Intro to Return Oriented Programming

UTD CSG

James McFadyen

Outline

- ret2libc
- Return Chaining
- ROP Basics
- Gadgets
- Example

ret2libc

- If we point EIP to a function in libc, such as system(), we can pass it our own arguments
- EIP will point to address of system(), next 4 bytes return address, next 4 bytes will be arguments
- Example:
  - Overwrite EIP with address of system. Call this offset “x”
  - At offset x+4, (right after EIP), we have a return address
  - At x+8, we have the arguments. “/bin/sh” would make great arguments..
  - ./program $(python -c 'print "A" * 104 + address_of_system + return_address + payload ')
  - Above assumes ‘program’ to be vulnerable, and the necessary offset is 104
- system(“bin/sh”) would spawn a shell

Return Chaining

- This is very useful, can use it in conjunction with the ret2libc methodology to bypass more protection mechanisms, such as ASCII Armoring
- If we try the ret2libc technique on a binary with ASCII Armor, we see that there are null bytes in the address. Ex: 0x00167100
- To evade this, we must repeatedly return into the PLT, in a “chain” of instructions

Return Chaining

- The PLT (Procedure Linkage Table) and the GOT (Global Offset Table) are two important sections
- When a program calls a function, it calls a function “stub” in the PLT, which then jumps to an address listed in the GOT.
- On first call to the GOT from PLT, the address for the wanted function will be resolved by dynamic linker and patched into the GOT.
- Next time the PLT entry jumps to the GOT, it will jump to the actual address.

Return Chaining

- If we have a libc function that has null bytes, we can take advantage of the PLT and GOT to achieve the same goal as ret2libc
- char *strcpy(char *dest, const char *src);
- *dest will be address of GOT for a function
- *src will be the bytes
- We have to do this byte at a time…
- How do we do this?

Return Chaining

- Repeatedly call strcpy, write a single byte into GOT for a function that gets called in the program
  - Example: replace printf() with system()
- Since there are null bytes in system, we write one byte at a time of the 4 byte address , null bytes included
- This changes printf() to system(), so when we call printf() in the program, it actually calls system (since system() probably won’t already exist in the binary)
- So what does that look like?

Return Chaining

- Basic example: pseudo-payload to overwrite GOT of printf() with system():
  - strcpy() + pop pop ret + printf@GOT[0] + 1st byte of system() address
  - strcpy() + pop pop ret + printf@GOT[1] + 2nd byte of system() address
  - strcpy() + pop pop ret + printf@GOT[2] + 3rd byte of system() address
  - strcpy() + pop pop ret + printf@GOT[3] + 4th byte of system() address

- Once this is accomplished, carry out ret2libc like normal, but instead of executing system(), we point to printf(), since it is overwritten

Return Chaining

- pop pop ret is an important gadget (Explained later)
- In an actual payload, it will be a memory address pointing to those instructions
- The next 4 bytes after strcpy() are the return address
- Since the return address has pop pop ret, it will execute those instructions, moving past the arguments to strcpy(): dst and src.

ROP Basics

- Return Oriented Programming
- This is not an introduction to x86 assembly
- Uses code that is already in the program’s address space
- No injection of a payload necessary
- Evades DEP / NX
- Many techniques exist to bypass even more protection mechanisms
- Based on return to library attacks, such as ret2libc

ROP Basics

- Evolution:
  - Ret2libc, “Borrowed Code Chunks”, ROP
- Extends beyond scope of ret2lib techniques by allowing the use of “gadgets”
- Allows loops and conditional branching
- Take control of stack
  - Especially interested in $esp / $rsp
- We will rely on the stack pointer rather than instruction pointer for ROP
- Take sequences of instructions that already exist in the code to control the program

ROP Basics

- Usefool tools (Linux):
  - Scripting language of choice
  - gdb
  - objdump
  - readelf
  - ROPGadget http://shell-storm.org/project/ROPgadget/
  - ROPEme

