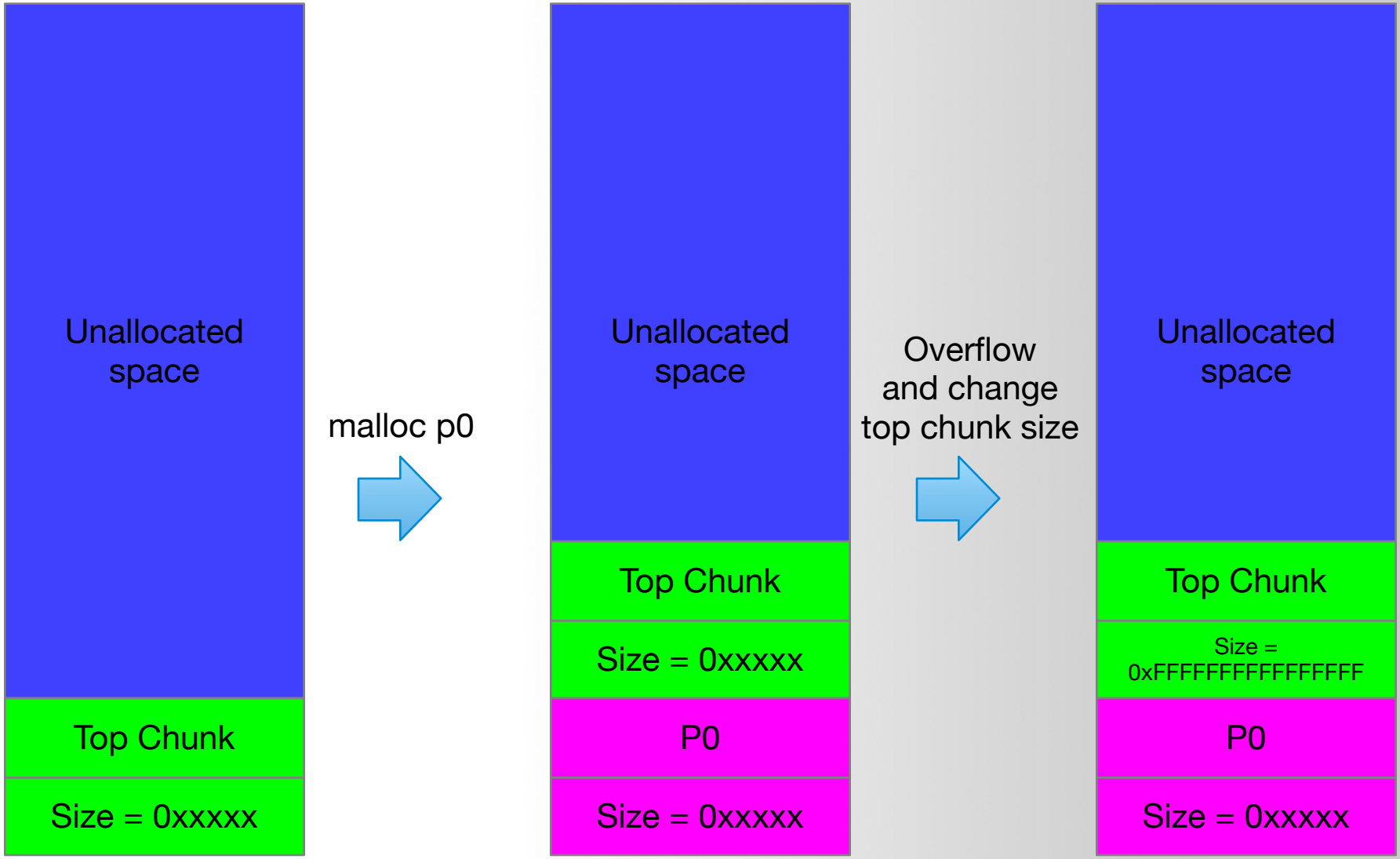
Class 18

CSC 472/583 Topics of Software Security
Heap Exploitation (4): Use After Free (UAF), Double Free
Kernel Exploitation

Dr. Si Chen (schen@wcupa.edu)

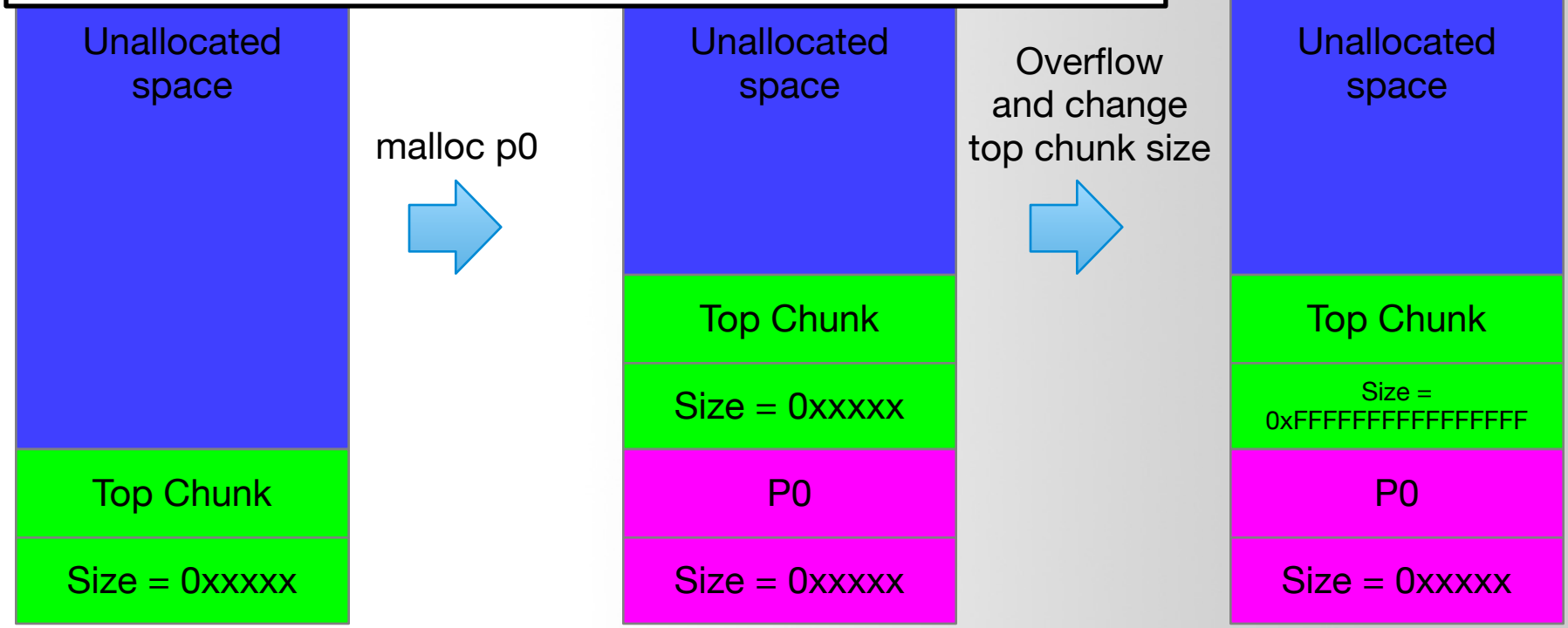
Review

House of Force



House of Force

- This attack assumes an overflow into the top chunk's header. The size is modified to a very large value (-1 in this example).
- This ensures that all initial requests will be services using the top chunk, instead of relying on mmap.
- On a 64 bit system, -1 evaluates to 0xFFFFFFFFFFFFFFFF.
- A chunk with this size can cover the entire memory space of the program.



House of Force

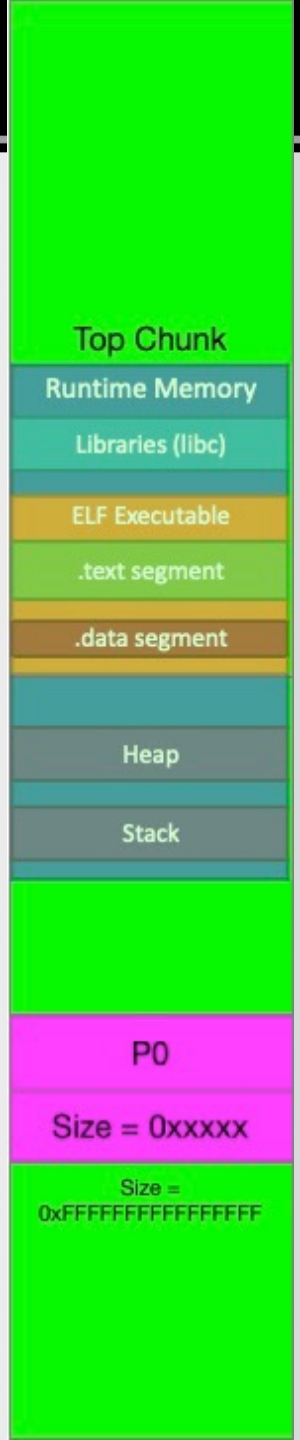
E.g. top_chunk=0x601200

malloc(0xffe00020)

0xffe00030 < top_chunk_size

0xffe00030+0x601200=0x100401230

top_chunk=0x401230

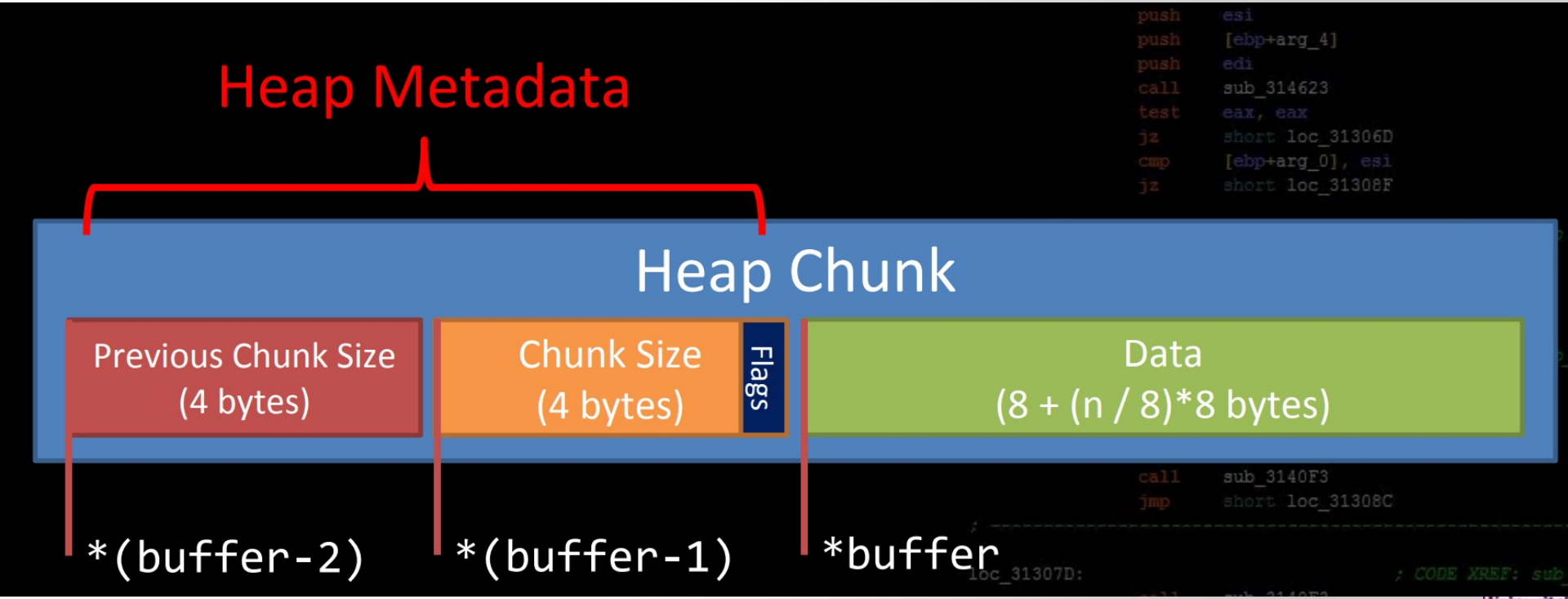


House of Force

- *Prerequisites*: Three malloc calls are required to successfully apply house of force as listed below:
  - Malloc 1: Attacker should be able to control the size of top chunk. Hence heap overflow should be possible on this allocated chunk which is physically located previous to top chunk.
  - Malloc 2: Attacker should be able to control the size of this malloc request.
  - Malloc 3: User input should be copied to this allocated chunk.

