Operating System Security «Offensive Approach»

Özgür KAYA

Outline

- Introduction
- Vulnarabilities
  1. Memory Corruption
  2. NULL Pointer
  3. Race Conditions
- Shellcodes
- Remote Kernel Explotion
- Conclusion

Introduction

- 1994: The pentium processor computes wrong divisions
  - INTEL forced to replace most processors
  - Economic damage: 450 million Dollars !
- 1995: The software MacInTax spreads the secrets of US tax payers
  - Error in the debug code distributed
  - Users can use it to access the server
  - Everybody can read and modify any tax

Introduction (2)

- 1995: Problems in Denver Airport
  - The fully automated baggage system fails
  - Considerable congestion and lack of design
  - The system is too complex to recover
  - In 2005 system is still not be working

- 1996: Vector Ariane 5 explodes during take off
  - The control software assigns a 64 bit number to a 16 bit variable
  - The code was recycled from Ariane 4
  - Ariane 5 is fast and its lateral speed does not fit in 16 bits
  - Result: Overflow – system shuts down..
  - The back up computer started
  - .. But still the software is same
  - Damage: 1 Billion Euros !

Introduction (3)

- Need to define what we want
- Need to prove properties rigorously
- Need modular verification techniques
- Need ways to automate the analysis

Security Reports-1

- From a November 4 article by Gregg Keizer's on ComputerWorld:

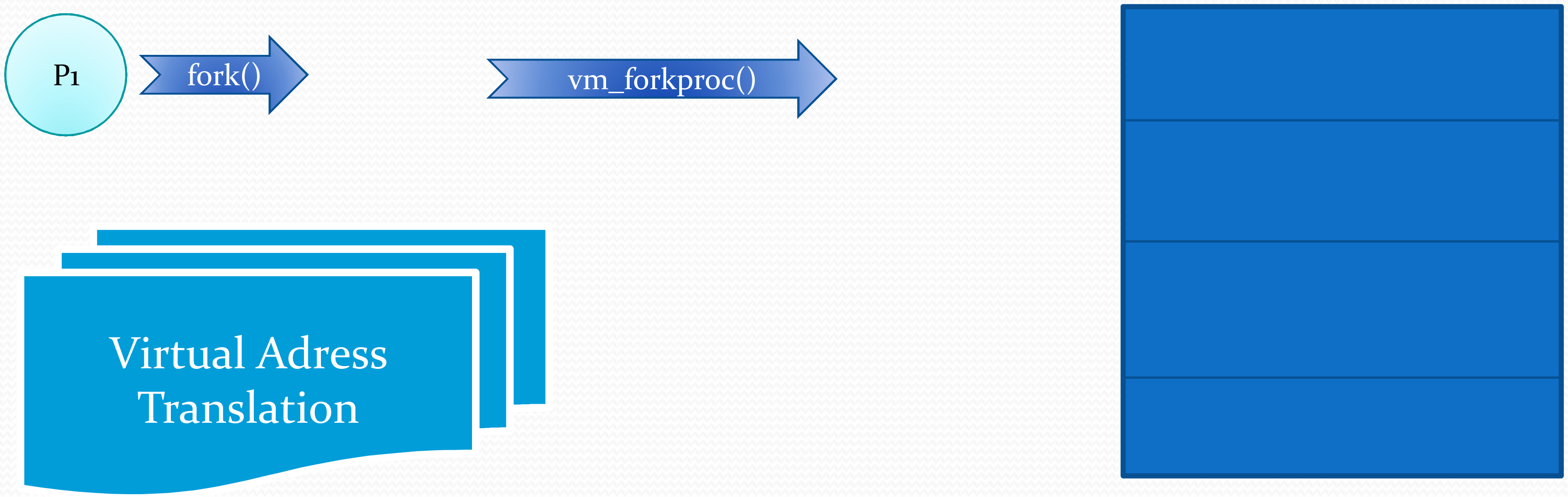
> *Microsoft has been extremely busy patching pieces of the Windows kernel this year.*
>
> *So far during 2011, Microsoft has patched 56 different kernel vulnerabilities with updates issued in February, April, June, July, August and October. In April alone, the company fixed 30 bugs, then quashed 15 more in July*

Security Reports-2

- Linux kernel vulnerabilities: State-of-the-art defenses and open problems (2011) :