ESP vs. EIP

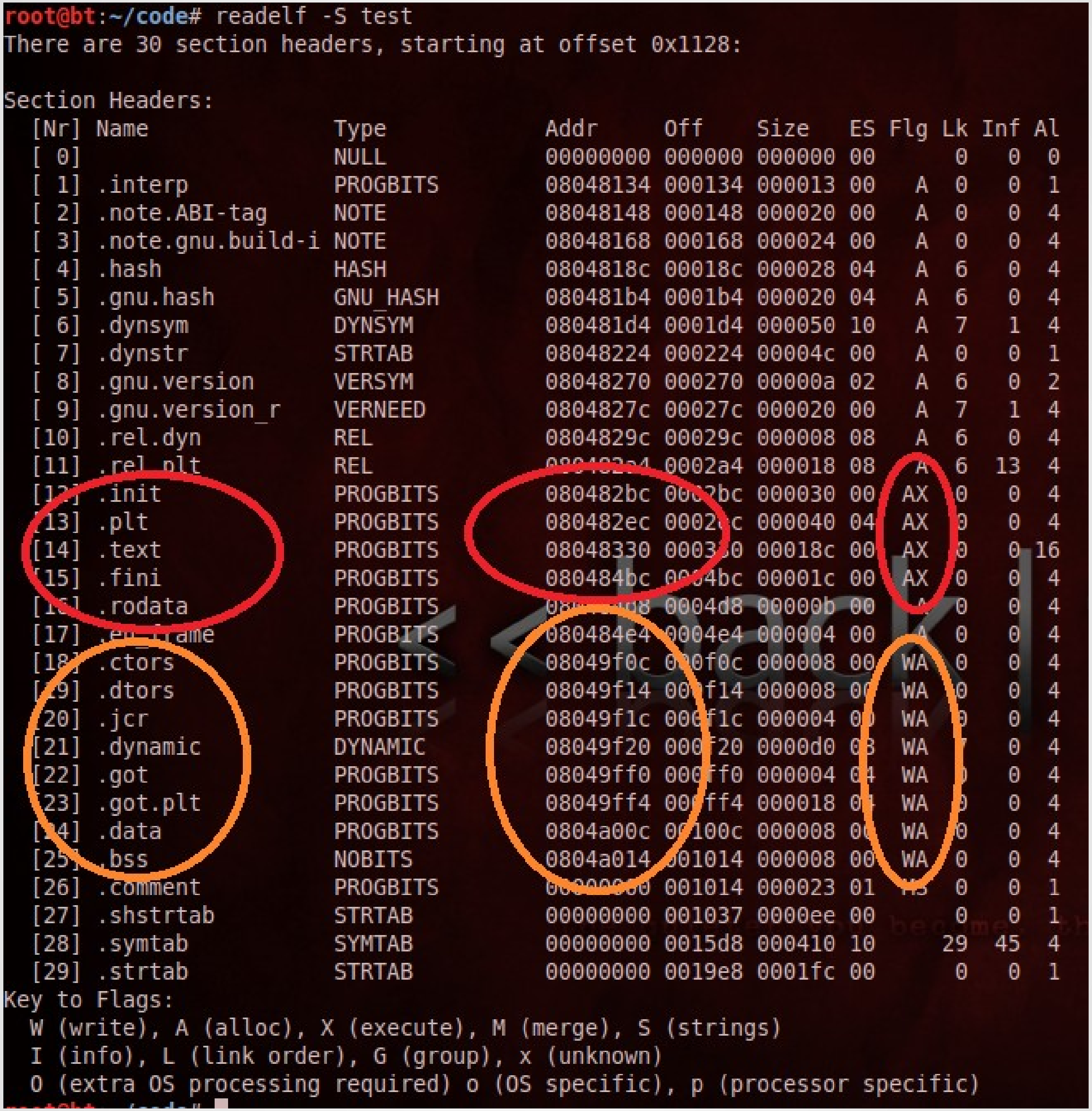
- EIP points to the current instruction to execute
- Processor automatically increments EIP upon execution
- ESP does not get incremented by the processor
- “ret” increments ESP
  - Note: not limited to ret for stack traversal