Metadata Corruption -- Unlink, House of Force

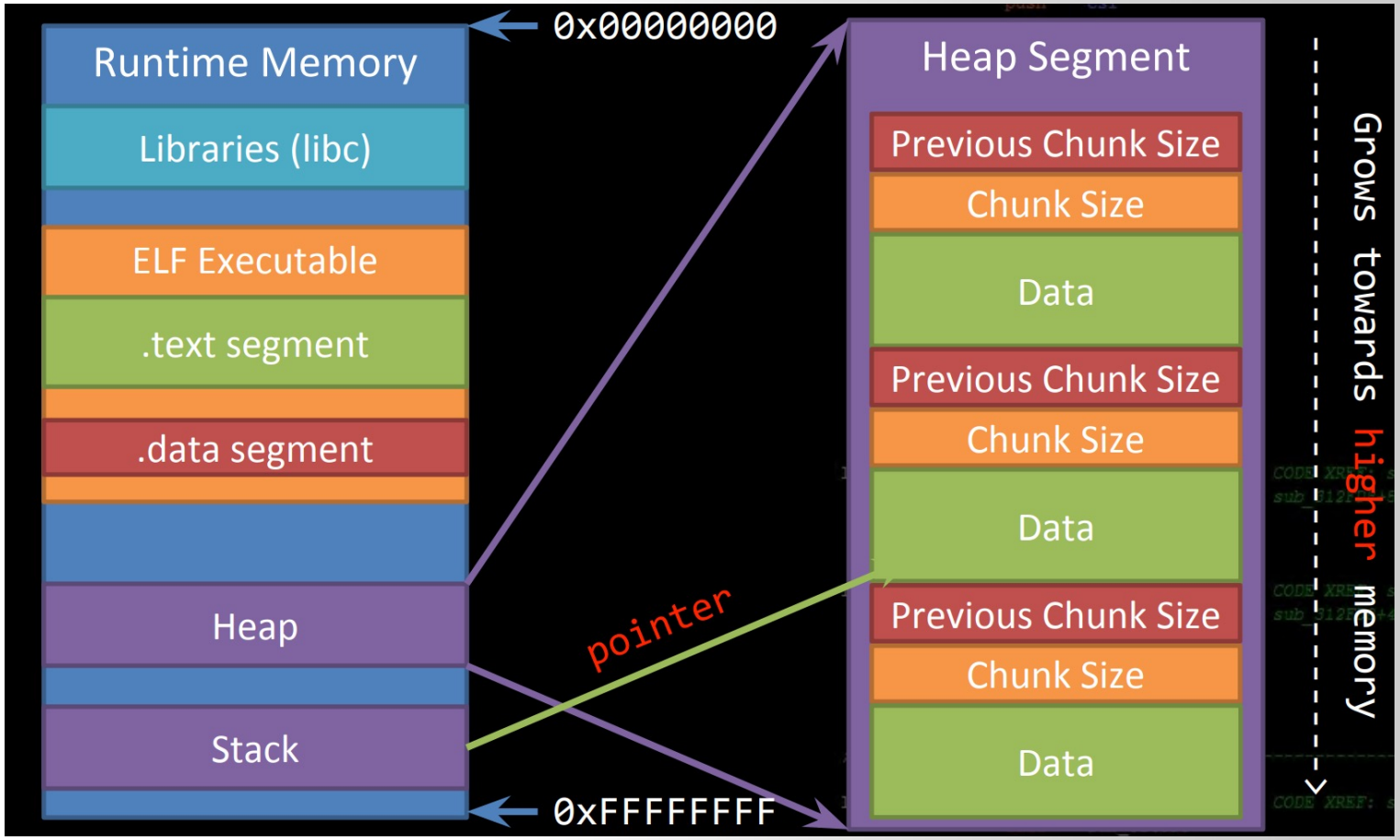
- Metadata corruption based exploits involve corrupting heap metadata in such a way that you can use the allocator's internal functions to cause a controlled write of some sort
- Generally involves faking chunks, and abusing its different coalescing or unlinking processes



Use After Free

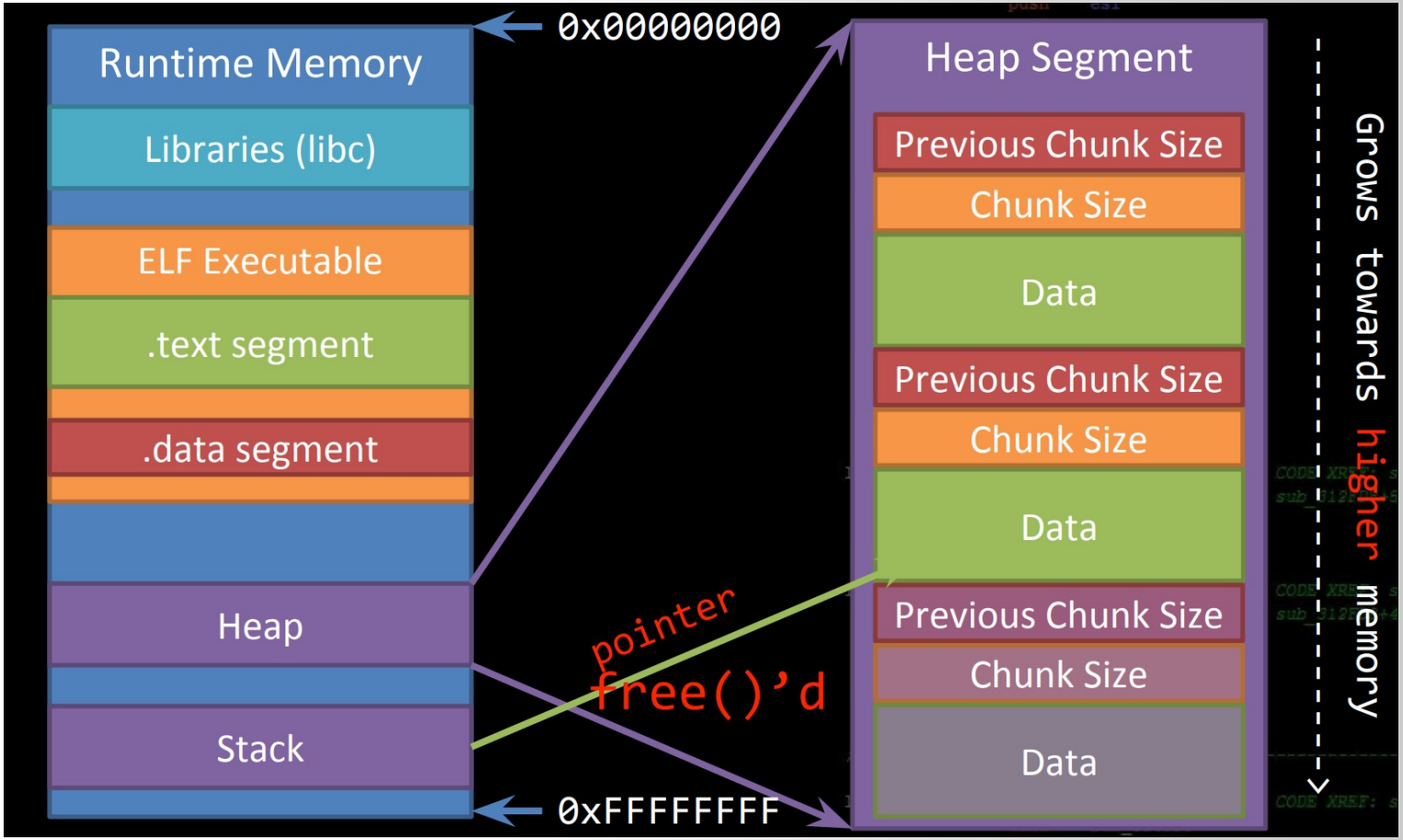
- **Use After Free**
  - A class of vulnerability where data on the heap is freed, but a leftover reference or '**dangling pointer**' is used by the code as if the data were still valid
  - Most popular in Web Browsers, complex programs
  - Also known as UAF

Use After Free



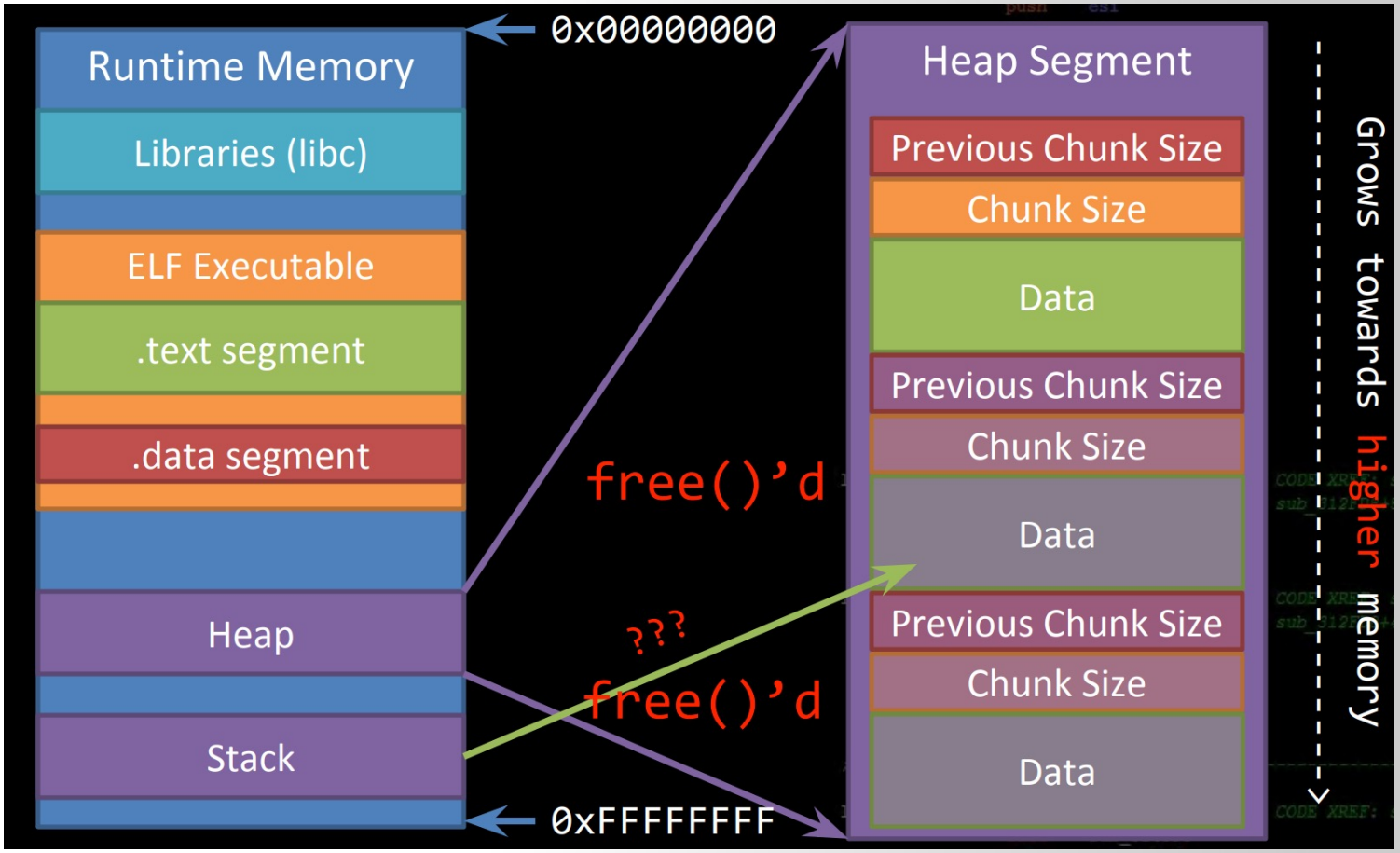
http://security.cs.rpi.edu/courses/binexp-spring2015/lectures/17/10_lecture.pdf

Use After Free



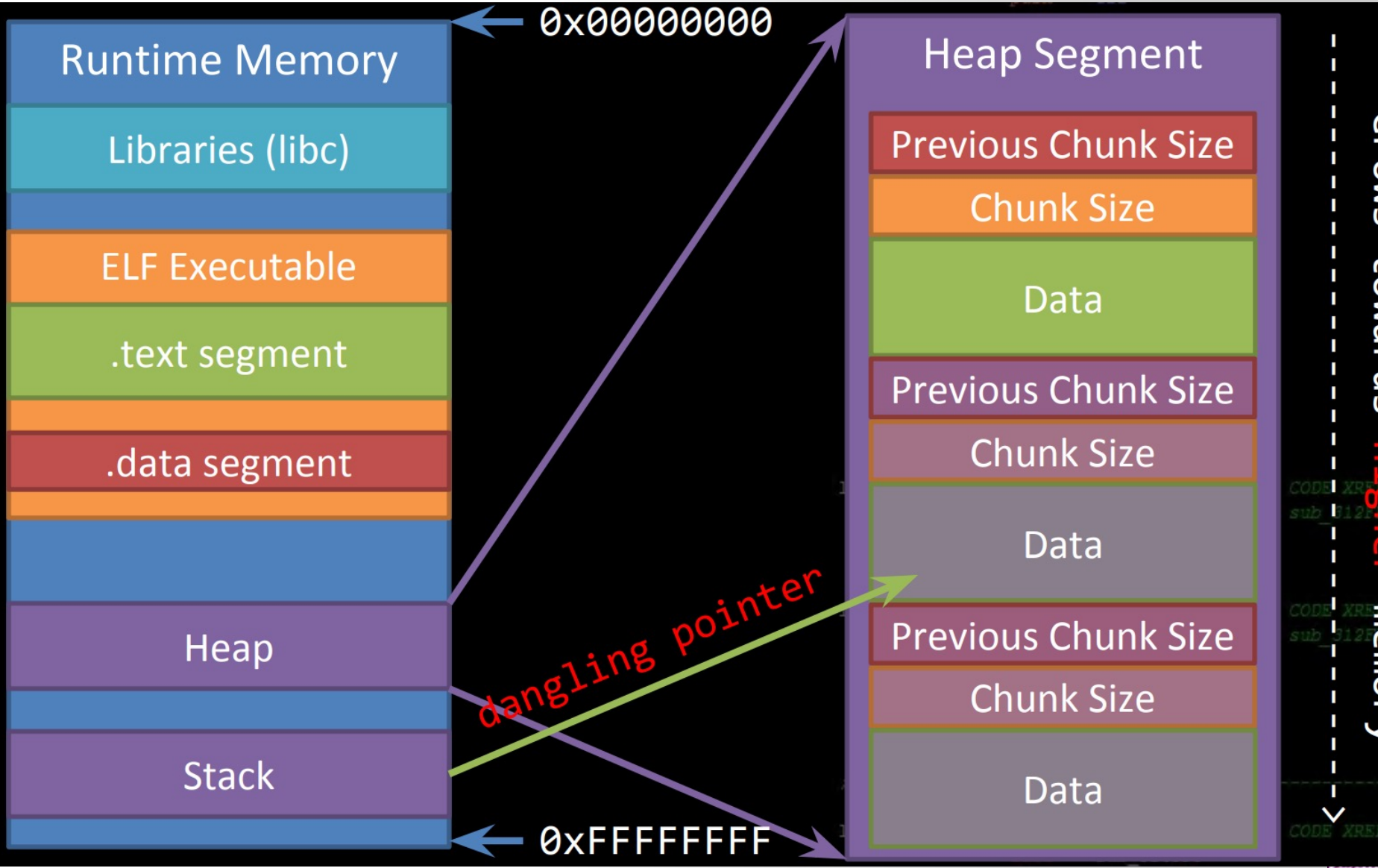
http://security.cs.rpi.edu/courses/binexp-spring2015/lectures/17/10_lecture.pdf

