COMP 4384: Assignment #3

Revision 2

Due on December 15, 2020 at 1:00 PM

130 Points (20% Overall)

In this assignment, we will experiment with a small vulnerable code to perform a buffer overflow attack. You are expected to take screenshots of the various step as well as comment on them and reason about the different values. You are expected to turn in a PDF with you solution.

1 Virtual Machine Setup (10 points)

For this assignment, we will perform our buffer overflow attack within a virtual machine that is pre-configured to disable different security countermeasures.

1. Download Virtual Box (6.0.x) from `https://www.virtualbox.org/wiki/Download_Old_Builds_6_0`.
2. Download the pre-configured virtual machine image from `http://www.benjaminsbox.com/pac/HackingLive.ova`. The distribution is an x86 (32-bit) Ubuntu distribution and contains all the tools you will need to complete the assignment. The username and passowrd for the virtual machine are: `pac` and `badpass`, respectively.
3. Import the downloaded virtual machine image into Virtual Box program: `File` > `Import Appliance`, then browse to downloaded virtual machine image.
4. Start the *HackingLive* virtual machine and login using the previous credentials.
5. (*8 points*) Take a screenshot of the running virtual machine to show that you have been able to successfully setup the machine and login.
6. (*2 points*) Using the terminal, navigate to Desktop and create a new directory named `bof` to store the files you will create and edit through this assignment. Take a screenshot of the Desktop to show that you have successfully created the proper directories.

2 Running the Vulnerable Code (12 points)

1. Using the terminal, navigate to ∼`/Desktop/bof` directory.
2. Using `vi` to write the following vulnerable code to a new file named `vuln.c`:

```
#include <stdio.h>
int main(int argc, char **argv) {
  char buf[??];
  strcpy(buf, argv[1]);
}
```

   The `buf` size to work with can be computed using the formula: $(\texttt{sum}\%41) + 66$. Where `sum` is the sum of the ASCII values corresponding to the uppercase of the first letter for each group member's first name. For example, if your group has Alex and Gaby in it. Then, the $\texttt{sum} = 41 + 42 = 83$, and the `buf` size would be: $(83\%41) + 66 = 67$. Then, you would replace the `??` symbol with 67.

3. (*5 points*) After you compute the proper `buf` size. Show the content of `vuln.c` via the command: `cat vuln.c` and take a screenshot of the output.
4. (*5 points*) Compile the above code with debugger option (`-g`) and name the executable file `vuln.o`.
5. (*2 points*) Run the executable file `vuln.o` with `AAAAA` as an input and write your observations.
6. Take a screenshot of the terminal output for steps **4** and **5**.

3 Fining EIP and EBP Offsets (36 points)

1. (*10 points*) Craft an input of repeated `B` characters that is long enough to cause the `vuln.o` program to output `Illegal Instruction`. You can use any of the following perl scripts to accomplish the task.

   - `perl -e 'print "B"x20'` - prints 20 `B` characters to `stdout`.
   - `perl -e 'print "B"x20' > input` - prints 20 `B` characters to file `input`.
   - `./vuln.o $(perl -e 'print "B"x20')` - passed 20 `B` characters to program `vuln.o`.

   Take a screenshot of the terminal output for your trials.

2. (*10 points*) Explain what are the values you have tried and what is the relation between these values and the size of the shellcode we want to inject as well as the offsets of the stored `EBP` and `EIP` register values in the stack.

3. (*6 points*) Use the GNU debugger (`gdb`) to disassemble the `main` function in `vuln.o`. Then, note down the start address of the `main` function and the addresses of the `call strcpy` instruction and `ret` instruction. We will use these to set breakpoints for debugging later. Take a screenshot of the debugger output and mark those addresses on the screenshot.

4. (*10 points*) Use the GNU debugger to confirm that the offsets your guessed for the stored `EBP` and `EIP` registers are correct. To do so, set a breakpoint at the `ret` instruction of the `main` function, then run the program with a crafted long input (`run ‘cat long_input‘`) that would overwrite the `EBP` register with `BBBB` and the `EIP` register with `CCCC`. When the breakpoint is reached use `info registers` command to see the values associated with the stored `EBP`. Then, type `c` to continue execution, then notice the error displayed due to overwriting the `EIP` register. Take a screenshot and mark the values for the overwritten values for `EBP` and `EIP` registers.