141 Linux kernel vulnerabilities discovered from January 2010 to March 2011

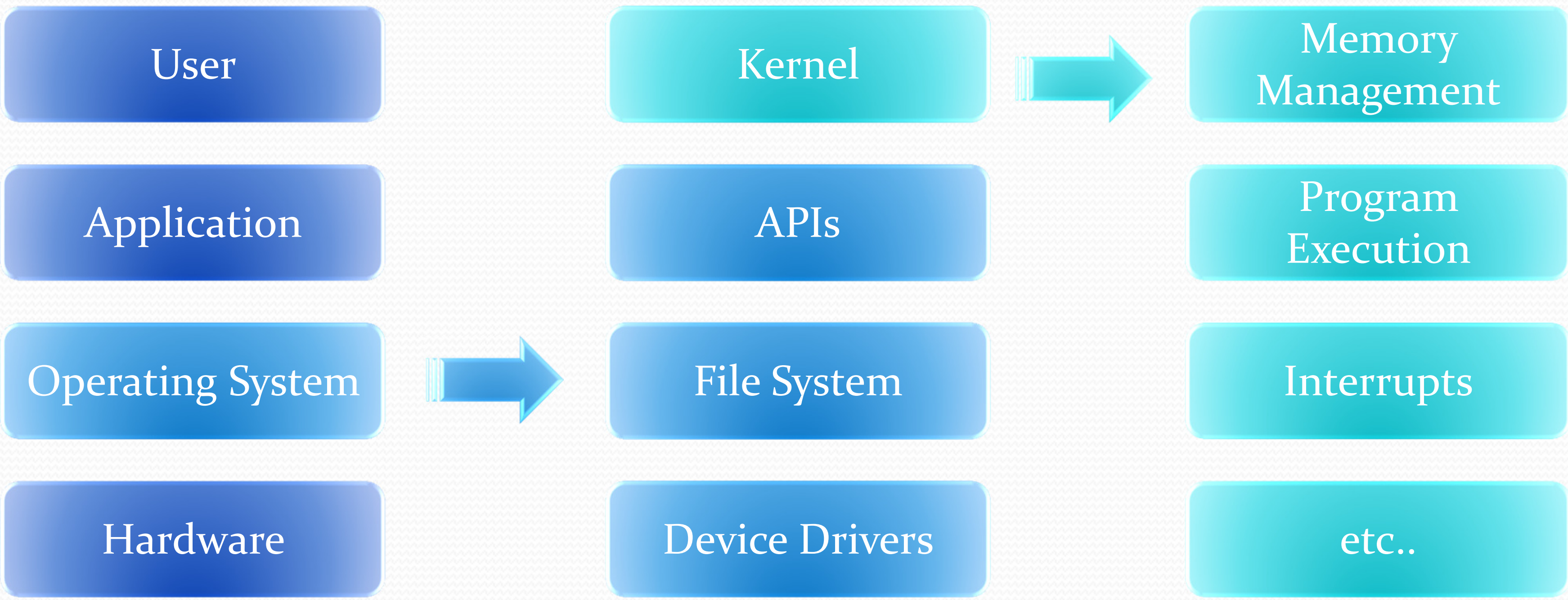
Example Scenario:
Running a program (in linux)



Page Number	Frame Address
2	101
3	102
4 (P1)	101
5	103

Secure Operating Systems

- So… Where does security begin from ?



Kernel Vulnerabilities

- Vulnerabilities (rows) vs. possible exploits (columns)

Vulnerability	Mem. corruption	Policy violation	DoS	Info. disclosure	Misc.
Missing pointer check	6	0	1	2	0
Missing permission check	0	15	3	0	1
Buffer overflow	13	1	1	2	0
Integer overflow	12	0	5	3	0
Uninitialized data	0	0	1	28	0
Null dereference	0	0	20	0	0
Divide by zero	0	0	4	0	0
Infinite loop	0	0	3	0	0
Data race / deadlock	1	0	7	0	0
Memory mismanagement	0	0	10	0	0
Miscellaneous	0	0	5	2	1
Total	32	16	60	37	2

Some vulnerabilities allow for more than one kind of exploit, but vulnerabilities that lead to memory corruption are not counted under other exploits.

Kernel Vulnerabilities

- Vulnerabilities (rows) vs. locations (columns)

Vulnerability	Total	core	drivers	net	fs	sound
Missing pointer check	8	4	3	1	0	0
Missing permission check	17	3	1	2	11	0
Buffer overflow	15	3	1	5	4	2
Integer overflow	19	4	4	8	2	1
Uninitialized data	29	7	13	5	2	2
Null dereference	20	9	3	7	1	0
Divide by zero	4	2	0	0	1	1
Infinite loop	3	1	1	1	0	0
Data race / deadlock	8	5	1	1	1	0
Memory mismanagement	10	7	1	1	0	1
Miscellaneous	8	2	0	4	2	0
Total	141	47	28	35	24	7

From the source

- Following snippet of code taken from the 2.6.9 version of the Linux Kernel

```
static int bluez_sock_create(struct socket *sock, int proto)
{
if (proto >= BLUEZ_MAX_PROTO)
return -EINVAL;
[…]
return bluez_proto[proto]->create(sock,proto);
}
```

Uninitialized/Nonvalidated/Corrupted Pointer Dereference

- Most famous kernel bug class:
  - NULL pointer dereference (1)

Uninitialized/Nonvalidated/Corrupted Pointer Dereference

- NULL pointer dereference vulnerabilities are a subset of a larger class
- A static declared pointer is initialized to NULL
  - what happens to a pointer declared as a local variable in a function?
  - what is the content of a pointer contained in a structure freshly allocated in memory? (1)

Uninitialized/Nonvalidated/Corrupted

Pointer Dereference

- Pointer is a variable:
  - it has a size
  - needs to be stored in memory

Data type	LP32	ILP32	LP64	ILP64	LLP64
Char	8	8	8	8	8
Short	16	16	16	16	16
Int	16	32	32	64	32
Long	32	32	64	64	32
Long long	64	64	64	64	64
Pointer	32	32	64	64	64

The size of the pointer depends on the data model

Uninitialized/Nonvalidated/Corrupted Pointer Dereference

- let's say the ILP32 model is in place;

```
#include <stdio.h>
#include <strings.h>

void big_stack_usage() {
char big[200];
memset(big,'A', 200);
}
void ptr_un_initialized() {
char *p;
printf("Pointer value: %p\n", p);
}
int main()
{
big_stack_usage();
ptr_un_initialized();
}
```

Is possible to predict the value of that memory ?

```
macosxbox$ gcc -o p pointer.c
macosxbox$ ./p
Pointer value: 0x41414141
macosxbox$
```