Use After Free



http://security.cs.rpi.edu/courses/binexp-spring2015/lectures/17/10_lecture.pdf

Use After Free



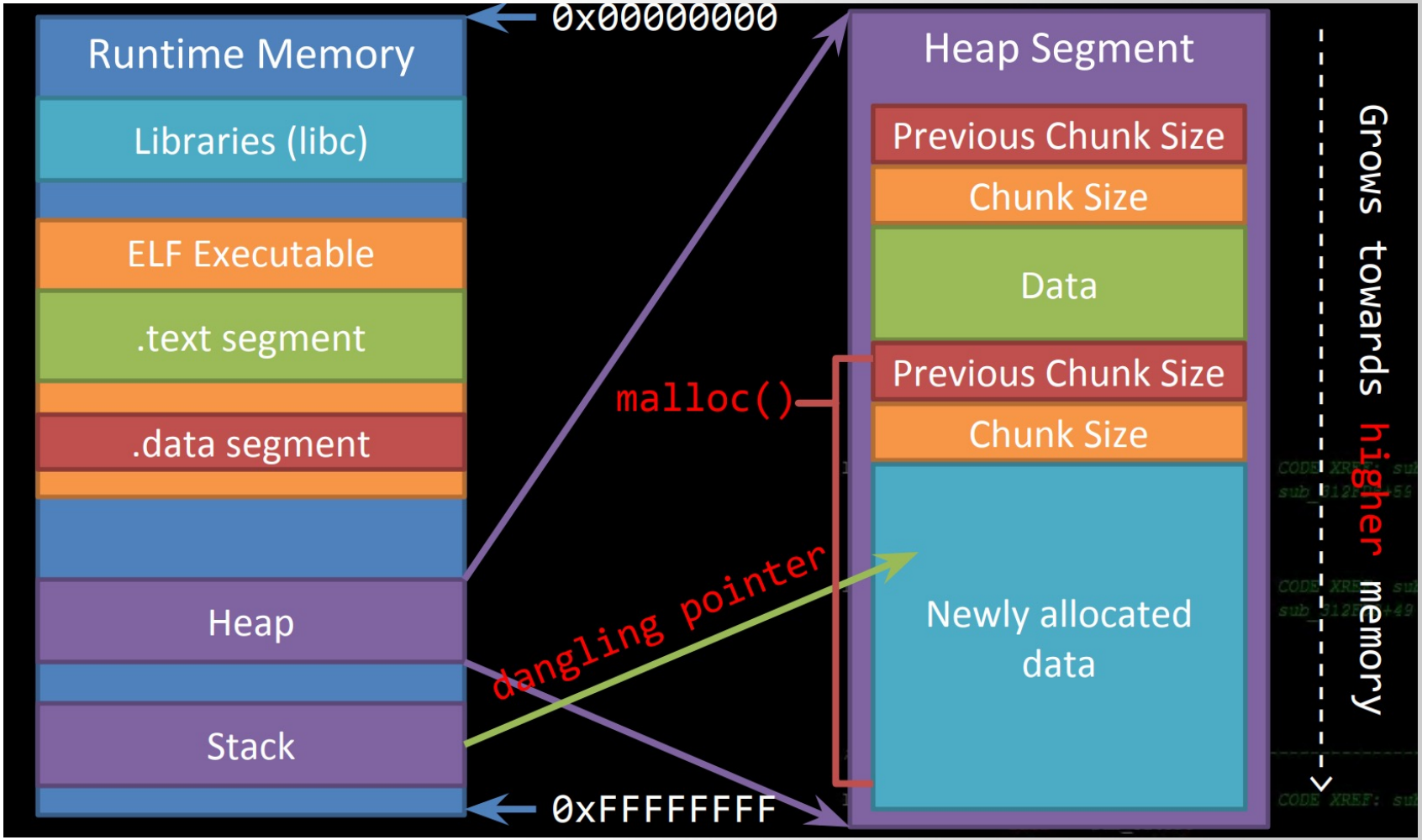
http://security.cs.rpi.edu/courses/binexp-spring2015/lectures/17/10_lecture.pdf

Use After Free

- **Dangling Pointer**
  - A left over pointer in your code that references free'd data and is prone to be re-used
  - As the memory it's pointing at was freed, there's no guarantees on what data is there now
  - Also known as **stale pointer, wild pointer**

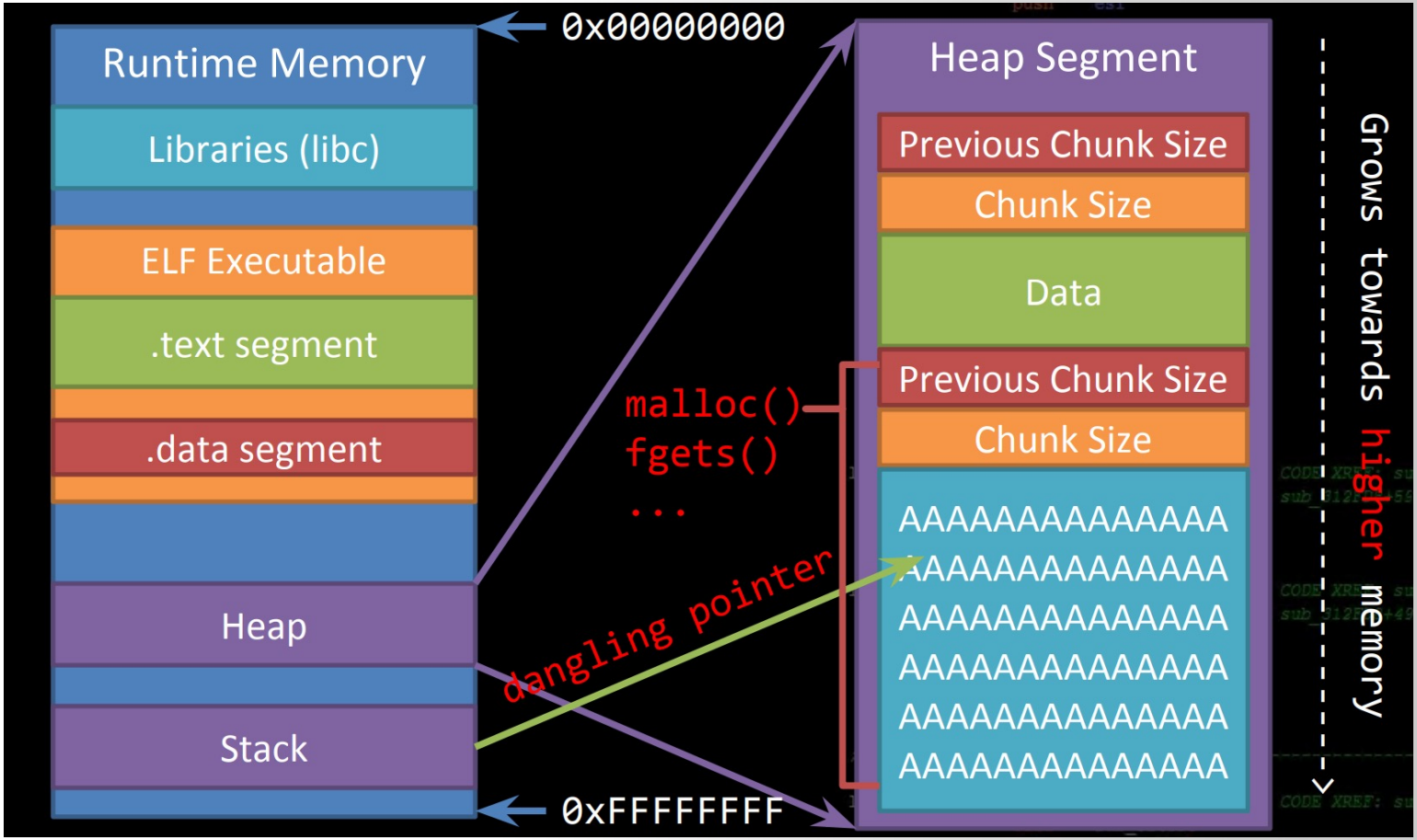
http://security.cs.rpi.edu/courses/binexp-spring2015/lectures/17/10_lecture.pdf

Use After Free



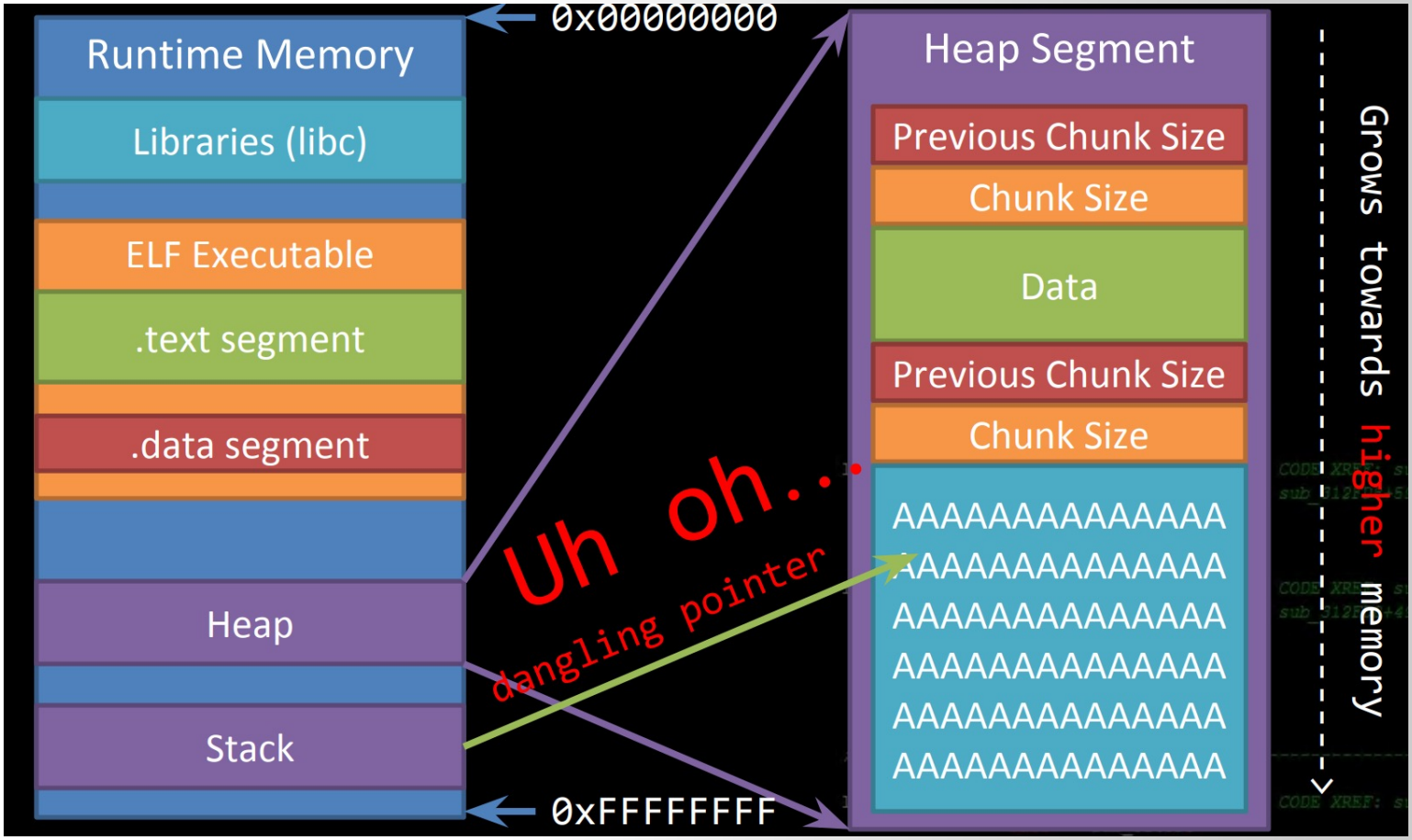
http://security.cs.rpi.edu/courses/binexp-spring2015/lectures/17/10_lecture.pdf

