a memory segment to have RWX permissions. You will need to know the starting address of the memory segment you want to change the permissions of as well as its size. The starting address will always be at a static offset from the variable that receives the allocated memory (HeapBuf). Use your l33t debugging skills to figure out what this offset is. Use PEDA's *vmmap* and GDB's *info proc mappings* commands to find the memory map associated with HeapBuf and its size.

Finally, note that if you add or remove stack variables, the location of Address might change which will break the asm code on line 52. You are thus recommended to not do so. If you absolutely feel the need to do so anyway, you'll need to figure out the new opcodes for the call instruction to Address.

Question 4.1 – Provide a screenshot showing successful execution of your chosen shellcode. Also provide either your customized Lab2.3.c or a screenshot of it. You must construct a proper call to *mprotect()*. You must also use the structure I've provided with a jump to a random address within the memory map. Finally, your output must show a 95% or higher success rate for the execution simulation.

Grading

- **Task 1 – 20 points (20 points per question)**
- **Task 2 – 30 points (15 points per question)**
- **Task 3 – 20 points (10 points per question)**
- **Task 4 – 30 points (30 points per question)**
- **Extra Credit – 20 points**