Uninitialized/Nonvalidated/Corrupted Pointer Dereference

This vulnerability allows a user to pass a kernel address to the kernel, and therefore directly access (modify) kernel memory.

vmsplice_to_user()

```
    }
    [...]
    sd.u.userptr = base;  2
    [...]
    size = __splice_from_pipe(pipe, &sd, pipe_to_user);
[...]
static int pipe_to_user(struct pipe_inode_info *pipe, struct
pipe_buffer *buf,   struct splice_desc *sd)
{
    if (!fault_in_pages_writeable(sd->u.userptr, sd->len)) {
    src = buf->ops->map(pipe, buf, 1);
        ret = __copy_to_user_inatomic(sd->u.userptr, src +
        buf->offset, sd->len);  3
    buf->ops->unmap(pipe, buf, src);
[...]
}
```

get_user() [1] destination pointer is never validated and is passed, through [2]

Memory Corruption Vulnerabilities

- There are two basic types of kernel memory:
  - the kernel stack:
    - associated to each thread/process whenever it runs at the kernel level
  - The kernel heap:
    - used each time a kernel path needs to allocate some small object or some temporary space
- Misbehaving code that overwrites the kernel's contents

Kernel **Stack** Vulnerabilities

- Comprise the growth direction
  - either downward, from higher addresses to lower addresses, or vice versa
- Register keeps track of its top address
  - stack pointer
- Procedures interact with it
  - how local variables are saved, how parameters are

Some operating systems, such as Linux, use so-called interrupt stacks. These are per-CPU stacks that get used each time the kernel has to handle some kind of interrupt (in the Linux kernel case, external hardware-generated interrupts). This particular stack is used to avoid putting too much pressure on the kernel stack size in case small (4KB for Linux) kernel stacks are used.

Kernel **Stack** Vulnerabilities

- Unsafe C functions, such as strcpy() or sprintf()
- An incorrect termination condition in a loop

```
#define ARRAY_SIZE 10
void func() {
    int array[ARRAY_SIZE];
    for (j = 0; j <= ARRAY_SIZE; j++) {
        array[j] = some_value;
        [...]
}
```

potentially overwriting sensitive memory !

- Safe C functions, such as strncpy(), memcpy(),or snprintf()
  - incorrectly calculating the size of the destination buffer

Kernel **Heap** Vulnerabilities

- Kernel implements a virtual memory abstraction:
  - creating the illusion of a large and independent virtual address space for all the user-land processes
    - indeed, for itself
- Using the physical page allocator for allocating space for a large variety of small objects would be extremely inefficient
  - Fragmentation
  - burden on the physical page allocator

Integer Issues

- Have a specific size which determines the range of values
  - Signed / Unsigned
- This kind of vulnerability is usually not exploitable!
- ..but it does lead to other vulnerabilities
  - in most cases, memory overflows

(Arithmetic) Integer Overflows

- Undefined behavior:

> Integer overflow occurs when you attempt to store inside an integer variable a value that is larger than the maximum value the variable can hold

- Integer overflows are the consequence of "wild" increments/multiplications, generally due to a lack of validation of the variables involved.
  - <u>As an example</u>:

At [3] which will affect the «ssize» depends on «nent»
on 32 bit systems likely to cause overflow

```
static int64_t
kaioc(long a0, long a1, long a2, long a3, long a4, long a5)
{
[…]
    switch ((int)a0 & ~AIO_POLL_BIT) {
[…]
    case AIOSUSPEND:
    error = aiosuspend((void *)a1, (int)a2,      1
    (timespec_t *)a3,  (int)a4, &rval, AIO_64);
    break;
[…]
/*ARGSUSED*/
static int
aiosuspend(void *aiocb, int nent, struct timespec *timout,
int flag, long *rval, int run_mode)
{
[…]
    size_t ssize;
[…]
        aiop = curproc->p_aio;
    if (aiop == NULL || nent <=0)   2
        return (EINVAL);
    if (model == DATAMODEL_NATIVE)
        ssize = (sizeof (aiocb_t *) * nent);
    else
        ssize = (sizeof (caddr32_t) * nent);   3
```

```
[…]
    cbplist = kmem_alloc(ssize, KM_NOSLEEP)   4
    if (cbplist == NULL)
        return (ENOMEM);
    if (copyin(aiocb, cbplist, ssize)) {
        error = EFAULT;
    goto done;
    }
[…]
    if (aiop->aio_doneq) {
        if (model == DATAMODEL_NATIVE)
            ucbp = (aiocb_t **)cbplist;
        else
            ucbp32 = (caddr32_t *)cbplist;
[…]
    for (i = 0; i < nent; i++) {   5
        if (model == DATAMODEL_NATIVE) {
            if ((cbp = *ucbp++) == NULL)
```

Sign Conversion Issues

- Occur when the same value is erroneously evaluated first as an unsigned integer and then as a signed one (or vice versa)