Use After Free



http://security.cs.rpi.edu/courses/binexp-spring2015/lectures/17/10_lecture.pdf

Use After Free



http://security.cs.rpi.edu/courses/binexp-spring2015/lectures/17/10_lecture.pdf

Use After Free

- You actually don't need any form of memory corruption to leverage a use after free
- It's simply an implementation issue
  - pointer mismanagement

http://security.cs.rpi.edu/courses/binexp-spring2015/lectures/17/10_lecture.pdf

Use After Free: PoC Example

```c
#include <stdio.h>

int main()
{
        char *p1;
        p1 = (char *)malloc(sizeof(char) * 10);
        memcpy(p1, "hello", 10);
        printf("P1 address:%x, %s\n", p1, p1);
        free(p1);
        char *p2;
        p2 = (char *)malloc(sizeof(char) * 10);
        memcpy(p2, "hello", 10);
        printf("P2 address:%x, %s\n", p2, p2);
        memcpy(p1, "hack!", 10);
        printf("P2 address:%x, %s\n", p2, p2);
        return 0;
}
```

Use After Free: PoC Example

```c
#include <stdio.h>

int main()
{
        char *p1;
        p1 = (char *)malloc(sizeof(char) * 10);
        memcpy(p1, "hello", 10);
        printf("P1 address:%x, %s\n", p1, p1);
        free(p1);
        char *p2;
        p2 = (char *)malloc(sizeof(char) * 10);
        memcpy(p2, "hello", 10);
        printf("P2 address:%x, %s\n", p2, p2);
        memcpy(p1, "hack!", 10);
        printf("P2 address:%x, %s\n", p2, p2);
        return 0;
}
```

```
→  heap ./uaf
P1 address:55756260, hello
P2 address:55756260, hello
P2 address:55756260, hack!
```

Use After Free

Search Results

There are **3263** CVE entries that match your search.

Name	Description
CVE-2019-9821	A use-after-free vulnerability can occur in AssertWorkerThread due to a race condition with shared workers. This results in a potentially exploitable crash. This vulnerability affects Firefox < 67.
CVE-2019-9820	A use-after-free vulnerability can occur in the chrome event handler when it is freed while still in use. This results in a potentially exploitable crash. This vulnerability affects Thunderbird < 60.7, Firefox < 67, and Firefox ESR < 60.7.
CVE-2019-9818	A race condition is present in the crash generation server used to generate data for the crash reporter. This issue can lead to a use-after-free in the main process, resulting in a potentially exploitable crash and a sandbox escape. *Note: this vulnerability only affects Windows. Other operating systems are unaffected.*. This vulnerability affects Thunderbird < 60.7, Firefox < 67, and Firefox ESR < 60.7.
CVE-2019-9796	A use-after-free vulnerability can occur when the SMIL animation controller incorrectly registers with the refresh driver twice when only a single registration is expected. When a registration is later freed with the removal of the animation controller element, the refresh driver incorrectly leaves a dangling pointer to the driver's observer array. This vulnerability affects Thunderbird < 60.6, Firefox ESR < 60.6, and Firefox < 66.
CVE-2019-9790	A use-after-free vulnerability can occur when a raw pointer to a DOM element on a page is obtained using JavaScript and the element is then removed while still in use. This results in a potentially exploitable crash. This vulnerability affects Thunderbird < 60.6, Firefox ESR < 60.6, and Firefox < 66.
CVE-2019-9767	Stack-based buffer overflow in Free MP3 CD Ripper 2.6, when converting a file, allows user-assisted remote attackers to execute arbitrary code via a crafted .wma file.
CVE-2019-9766	Stack-based buffer overflow in Free MP3 CD Ripper 2.6, when converting a file, allows user-assisted remote attackers to execute arbitrary code via a crafted .mp3 file.
CVE-2019-9706	Vixie Cron before the 3.0pl1-133 Debian package allows local users to cause a denial of service (use-after-free and daemon crash) because of a force_rescan_user error.
CVE-2019-9489	A directory traversal vulnerability in Trend Micro Apex One, OfficeScan (versions XG and 11.0), and Worry-Free Business Security (versions 10.0, 9.5 and 9.0) could allow an attacker to modify arbitrary files on the affected product's management console.
CVE-2019-9458	In the Android kernel in the video driver there is a use after free due to a race condition. This could lead to local escalation of privilege with no additional execution privileges needed. User interaction is not needed for exploitation.
CVE-2019-9447	In the Android kernel in the FingerTipS touchscreen driver there is a possible use-after-free due to improper locking. This could lead to a local escalation of privilege with System execution privileges needed. User interaction is not needed for exploitation.
CVE-2019-9442	In the Android kernel in the mnh driver there is possible memory corruption due to a use after free. This could lead to local escalation of privilege with System privileges required. User interaction is not needed for exploitation.
CVE-2019-9431	In Bluetooth, there is a possible out of bounds read due to a use after free. This could lead to remote information disclosure with heap information written to the log with System execution privileges needed. User interaction is not needed for exploitation. Product: AndroidVersions: Android-10Android ID: A-109755179
CVE-2019-9427	In Bluetooth, there is a possible information disclosure due to a use after free. This could lead to local information disclosure with no additional execution privileges needed. User interaction is not needed for exploitation. Product: AndroidVersions: Android-10Android ID: A-110166350
CVE-2019-9381	In netd, there is a possible out of bounds read due to a use after free. This could lead to remote information disclosure with no additional execution privileges needed. User interaction is not needed for exploitation. Product: AndroidVersions: Android-10Android ID: A-122677612
CVE-2019-9350	In Keymaster, there is a possible EoP due to a use after free. This could lead to local escalation of privilege with no additional execution privileges needed. User interaction is not needed for exploitation. Product: AndroidVersions: Android-10Android ID: A-129562815

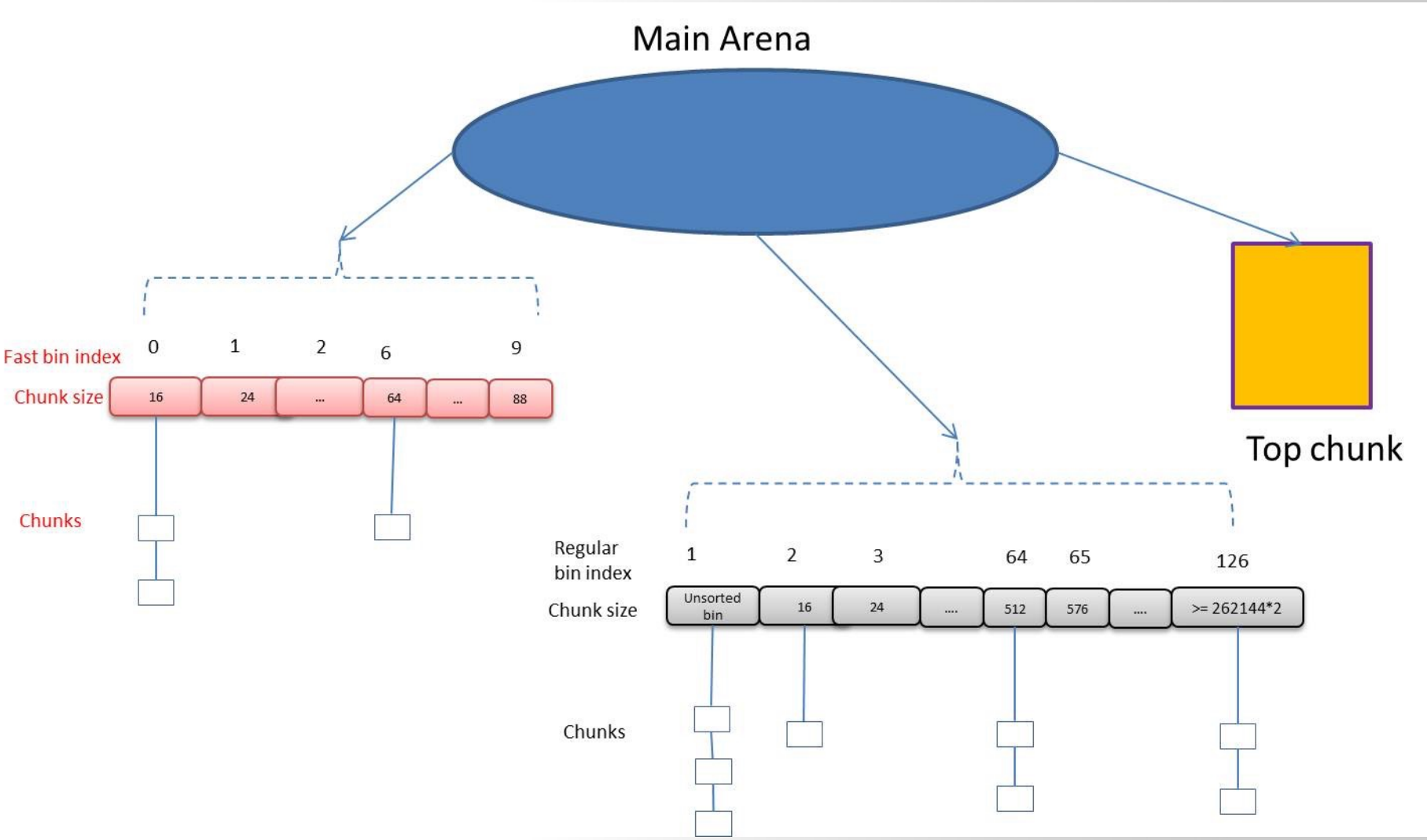
The ‘hot’ vulnerability nowadays, almost every modern browser exploit leverages a UAF

Use After Free

- From the defensive perspective, trying to detect use after free vulnerabilities in complex applications is **very difficult**, even in industry
- Why?
  - UAF's **only exist in certain states of execution**, so statically scanning source for them won't go far
  - They're **usually only found through crashes**, but symbolic execution and constraint solvers are helping find these bugs faster

Double Free

- **Double Free**
  - Freeing a resource more than once can lead to memory leaks.
  - The allocator's data structures get corrupted and can be exploited by an attacker.


Main Arena
Top chunk
Fast bin index
0
1
2
6
9
Chunk size
16
24
...
64
...
88
Chunks
Regular
bin index
1
2
3
64
65
126
Chunk size
Unsorted
bin
16
24
....
512
576
....
>= 262144*2
Chunks

fastbin_dup.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

int main()
{
        setbuf(stdout, NULL);

        printf("This file demonstrates a simple double-free attack with fastbins.\n");

        printf("Fill up tcache first.\n");
        void *ptrs[8];
        for (int i=0; i<8; i++) {
                ptrs[i] = malloc(8);
        }
        for (int i=0; i<7; i++) {
                free(ptrs[i]);
        }

        printf("Allocating 3 buffers.\n");
        int *a = calloc(1, 8);
        int *b = calloc(1, 8);
        int *c = calloc(1, 8);

        printf("1st calloc(1, 8): %p\n", a);
        printf("2nd calloc(1, 8): %p\n", b);
        printf("3rd calloc(1, 8): %p\n", c);

        printf("Freeing the first one...\n");
        free(a);

        printf("If we free %p again, things will crash because %p is at the top of the free list.\n", a, a);
        // free(a);

        printf("So, instead, we'll free %p.\n", b);
        free(b);

        printf("Now, we can free %p again, since it's not the head of the free list.\n", a);
        free(a);

        printf("Now the free list has [ %p, %p, %p ]. If we malloc 3 times, we'll get %p twice!\n", a, b, a, a);
        a = calloc(1, 8);
        b = calloc(1, 8);
        c = calloc(1, 8);
        printf("1st calloc(1, 8): %p\n", a);
        printf("2nd calloc(1, 8): %p\n", b);
        printf("3rd calloc(1, 8): %p\n", c);