Gadgets

- Different instruction sequences ending in “ret”
- They perform specific tasks, such as moving the stack pointer or writing a value
- Ex: pop eax; ret
  - Load address at stack pointer into eax
- Ex: pop eax; pop ebx; ret
  - Load two consecutive words into eax and ebx.
  - Also good for “stepping over” instructions and incrementing the stack pointer
- Need to understand assembly, these can get very complicated when dealing with logic and control flow

Gadgets

- We can use gadgets to pivot the stack into an area that we control
- Ex: mov esp, ebx; ret
- Strings of these gadgets form chains of instructions
- Gadgets placed in specific orders can execute specific tasks
- Only requirement is a sequence of useable bytes somewhere in executable memory region



```
root@bt:~/code# readelf -S test
There are 30 section headers, starting at offset 0x1128:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        08048134 000134 000013 00   A  0   0  1
  [ 2] .note.ABI-tag     NOTE            08048148 000148 000020 00   A  0   0  4
  [ 3] .note.gnu.build-i NOTE            08048168 000168 000024 00   A  0   0  4
  [ 4] .hash             HASH            0804818c 00018c 000028 04   A  6   0  4
  [ 5] .gnu.hash         GNU_HASH        080481b4 0001b4 000020 04   A  6   0  4
  [ 6] .dynsym           DYNSYM          080481d4 0001d4 000050 10   A  7   1  4
  [ 7] .dynstr           STRTAB          08048224 000224 00004c 00   A  0   0  1
  [ 8] .gnu.version      VERSYM          08048270 000270 00000a 02   A  6   0  2
  [ 9] .gnu.version_r    VERNEED         0804827c 00027c 000020 00   A  7   1  4
  [10] .rel.dyn          REL             0804829c 00029c 000008 08   A  6   0  4
  [11] .rel.plt          REL             080482a4 0002a4 000018 08   A  6  13  4
  [12] .init             PROGBITS        080482bc 0002bc 000030 00  AX  0   0  4
  [13] .plt              PROGBITS        080482ec 0002ec 000040 04  AX  0   0  4
  [14] .text             PROGBITS        08048330 000330 00018c 00  AX  0   0 16
  [15] .fini             PROGBITS        080484bc 0004bc 00001c 00  AX  0   0  4
  [16] .rodata           PROGBITS        080484d8 0004d8 00000b 00   A  0   0  4
  [17] .eh_frame         PROGBITS        080484e4 0004e4 000004 00   A  0   0  4
  [18] .ctors            PROGBITS        08049f0c 000f0c 000008 00  WA  0   0  4
  [19] .dtors            PROGBITS        08049f14 000f14 000008 00  WA  0   0  4
  [20] .jcr              PROGBITS        08049f1c 000f1c 000004 00  WA  0   0  4
  [21] .dynamic          DYNAMIC         08049f20 000f20 0000d0 08  WA  7   0  4
  [22] .got              PROGBITS        08049ff0 000ff0 000004 04  WA  0   0  4
  [23] .got.plt          PROGBITS        08049ff4 000ff4 000018 04  WA  0   0  4
  [24] .data             PROGBITS        0804a00c 00100c 000008 00  WA  0   0  4
  [25] .bss              NOBITS          0804a014 001014 000008 00  WA  0   0  4
  [26] .comment          PROGBITS        00000000 001014 000023 01  MS  0   0  1
  [27] .shstrtab         STRTAB          00000000 001037 0000ee 00      0   0  1
  [28] .symtab           SYMTAB          00000000 0015d8 000410 10     29  45  4
  [29] .strtab           STRTAB          00000000 0019e8 0001fc 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
root@bt:~/code#
```