4 Crafting the Shellcode (62 points)

1. (*2 points*) To prevent you missing with assembly code, we have drafed a sample shellcode for you located in ∼`/uscc/shellcode2.asm`. Copy the shellcode from ∼`/uscc/shellcode2.asm` to `bof` directory and name it `shellcode.asm`. Take a screenshot showing the output of the command `cat shellcode.asm` to confirm that you are able to copy it correctly and have the right file.

2. (*6 points*) For this assignment you need to modify the `shellcode.asm` to print the uppercase of the first letter of each group's member first name or the repeated first letter of your first name if you are alone. The number fo characters allowed in this step is 8 characters only. If you think you are brave to modify the code to print more than that, then do it to claim bonus points. For example, my group has Alex and Gaby, then the message to be printed should be changed to `AAAAGGGG` instead of `Owned!!!`. Remember that you are dealing with little-endian byte ordering. Take a screenshot of your changes by displaying the output of the command: `cat shellcode.asm`.

3. (*2 points*) Now compile the shellcode using: `nasm -f elf shellcode.asm`. Take a screenshot of the terminal output for this step.

4. (*6 points*) Inspect the compiled code using `objdump -M intel -d shellcode.o` if it has any `null` bytes. Take a screenshot of the `objdump` output. If any `null` bytes are found in the `shellcode.o`, try to resolve them and see what changes you have made in step **2**.

5. (*6 points*) Use perl scripts to store the binary content of the `shellcode.o` into a file named `payload` and use `wc` command to compute the number of bytes in the file so that we know what is the size of NOP sled to inject. Take a screenshot for the output of these commands.

6. (*6 points*) To check whether your crafted shellcode is well-formed and executable, we need to pass it as an input to a program that will be able to run it. This program is stored in ∼`/uscc/harness.c`. Copy `harness.c` into your `bof` directory and compile it. Then run the command `./harness ‘cat payload‘`. If the message you encoded at step **2** displayed correctly, then you were successful to modify the shellcode and ready to go to next step. Take a screenshot for the output of this step.

7. (*6 points*) Based on the number of bytes stored in the `payload`, compute the number of NOP (`x90`) operations to prefix the `payload` file with and use the proper perl (or python) script to append the proper number of NOP operations to the `payload` file and run the `wc` command to confirm the total size. Take a screenshot of the output of the executed commands on this step.

8. (*6 points*) Using `hexedit`, append the `payload` file with two placeholders for overwriting the stored `EBP` and `EIP` registers. You may use unique placeholder values such as `CCCC` for `EBP` and `DDDD` for `EIP`. Take a screenshot to show your changes. Note that `CTRL+x` will prompt you to save any changes within `hexedit`.

9. (*6 points*) Use the GNU debugger to verify that your shellcode overwrites the stored values for `EBP` and `EIP` registers with the corresponding placeholder values in the previous step. Take a screenshot to show that you have accomplished this task.

10. (*10 points*) Now, we need to find the proper address for overwriting the `EIP` register correctly somehow in the middle of the NOP sled. To discover the address, we can use the GNU debugger to debug the stack before and after the call to `strcpy` instruction. To do so, set a breakpoint at the `call strcpy` instruction and run the program with `payload` file's content. When the breakpoint is reached, dump 80 bytes from the current stack using `x/80bx $esp`. Then, type `next` to step after the call, inspect the stack again and pick an address somehow in the middle of the NOP sled. Take a screenshot marking the address you would like to consider for overwriting the `EIP` register value.

11. (*6 points*) Use `hexedit` to encode the chosen address value in the previous step instead the placeholder value to overwrite the `EIP` register value - note that the address should be stored in a little-endian byte ordering. Take a screenshot showing your changes.

5 Moment of Truth (10 points)

1. (*10 points*) Run the GNU debugger and run the program with `payload` file (`run ‘cat payload‘`) and see if it prints the encoded message successful. If it prints your encoded message, then take a screen shot and celebrate! You have successfully performed your first buffer overflow attack, happy hacking!