        assert(a == c);
}
```

From how2heap
https://github.com/shellphish/how2heap/blob/master/glibc_2.34/fastbin_dup.c

Introduction

So far, we have been exploiting binaries running in userspace.

Userspace is an *abstraction* that runs “on top” of the kernel.

1. Filesystem I/O
2. Privilege Levels (Per User/Per Group)
3. Syscalls
4. Processes
5. And so much more

Introduction

So far, we have been exploiting binaries running in userspace.

Userspace is an *abstraction* that runs "on top" of the kernel.

1. Filesystem I/O
2. Privilege Levels (Per User/Per Group)
3. Syscalls
4. Processes
5. And so much more

These are all "services" provided by the Kernel

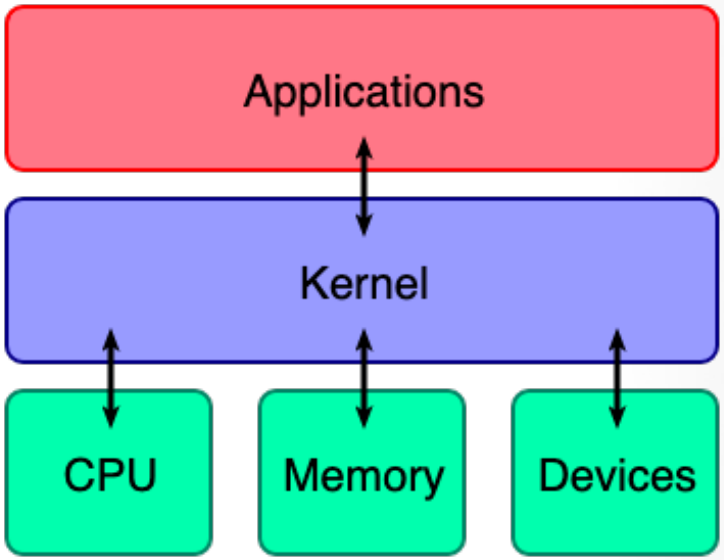
What's a Kernel?

Low Level code with two major responsibilities

1. Interact with and control hardware components
2. Provide an Environment in which Applications can run

The Kernel is the core of the operating system

Introduction



The kernel is also a **program** that:

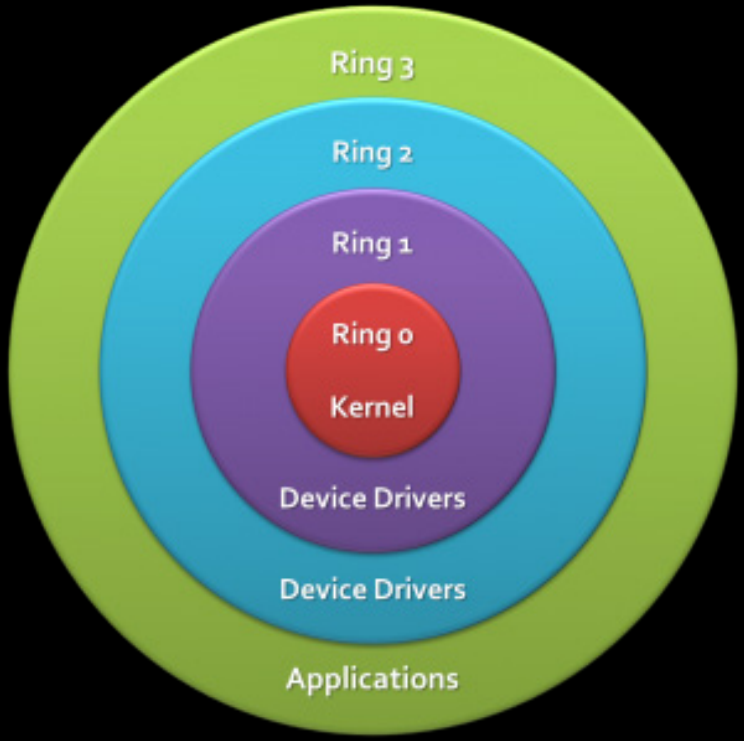
- Manages the data I/O requirements issued by the software
- Escaping these requirements into instructions
- Handing them over to the CPU

Ring Model

Hardware Enforced Model

0: Privileged, Kernelspace

3: Restricted, Userspace



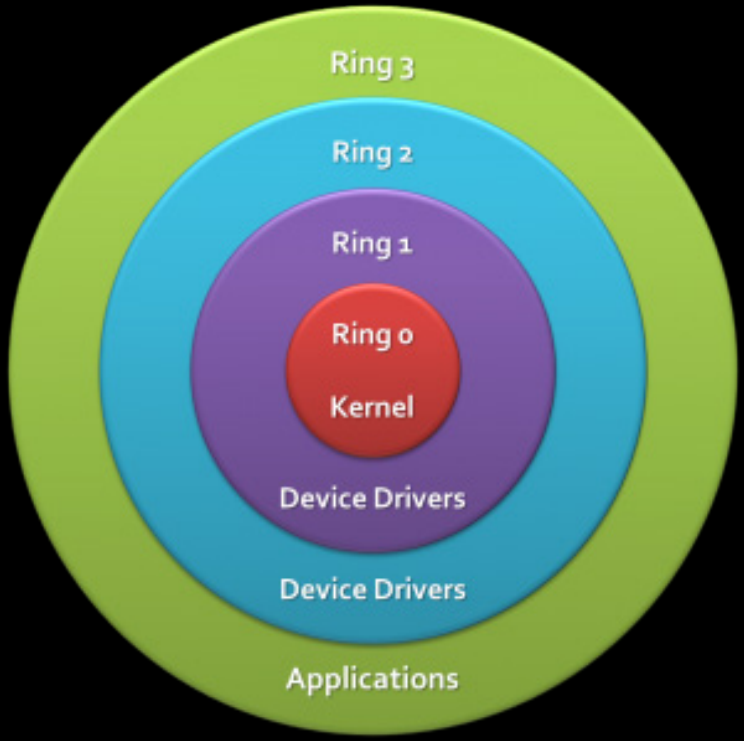
Ring Model

Hardware Enforced Model

0: Privileged, Kernelspace

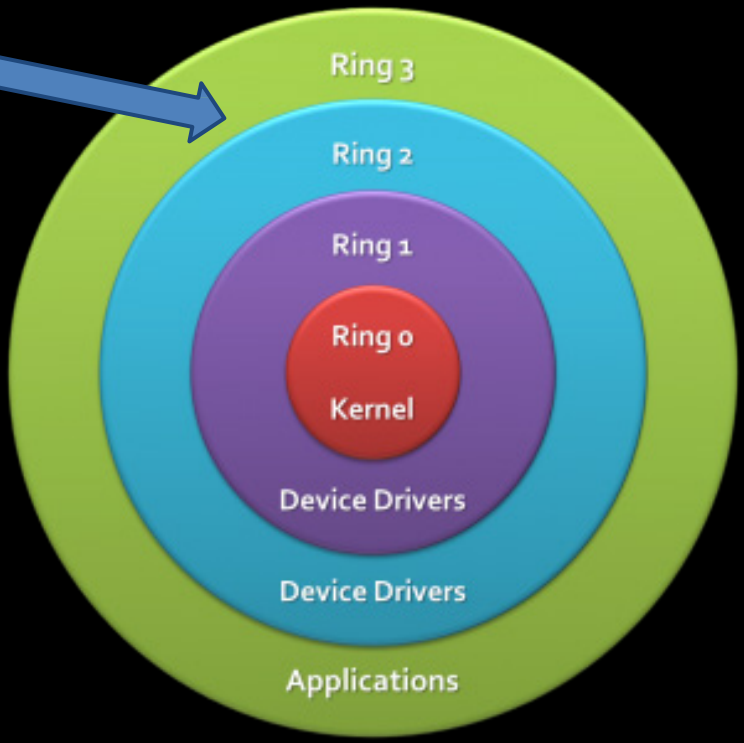
3: Restricted, Userspace

Ring 1 and Ring 2 are not utilized by most popular/modern Operating Systems (Linux / Windows / OSX)



Ring Model

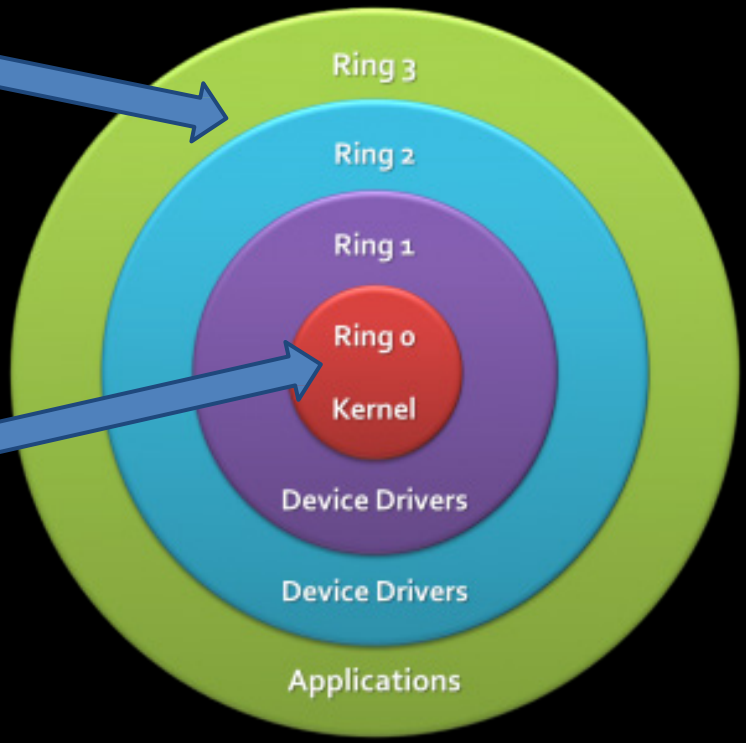
We've Been Here



Ring Model

We've Been Here

We're Going Here

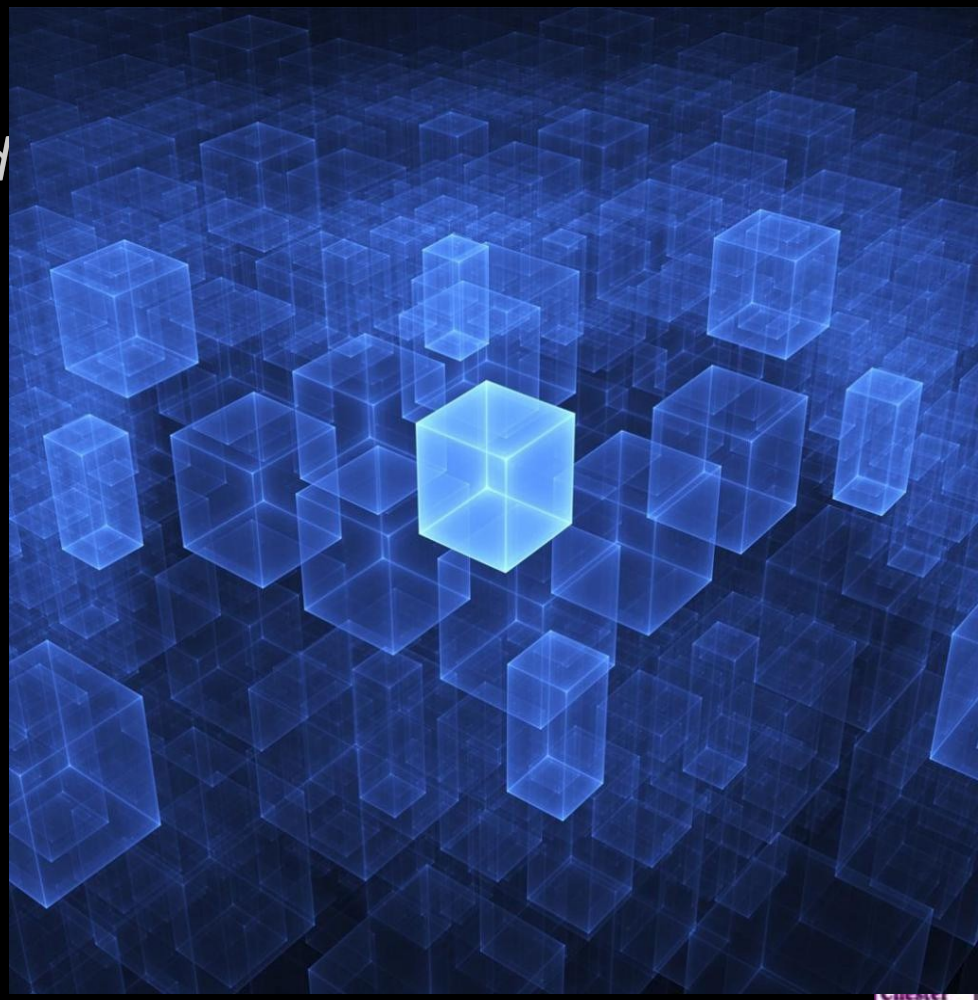


“Matrix”

“The Matrix is the world that has been pulled over your eyes to blind you from the truth.” - Morpheus

The kernel provides the “matrix” your programs run in

Break out of the Matrix, and you pwn the entire system



Kernel Pwning

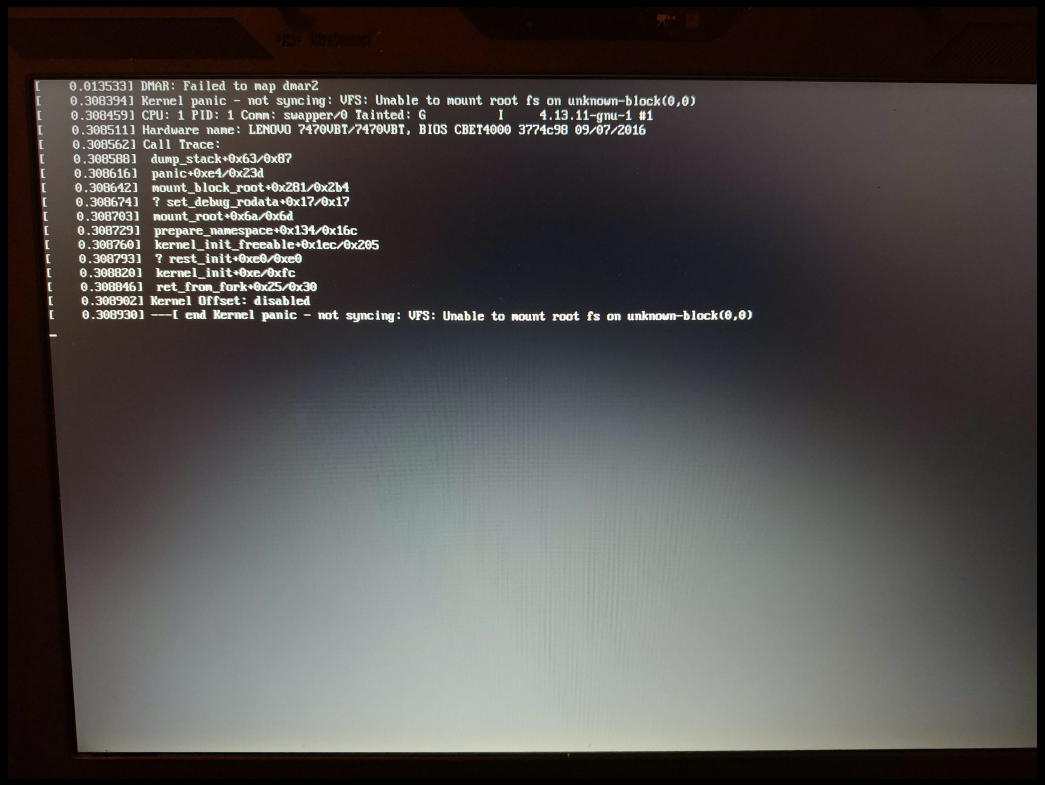
"Jailbreaking" or "rooting" devices often depends on finding and leveraging Kernel bugs

Remember JailbreakMe?

It used a remote code execution primitive inside Safari to trigger a kernel-level exploit to bypass Apple's code-signing protection

Jailbreak Game Console

Kernel Basics



Your Kernel is:

Managing your Processes
Managing your Memory
Coordinating your Hardware

A crash oftentimes means a reboot!

In general, we want to spend as little time there as possible.

West Chester University

Kernel Basics

The Kernel is typically *the most powerful* place we can find bugs

But, how do we go from "vulnerability" to "privileged execution" *without bringing down the rest of the system?*

Kernel Exploitation Strategy

1. Find vulnerability in kernel code
2. Manipulate it to gain code execution
3. Elevate our process's privilege level
4. Survive the "trip" back to userland