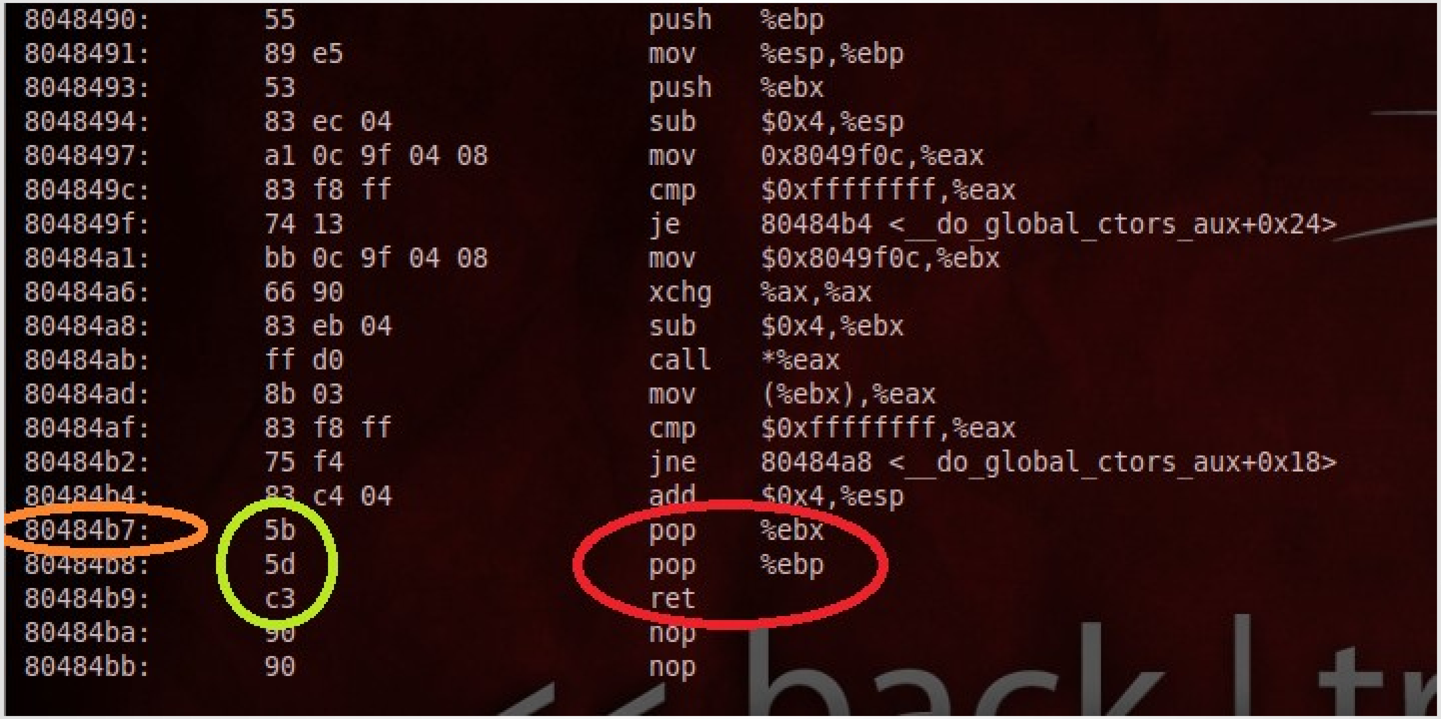
- (note red: executable, orange: writeable)

Gadgets

- “readelf –s binary” – displays section headers for binary
- Areas such as .bss are writeable - this allows us to throw payloads here, create custom stacks, etc…
- .got is also an important place to write, since we can manipulate a program by changing the functions
  - Ex: replace a printf() with exec() in a binary that does not contain exec()