- Same value differs in signed or unsigned

```
int fw_ioctl (struct cdev *dev, u_long cmd, caddr_t data, int flag,
fw_proc *td)
{
    [...]
    int s, i, len, err = 0;   (1)
    [...]
    struct fw_crom_buf *crom_buf = (struct fw_crom_buf *)data;   (2)
    [...]
    if (fwdev == NULL) {
    [...]
        len = CROMSIZE;
    [...]
    } else {
    [...]
        if (fwdev->rommax < CSRROMOFF)
            len = 0;
        else
            len = fwdev->rommax - CSRROMOFF + 4;
        }
    if (crom_buf->len < len)   (3)
        len = crom_buf->len;
    else
        crom_buf->len = len;
    err = copyout(ptr, crom_buf->ptr, len);   (4)
```

len [1] ,crom_buf->len are of the signed integer type

at [3] can be satisfied by setting crom_buf->len to a negative value

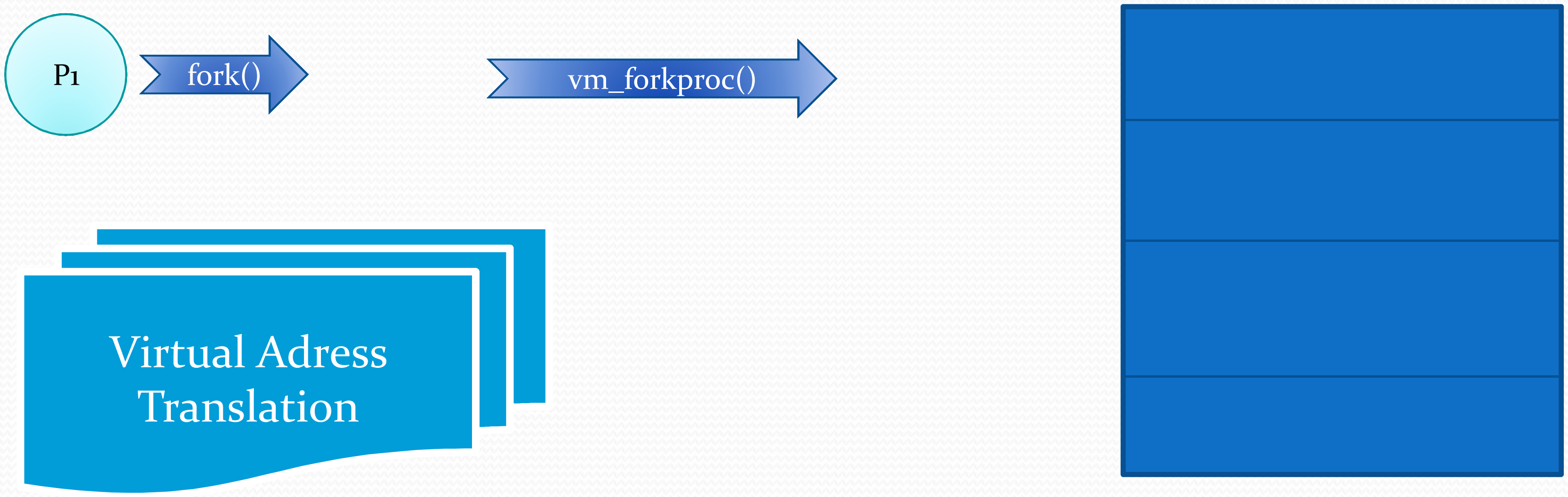
int copyout(
const void * __restrict kaddr,
void * __restrict udaddr,
size_t len)

Size_t is an unsigned int < 0

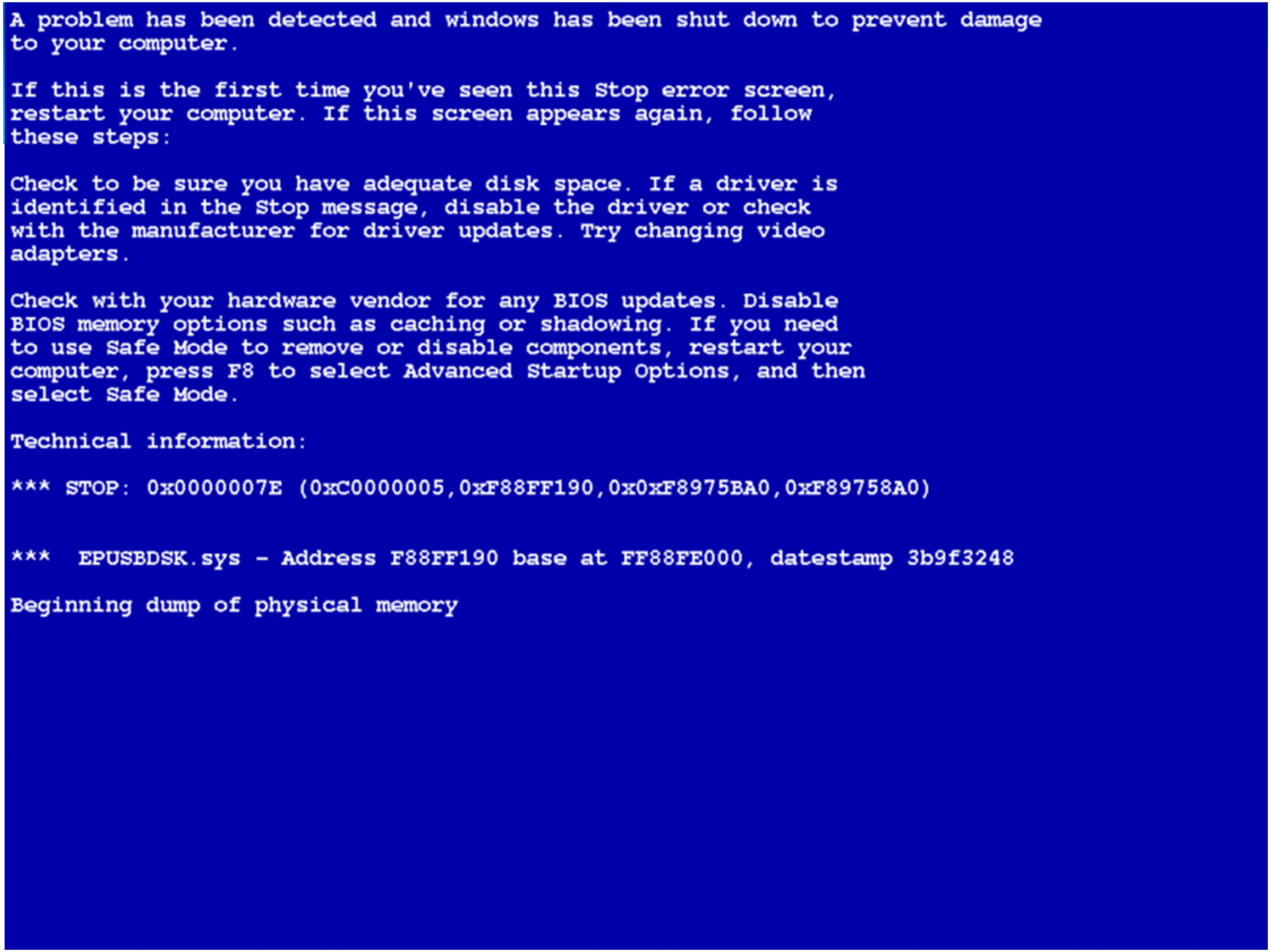
this issue translates to an arbitrary read of kernel memory