5. Enjoy our root privileges

Kernel Exploitation Strategy

You already know how to find these!

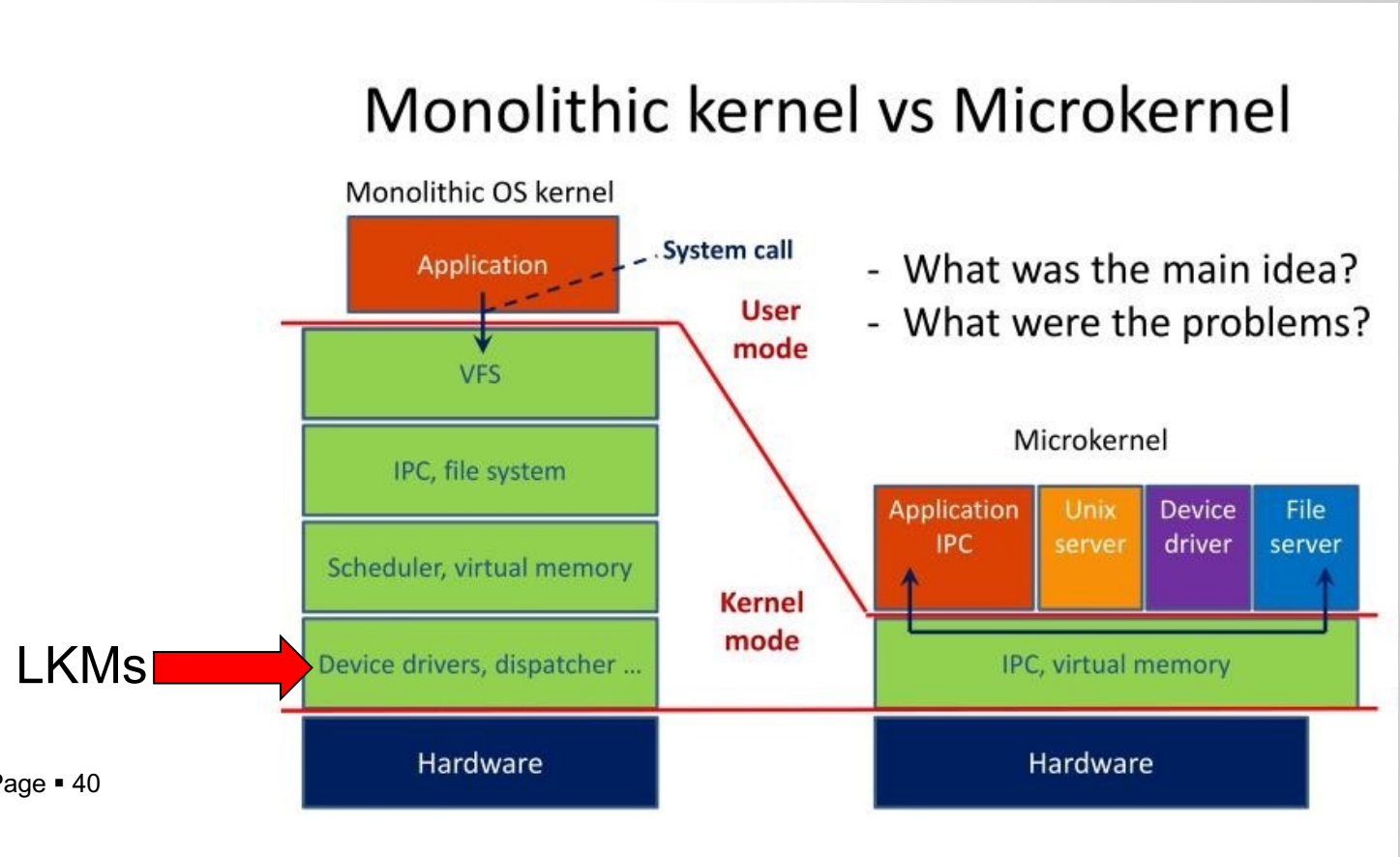
Kernel vulnerabilities are almost *exactly* the same as userland vulnerabilities.

1. Stack Overflows
2. Heap Overflows

By now, finding these should be a familiar process

Kernel Exploitation Strategy

- **Monolithic Kernel**
  - Monolithic kernel is a single large processes running entirely in a single address space. It is a single static binary file. All kernel services exist and execute in kernel address space. The kernel can invoke functions directly.



Kernel Exploitation Strategy

The most common place to find vulnerabilities is inside of Loadable Kernel Modules (LKMs).

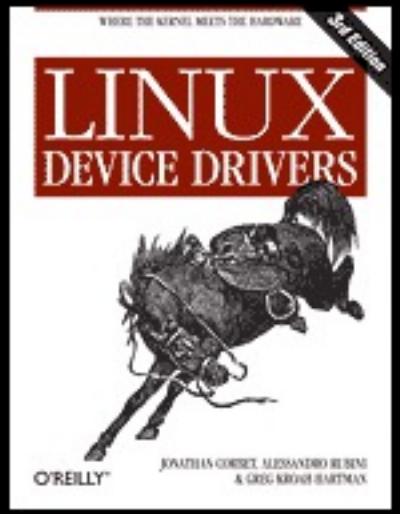
LKMs are like executables that run in Kernel Space.
A few common uses are listed below:

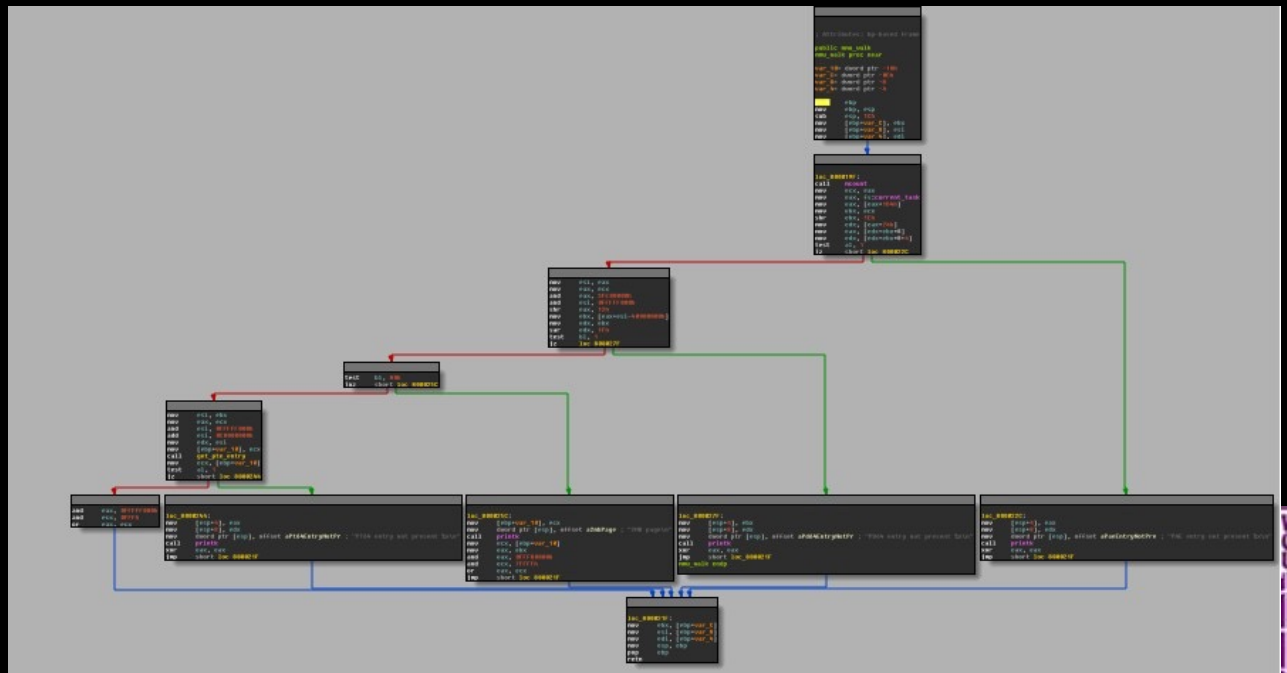
> Device Drivers
> Filesystem Drivers
> Networking Drivers
> Executable Interpreters
> Kernel Extensions
> (rootkits :P)

Kernel Exploitation Strategy

LKMs are just binary blobs like your familiar ELF's, EXE's and MACH-O's. (On Linux, they even use the ELF format)

You can drop them into GDB and reverse-engineer them like you're used to already.





West Chester University

Kernel Exploitation Strategy

There's a few useful commands that deal with LKMs on Linux.

insmod	--->	Insert a module into the running kernel
rmmod	--->	Remove a module from the running kernel
lsmod	--->	List currently loaded modules

A general familiarity with these is helpful

Kernel Exploitation Strategy

The same basic exploitation techniques apply to Kernelspace (After all, it's just x86 code!)

Shellcoding, ROP, Pointer Overwrites,
, etc can all be used to execute code in Kernel Land.

Kernel Functions

Common Library calls are sometimes *different* , so there is a slight learning curve involved.

printf()	--->	printk()
memcpy()	--->	copy_from_user()/copy_to_user()
malloc()	--->	kmalloc() (slab/slub allocator)
free()	--->	kfree()

Typically, whatever you want to know is a quick google-search or man page away.

Kernel Debugging

Debugging kernel code can be difficult

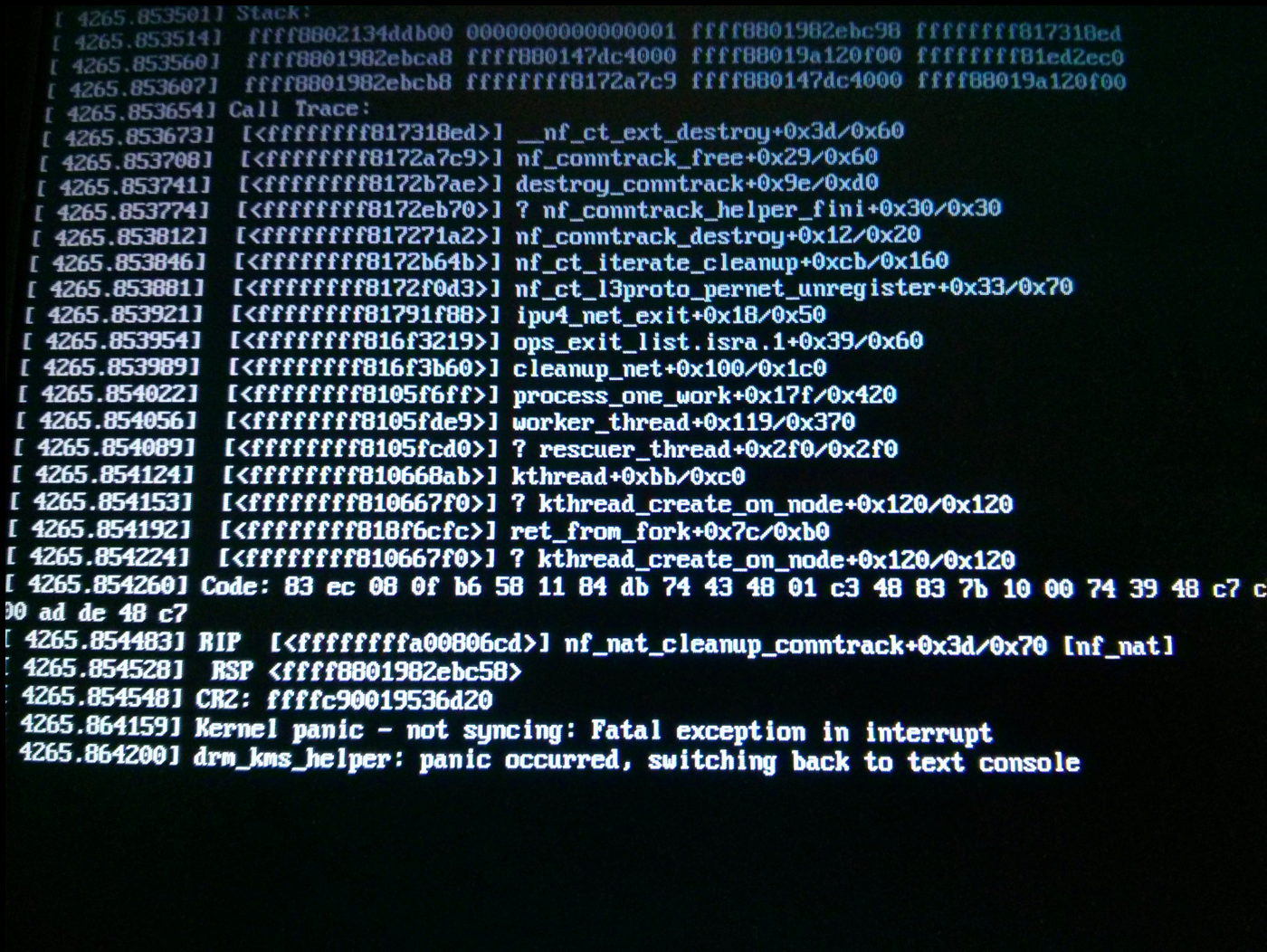
We can't just run the kernel in gdb

You will often have to rely on stack dumps, error messages, and other "black box" techniques to infer what's going on inside the kernel.

Kernel Debugging

This is an example of what you might see if you get a crash in the kernel.

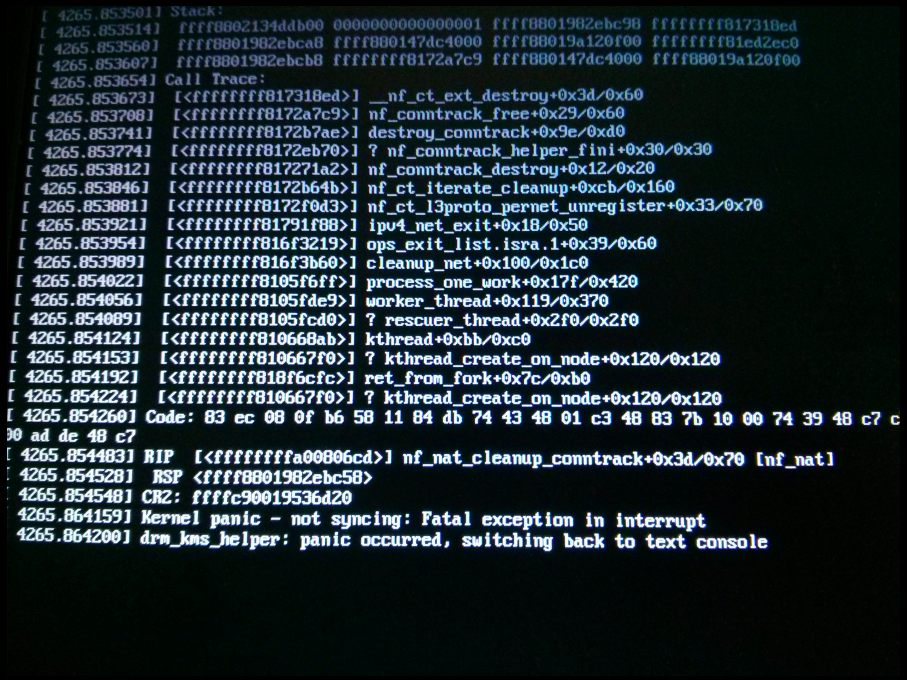
Stack Dump
Call Trace
Register Dump



Kernel Debugging

This is an example of what you might see if you get a crash in the kernel.

Stack Dump
Call Trace
Register Dump



You might be able to see this with dmesg if the crash is not fatal.

West Chester University

Traditional UNIX credentials.

- Real User ID
- Real Group ID