Gadgets

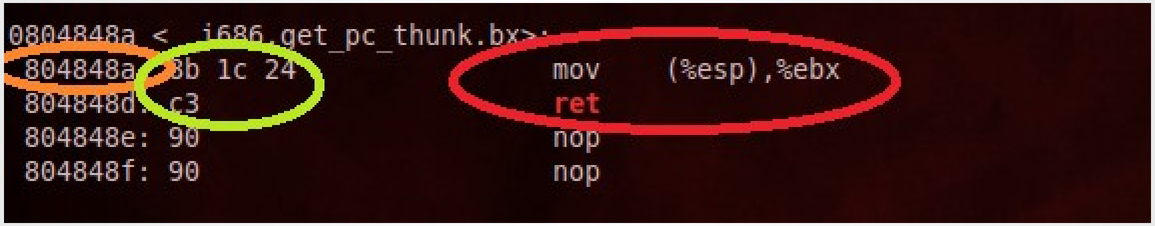
- Sample objdump output from a Linux binary



```
8048490:       55                      push   %ebp
8048491:       89 e5                   mov    %esp,%ebp
8048493:       53                      push   %ebx
8048494:       83 ec 04                sub    $0x4,%esp
8048497:       a1 0c 9f 04 08          mov    0x8049f0c,%eax
804849c:       83 f8 ff                cmp    $0xffffffff,%eax
804849f:       74 13                   je     80484b4 <__do_global_ctors_aux+0x24>
80484a1:       bb 0c 9f 04 08          mov    $0x8049f0c,%ebx
80484a6:       66 90                   xchg   %ax,%ax
80484a8:       83 eb 04                sub    $0x4,%ebx
80484ab:       ff d0                   call   *%eax
80484ad:       8b 03                   mov    (%ebx),%eax
80484af:       83 f8 ff                cmp    $0xffffffff,%eax
80484b2:       75 f4                   jne    80484a8 <__do_global_ctors_aux+0x18>
80484b4:       83 c4 04                add    $0x4,%esp
80484b7:       5b                      pop    %ebx
80484b8:       5d                      pop    %ebp
80484b9:       c3                      ret
80484ba:       90                      nop
80484bb:       90                      nop
```