Scenario:

Arbitrary Read of Kernel Memory



Page Number	Frame Address
2	101
3	102
4 (P1)	101
5	103


A problem has been detected and windows has been shut down to prevent damage
to your computer.
If this is the first time you've seen this Stop error screen,
restart your computer. If this screen appears again, follow
these steps:
Check to be sure you have adequate disk space. If a driver is
identified in the Stop message, disable the driver or check
with the manufacturer for driver updates. Try changing video
adapters.
Check with your hardware vendor for any BIOS updates. Disable
BIOS memory options such as caching or shadowing. If you need
to use Safe Mode to remove or disable components, restart your
computer, press F8 to select Advanced Startup Options, and then
select Safe Mode.
Technical information:
*** STOP: 0x0000007E (0xC0000005,0xF88FF190,0x0xF8975BA0,0xF89758A0)
*** EPUSBDSK.sys - Address F88FF190 base at FF88FE000, datestamp 3b9f3248
Beginning dump of physical memory

Sign Conversion Issues

- 1996: Vector Ariane 5 explodes during take off
    - The control software assigns a 64 bit number to a 16 bit variable
    - The code was recycled from Ariane 4
    - Ariane 5 is fast and its lateral speed does not fit in 16 bits
    - Result: Overflow – system shuts down..
    - The back up computer started
    - .. But still the software is same
    - Damage: 1 Billion Euros !
- Aside from the C99 standard, a very good reference for helping to understand these rules and related issues is the CERT Secure Coding Standard

Race Conditions

- Occur:
  - the (two or more) actors need to execute their action concurrently (SMP)
  - At least, be interleaved one with the other (UP) (1)

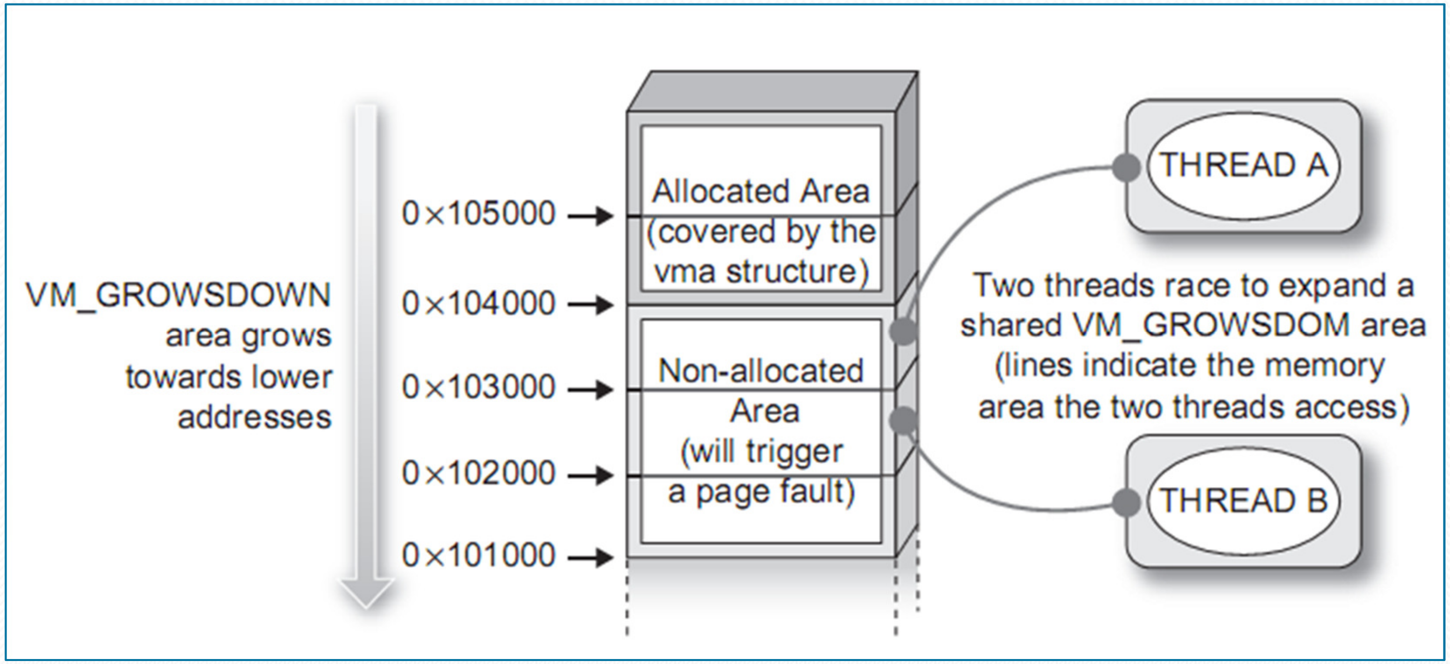
- Solution :
  - Synchronization *(synchronization primitives : e.g., locks, semaphores, conditional variables, etc.) (2)*