```
➜  give_to_player ls -l
total 19216
-rwxrwxr-x  1 schen schen      202 May  9  2019 boot.sh
-rw-rw-r--  1 schen schen  4127776 May  9  2019 bzImage
-rwxrwxr-x  1 schen schen   898440 Nov 18 01:43 exp
-rwxrwxr-x  1 schen schen   897912 Nov 18 01:33 exp0
-rw-rw-r--  1 schen schen      722 Nov 18 01:33 exp0.c
-rw-rw-r--  1 schen schen     1979 Nov 18 01:27 exp1.c
-rwxrwxr-x  1 schen schen   902704 Nov 18 01:28 exp2
-rw-rw-r--  1 schen schen     2061 Nov 18 01:28 exp2.c
-rwxrwxr-x  1 schen schen   898584 Nov 18 01:29 exp3
-rw-rw-r--  1 schen schen     1072 Nov 18 01:29 exp3.c
drwxrwxr-x 12 schen schen     4096 Nov 18 01:35 fs
-rw-rw-r--  1 schen schen 11913216 Nov 18 01:43 initramfs.img
➜  give_to_player id
uid=1000(schen) gid=1000(schen) groups=1000(schen),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),116(lpadmin),126(sambashare),450(hmacc
)
```

```
  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
31380 schen      20   0 26568  4872  3328 R  0.7  0.0  0:00.24 htop
  458 root       20   0 38232  3148  2752 S  0.7  0.0  3h56:48 @sbin/plymouthd --mode=boot --pid-file=/run/plymouth/pid --attach-to-ses
 1186 gdm        20   0  665M 37460 18068 S  0.7  0.2  3h38:32 /usr/lib/gnome-settings-daemon/gsd-color
    1 root       20   0  220M  9780  6884 S  0.0  0.1 38:28.36 /lib/systemd/systemd --system --deserialize 28
  379 root       20   0 29856  1228  1080 S  0.0  0.0  0:00.00 /sbin/ureadahead -q
  801 root       20   0  424M  9304  7884 S  0.0  0.1  0:00.00 /usr/sbin/ModemManager --filter-policy=strict
  804 root       20   0  424M  9304  7884 S  0.0  0.1  0:01.04 /usr/sbin/ModemManager --filter-policy=strict
  791 root       20   0  424M  9304  7884 S  0.0  0.1  0:01.37 /usr/sbin/ModemManager --filter-policy=strict
  796 messagebu  20   0  143M 11200  8240 S  0.0  0.1  0:36.43 /usr/bin/dbus-daemon --system --address=systemd: --nofork --nopidfile --
  941 root       20   0  165M 16960  9092 S  0.0  0.1  0:00.00 /usr/bin/python3 /usr/bin/networkd-dispatcher --run-startup-triggers
  805 root       20   0  165M 16960  9092 S  0.0  0.1  0:00.04 /usr/bin/python3 /usr/bin/networkd-dispatcher --run-startup-triggers
  814 root       20   0  107M  3516  3180 S  0.0  0.0  0:00.00 /usr/sbin/irqbalance --foreground
  806 root       20   0  107M  3516  3180 S  0.0  0.0  8:53.03 /usr/sbin/irqbalance --foreground
  824 root       20   0  497M 12432 10104 S  0.0  0.1  0:00.00 /usr/lib/udisks2/udisksd
  828 root       20   0  497M 12432 10104 S  0.0  0.1  0:00.78 /usr/lib/udisks2/udisksd
  899 root       20   0  497M 12432 10104 S  0.0  0.1  0:00.00 /usr/lib/udisks2/udisksd
  909 root       20   0  497M 12432 10104 S  0.0  0.1  0:00.00 /usr/lib/udisks2/udisksd
  807 root       20   0  497M 12432 10104 S  0.0  0.1  0:05.08 /usr/lib/udisks2/udisksd
 1106 syslog     20   0  347M  9980  7716 S  0.0  0.1  4:17.49 /usr/sbin/rsyslogd -n
 1107 syslog     20   0  347M  9980  7716 S  0.0  0.1  0:00.01 /usr/sbin/rsyslogd -n
 1108 syslog     20   0  347M  9980  7716 S  0.0  0.1  3:59.20 /usr/sbin/rsyslogd -n
  808 syslog     20   0  347M  9980  7716 S  0.0  0.1  8:17.01 /usr/sbin/rsyslogd -n
  809 root       20   0 62804  6304  5120 S  0.0  0.0  0:14.41 /lib/systemd/systemd-logind
  816 root       20   0  387M 13876 11588 S  0.0  0.1  7:44.09 /usr/lib/accountsservice/accounts-daemon
```

Elevate Privileges

Remember: The Kernel manages running processes

Therefore: The Kernel keeps track of permissions

```
struct cred {
    atomic_t    usage;
#ifdef CONFIG_DEBUG_CREDENTIALS
    atomic_t    subscribers;    /* number of processes subscribed */
    void        *put_addr;
    unsigned    magic;
#define CRED_MAGIC  0x43736564
#define CRED_MAGIC_DEAD 0x44656144
#endif
    kuid_t      uid;        /* real UID of the task */
    kgid_t      gid;        /* real GID of the task */
    kuid_t      suid;       /* saved UID of the task */
    kgid_t      sgid;       /* saved GID of the task */
    kuid_t      euid;       /* effective UID of the task */
    kgid_t      egid;       /* effective GID of the task */
    kuid_t      fsuid;      /* UID for VFS ops */
    kgid_t      fsgid;      /* GID for VFS ops */
    unsigned    securebits; /* SUID-less security management */
    kernel_cap_t    cap_inheritable; /* caps our children can inherit */
    kernel_cap_t    cap_permitted;  /* caps we're permitted */
    kernel_cap_t    cap_effective;  /* caps we can actually use */
    kernel_cap_t    cap_bset;   /* capability bounding set */
    kernel_cap_t    cap_ambient;    /* Ambient capability set */
#ifdef CONFIG_KEYS
    unsigned char   jit_keyring;    /* default keyring to attach requested
                     * keys to */
    struct key __rcu *session_keyring; /* keyring inherited over fork */
    struct key  *process_keyring; /* keyring private to this process */
    struct key  *thread_keyring; /* keyring private to this thread */
    struct key  *request_key_auth; /* assumed request_key authority */
#endif
#ifdef CONFIG_SECURITY
    void        *security;  /* subjective LSM security */
#endif
    struct user_struct *user;   /* real user ID subscription */
    struct user_namespace *user_ns; /* user_ns the caps and keyrings are relative to. */
    struct group_info *group_info;  /* supplementary groups for euid/fsgid */
    struct rcu_head rcu;        /* RCU deletion hook */
} __randomize_layout;
```

https://code.woboq.org/linux/linux/include/linux/cred.h.html#cred

Elevate Privileges

Conveniently, the Linux Kernel has two wrapper functions for updating process credentials and generating process credentials!

```
int commit_creds(struct cred *new) {
        ...
}
```

```
struct cred *prepare_kernel_cred(struct task_struct *daemon) {

}
```

Elevate Privileges

Now we can map out what we need to do

```
commit_creds(prepare_kernel_cred(0));
```

We can find their addresses in **/proc/kallsyms**

```
/ $ cat /proc/kallsyms | grep commit_creds
ffffffff810a1420 T commit_creds
ffffffff81d88f60 R __ksymtab_commit_creds
ffffffff81da84d0 r __kcrctab_commit_creds
ffffffff81db948c r __kstrtab_commit_creds
/ $ cat /proc/kallsyms | grep prepare_kernel_cred
ffffffff810a1810 T prepare_kernel_cred
ffffffff81d91890 R __ksymtab_prepare_kernel_cred
ffffffff81dac968 r __kcrctab_prepare_kernel_cred
ffffffff81db9450 r __kstrtab_prepare_kernel_cred
```

Returning to UserSpace

Why bother returning to Userspace?

Most useful things we want to do are *much* easier from userland.

In KernelSpace, there's no easy way to:

> Modify the filesystem
> Create a new process
> Create network connections

Returning to UserSpace

How does the kernel do it?