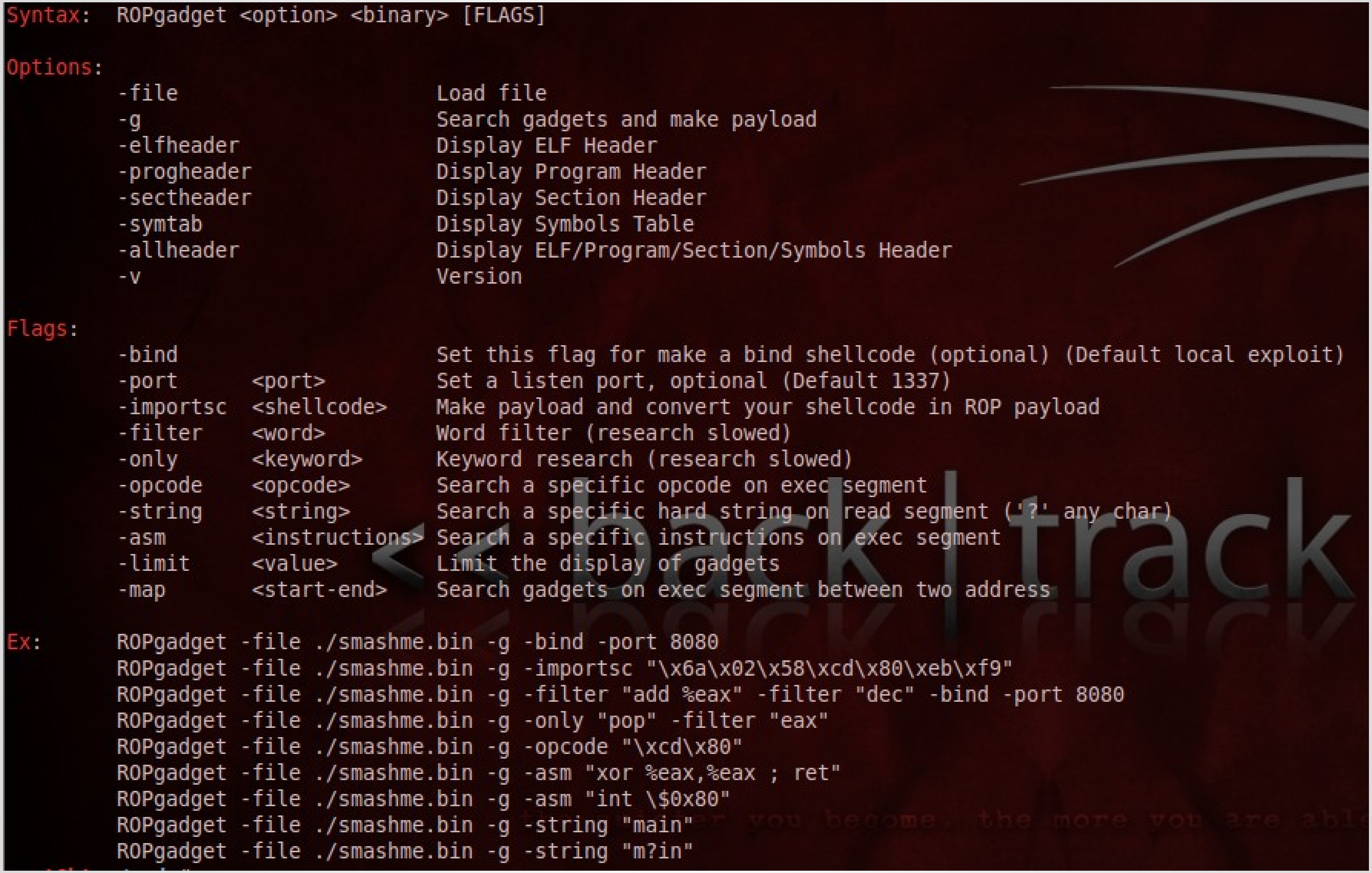
Gadgets

- Orange: memory location
- Green: opcodes
- Red: instructions
- You can see the gadgets can be obtained from within the same binary



```
0804848a <__i686.get_pc_thunk.bx>:
 804848a:  8b 1c 24             mov    (%esp),%ebx
 804848d:  c3                   ret
 804848e:  90                   nop
 804848f:  90                   nop
```

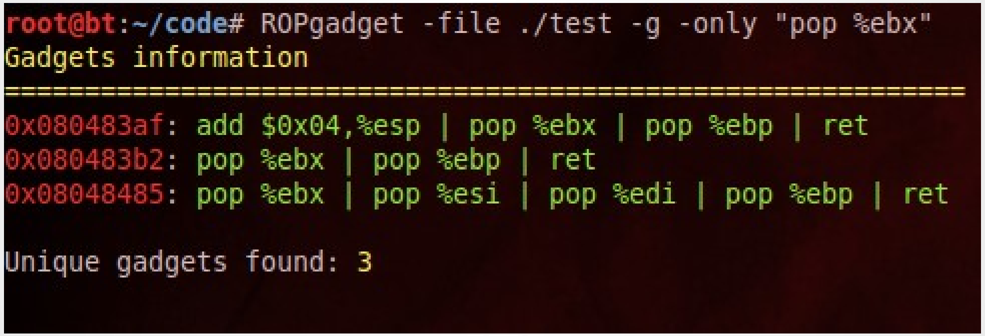
- But do we really want to dig through objdump output?



```
Syntax:  ROPgadget <option> <binary> [FLAGS]

Options:
         -file                    Load file
         -g                       Search gadgets and make payload
         -elfheader               Display ELF Header
         -progheader              Display Program Header
         -sectheader              Display Section Header
         -symtab                  Display Symbols Table
         -allheader               Display ELF/Program/Section/Symbols Header
         -v                       Version

Flags:
         -bind                    Set this flag for make a bind shellcode (optional) (Default local exploit)
         -port      <port>        Set a listen port, optional (Default 1337)
         -importsc  <shellcode>   Make payload and convert your shellcode in ROP payload
         -filter    <word>        Word filter (research slowed)
         -only      <keyword>     Keyword research (research slowed)
         -opcode    <opcode>      Search a specific opcode on exec segment
         -string    <string>      Search a specific hard string on read segment ('?' any char)
         -asm       <instructions> Search a specific instructions on exec segment
         -limit     <value>       Limit the display of gadgets
         -map       <start-end>   Search gadgets on exec segment between two address

Ex:      ROPgadget -file ./smashme.bin -g -bind -port 8080
         ROPgadget -file ./smashme.bin -g -importsc "\x6a\x02\x58\xcd\x80\xeb\xf9"
         ROPgadget -file ./smashme.bin -g -filter "add %eax" -filter "dec" -bind -port 8080
         ROPgadget -file ./smashme.bin -g -only "pop" -filter "eax"
         ROPgadget -file ./smashme.bin -g -opcode "\xcd\x80"
         ROPgadget -file ./smashme.bin -g -asm "xor %eax,%eax ; ret"
         ROPgadget -file ./smashme.bin -g -asm "int \$0x80"
         ROPgadget -file ./smashme.bin -g -string "main"
         ROPgadget -file ./smashme.bin -g -string "m?in"
```