In recent years, race conditions have led to some of the most fascinating bugs and exploits at the kernel level, among them **sys_uselib** and the **page fault handler** issues on the Linux kernel.

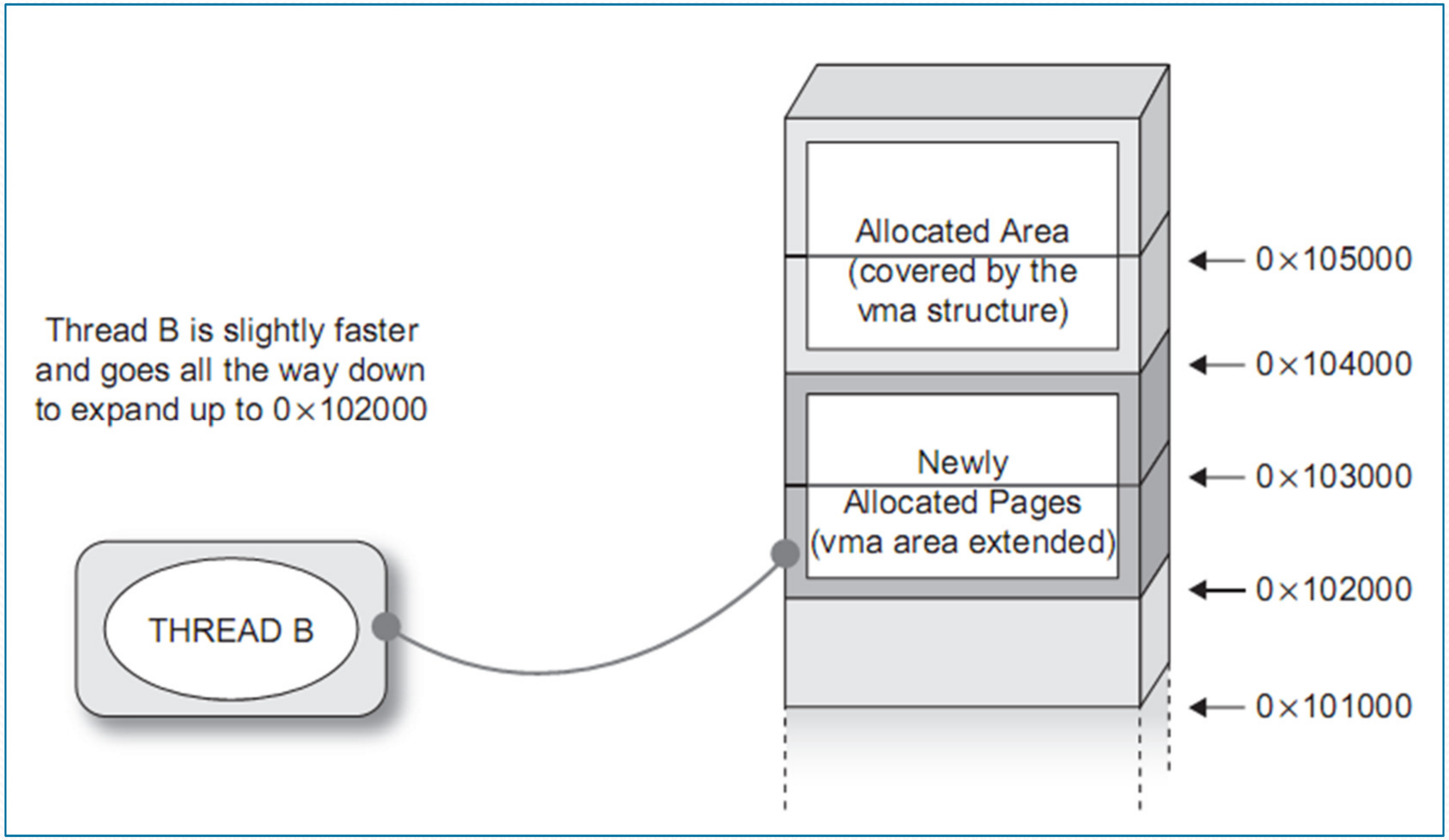
Page Fault Handler

```
down_read(&mm->mmap_sem);
    vma = find_vma(mm, address);
    if (!vma)                                  1
        goto bad_area;
    if (vma->vm_start <= address)              2
        goto good_area;
    if (!(vma->vm_flags & VM_GROWSDOWN))       3
        goto bad_area;
    if (error_code & 4) {
        /*
        * accessing the stack below %esp is always a bug.
        * The "+32" is there due to some instructions (like
        * pusha) doing post-decrement on the stack and that
        * doesn't show up until later..
        */
    if (address + 32 < regs->esp)
        goto bad_area;
    }
    if (expand_stack(vma, address))            4
        goto bad_area;
```