```
push    $SS_USER_VALUE
push    $USERLAND_STACK
push    $USERLAND_EFLAGS
push    $CS_USER_VALUE
push    $USERLAND_FUNCTION_ADDRESS
swapgs
iretq
```

This *will usually* get you out of “Kernel Mode” safely.

Returning to UserSpace

For exploitation, the easiest strategy is highjacking execution, and letting the kernel return by itself.

> Function Pointer Overwrites
> Syscall Table Highjacking
> Use-After-Free

You need to be very careful about destroying Kernel state.

A segfault probably means a reboot!

Example: Babydriver

```
→  babydriver ls -l
total 13228
-rwxrwxr-x 1 schen schen     216 Jul  4  2017 boot.sh
-rw-rw-r-- 1 schen schen 7009392 Jun 16  2017 bzImage
-rw-rw-r-- 1 schen schen 6528512 Nov 18 01:09 rootfs.cpio
```

https://github.com/ctf-wiki/ctf-challenges/tree/master/pwn/kernel

Kernel Space Protections

By now, you're familiar with the alphabet soup of exploit mitigations

DEP
ASLR
Canaries
etc...

Green: Present in Kernel Space
Yellow: Present, with caveats

There's a whole new alphabet soup for Kernel Mitigations!

Kernel Space Protections

Some new words in our soup (There's plenty more...)

MMAP_MIN_ADDR
KALLSYMS
RANDSTACK
STACKLEAK
SMEP / SMAP

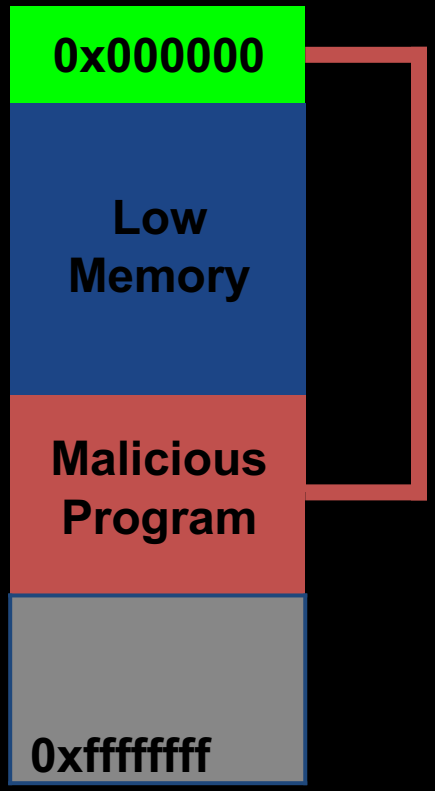
Most of these will be off for the labs!

MMAP_MIN_ADDR

This makes exploiting NULL pointer dereferences harder.

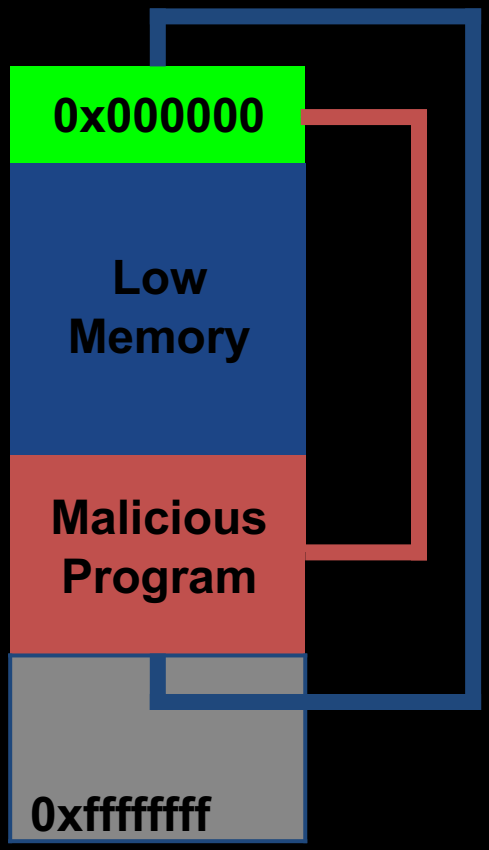
MMAP_MIN_ADDR

This makes exploiting NULL pointer dereferences harder.



Program does mmap(0,....)

MMAP_MIN_ADDR



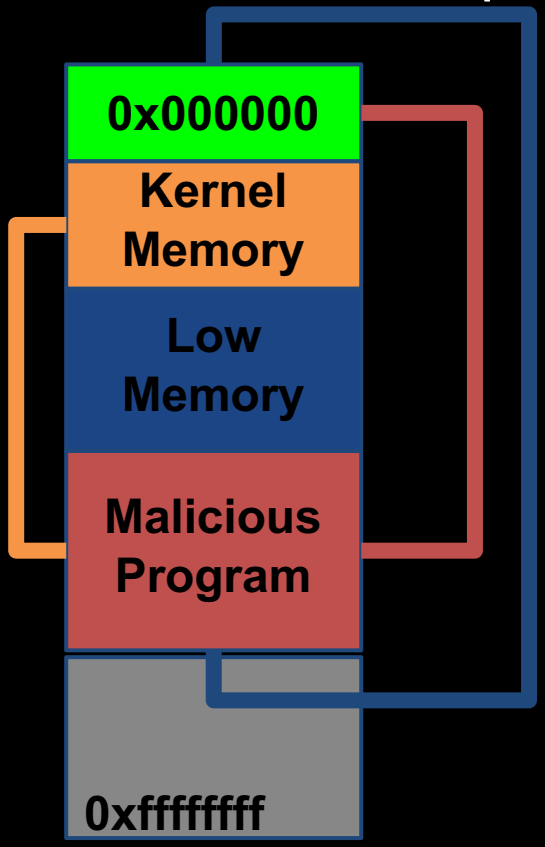
NULL pointer dereferences

- Program does mmap(0,....)

Program writes malicious Code

MMAP_MIN_ADDR

This makes exploiting NULL pointer dereferences harder.



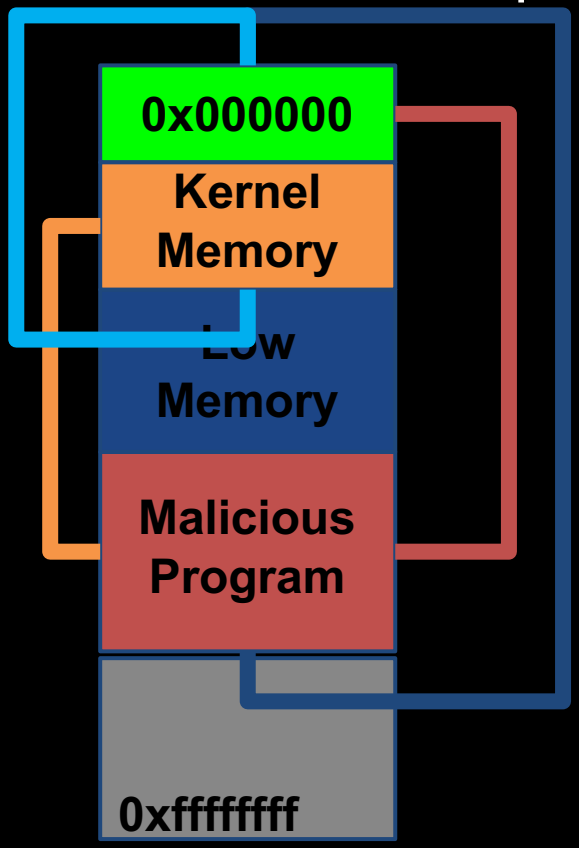
Program does mmap(0,....)

Program writes malicious Code

Program triggers Kernel Bug

MMAP_MIN_ADDR

This makes exploiting NULL pointer dereferences harder.



Program does mmap(0,....)

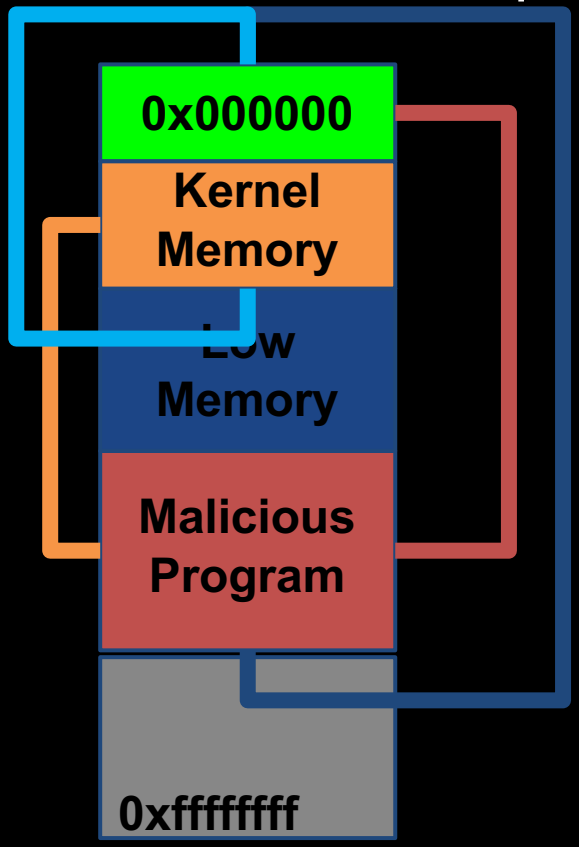
Program writes malicious Code

Program triggers Kernel Bug

Kernel starts executing malicious Code

MMAP_MIN_ADDR

This makes exploiting NULL pointer dereferences harder.



mmap_min_addr disallows programs from allocating low memory.

Makes it much more difficult to exploit a simple NULL pointer dereference in the kernel.

KALLSYMS

/proc/kallsyms gives the address of all symbols in the kernel.

We need this information to write reliable exploits without an info-leak!

```
softsec@softsec-VirtualBox:~$ sudo cat /proc/kallsyms | grep commit_creds
c106bc60 T commit_creds
c17faad4 r __ksymtab_commit_creds
c1806e0c r __kcrctab_commit_creds
c180f2b2 r __kstrtab_commit_creds
softsec@softsec-VirtualBox:~$
```

KALLSYMS

kallsyms used to be world-readable.

Now, it returns 0's for unprivileged users

```
softsec@softsec-VirtualBox:~$ cat /proc/kallsyms | grep commit_creds
00000000 T commit_creds
00000000 r __ksymtab_commit_creds
00000000 r __kcrctab_commit_creds
00000000 r __kstrtab_commit_creds
```

Can still be a useful source of information on older systems

SMEP / SMAP

SMEP: Supervisor Mode Execution Protection

Introduced in Intel IvyBridge

SMAP: Supervisor Mode Access Protection

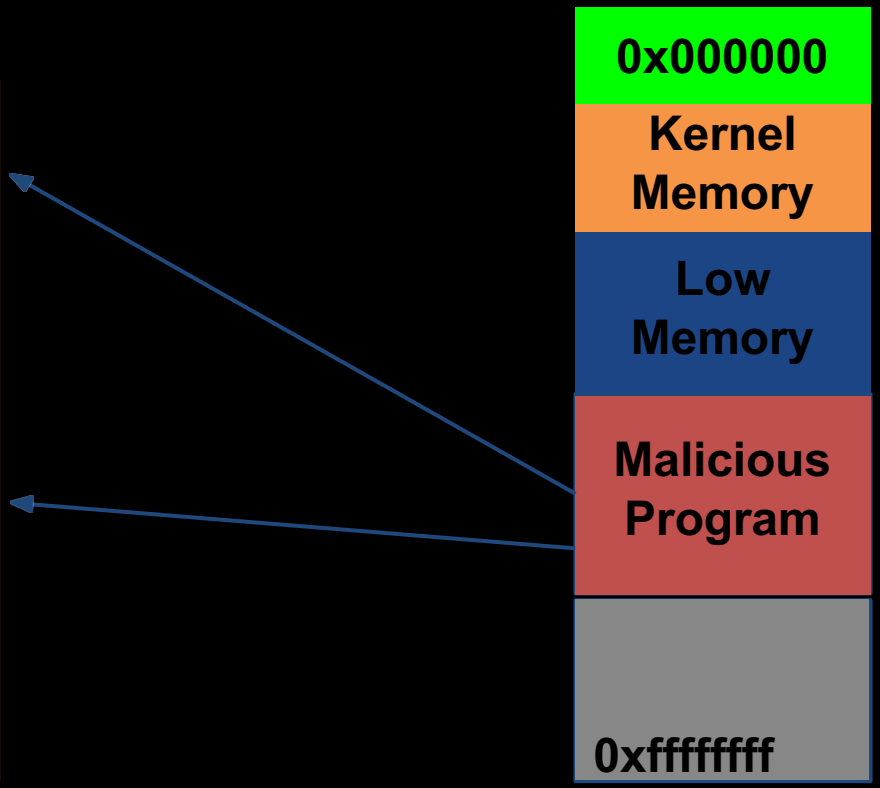
Introduced in Intel Haswell

SMEP / SMAP

Common Exploitation Technique: Supply your own “get root” code.

```
void get_r00t() {
        commit_creds(prepare_kernel_cred(0));
}

int main(int argc, char * argv) {
        …
        trigger_fp_overwrite(&get_r00t);
        …
        //trigger fp use
        trigger_vuln_fp();
        // Kernel Executes get_r00t
        ...
        // Now we have root
        system(“/bin/sh”);
}
```



SMEP / SMAP

SMEP prevents this type of attack by triggering a page fault if the processor tries to execute memory that has the “user” bit set while in “ring 0”.

SMAP works similarly, but for data access in general

This doesn’t *prevent* vulnerabilities, but it adds considerable work to developing a working exploit

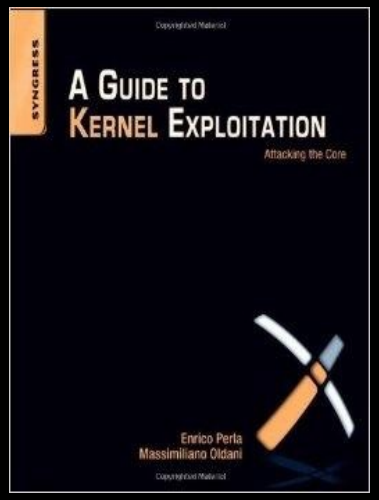
We need to use ROP, or somehow get executable code into kernel memory.

Conclusion

Kernel Exploitation is *weird*, but *extremely powerful*

As userland exploit-dev becomes more challenging and more expensive, kernelspace is becoming a more attractive target.

A single bug can be used to bypass sandboxes, and gain root privileges, which may otherwise be impossible



West Chester University

Q & A