Gadgets

- Example: Assume we control the stack, and we need to place a value from our stack into EBX..



```
root@bt:~/code# ROPgadget -file ./test -g -only "pop %ebx"
Gadgets information
============================================================
0x080483af: add $0x04,%esp | pop %ebx | pop %ebp | ret
0x080483b2: pop %ebx | pop %ebp | ret
0x08048485: pop %ebx | pop %esi | pop %edi | pop %ebp | ret

Unique gadgets found: 3
```

- If we execute memory address 0x080483b2, it will pop our value from the stack pointer, store it in EBX, which increments the stack pointer, pop the next value into EBP, which increment thes stack pointer, and return, which increments the stack pointer.

Gadgets

- If you can solve a problem in assembly, all you need to do is find gadgets that will accomplish your goal
- Looping, conditions, etc...

Demo

- Some ROP basics
- Return Chaining
- ROPGadget

CTF Writeups

- http://www.vnsecurity.net/2011/01/padocon-2011-ctf-karma-400-exploit-the-data-re-use-way/
- http://leetmore.ctf.su/wp/defcon-ctf-quals-2011-pwnables-400/
- http://www.vnsecurity.net/2011/10/hack-lu-ctf-2011-nebula-death-stick-services-writeup/
- http://www.vnsecurity.net/2010/04/return-oriented-programming-practice-exploiting-codegate-2010-challenge-5/
- http://auntitled.blogspot.com/2011/03/codegate-c
- tf-2011-vuln300-writeup.html

Additional Resources

- http://www.youtube.com/watch?v=rVhOnqlfIvQ
- http://www.phrack.com/issues.html?issue=58&id=4
  http://isisblogs.poly.edu/2011/10/21/geras-insecure-programming-warming-up-stack-1-rop-nxaslr-bypass/
- http://falken.tuxfamily.org/?p=115
- http://cseweb.ucsd.edu/~hovav/papers/rbss11.html
- http://divine-protection.com/wordpress/?p=20
- http://www.corelan.be/index.php/2010/06/16/exploit-writing-tutorial-part-10-chaining-dep-with-rop-the-rubikstm-cube/
-

References

- A Gentle Introduction to Return-Oriented Programming by Tim Kornau - http://blog.zynamics.com/2010/03/12/a-gentle-introduction-to-return-oriented-programming/
- PLT and GOT – The key to code sharing and dynamic libraries - http://www.technovelty.org/linux/pltgot.html
- "Return-oriented Programming: Exploits Without Code Injection" Erik Buchanan, Ryan Roemer… http://cseweb.ucsd.edu/~hovav/talks/blackhat08.html

References contd..

- “Payload Already Inside: Data Reuse for ROP Exploits” Blackhat 2010, longld @ vnsecurity.net - http://media.blackhat.com/bh-us-10/whitepapers/Le/BlackHat-USA-2010-Le-Paper-Payload-already-inside-data-reuse-for-ROP-exploits-wp.pdf
- “Practical Return-Oriented Programming” Dino A. Dai Zovi - http://trailofbits.files.wordpress.com/2010/04/practical-rop.pdf