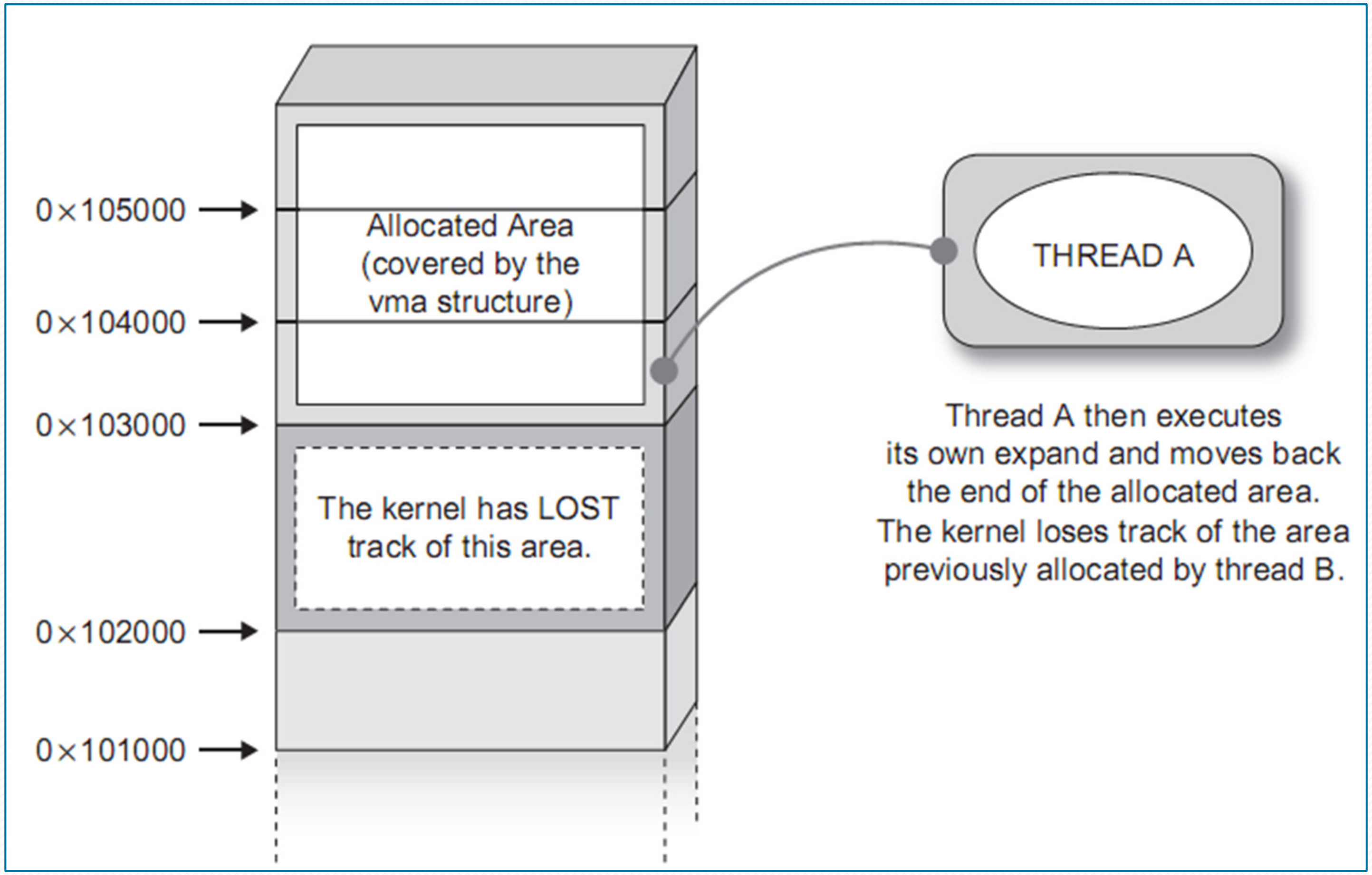
Two threads racing to expand a common VM_GROWSDOWN area.



Intermediate memory layout when thread B succeeds.



Final memory layout once thread A is also complete.



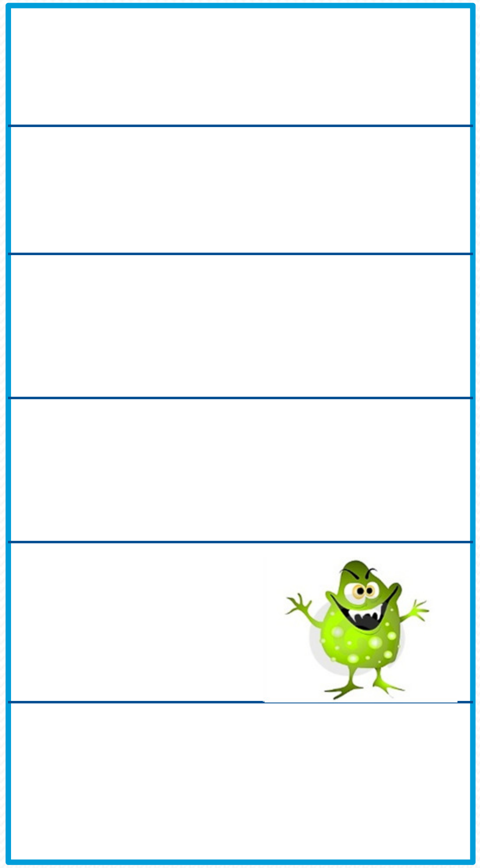
Scenario:

Arbitrary Read of Kernel Memory

P1 → fork() → vm_forkproc()

Virtual Adress Translation

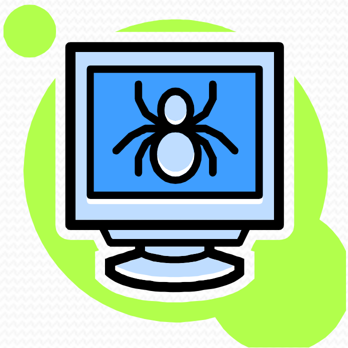
Page Number	Frame Address
2	101
3	102
4 (P1)	101
5	103

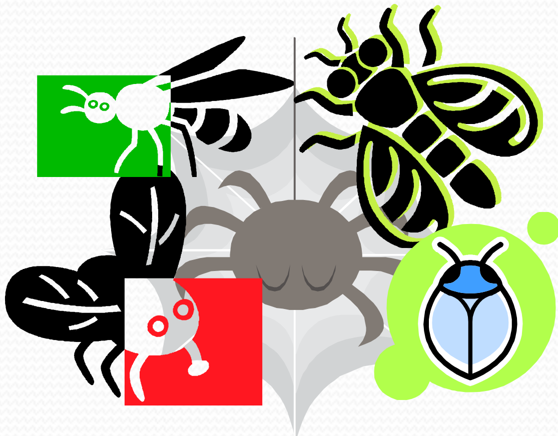


Encountering new malware

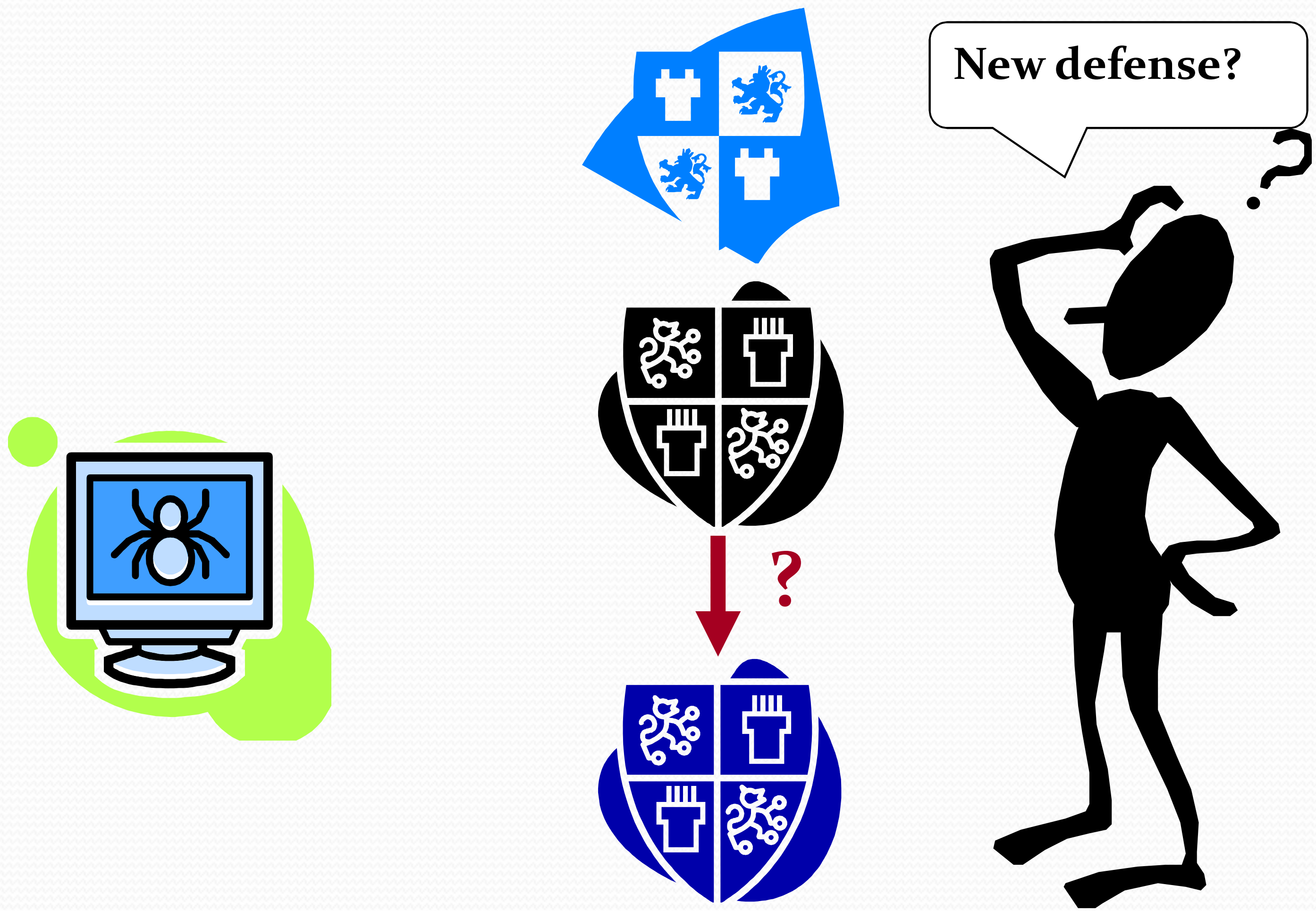
Have I seen this before?

How closely related is it to what I have seen before?

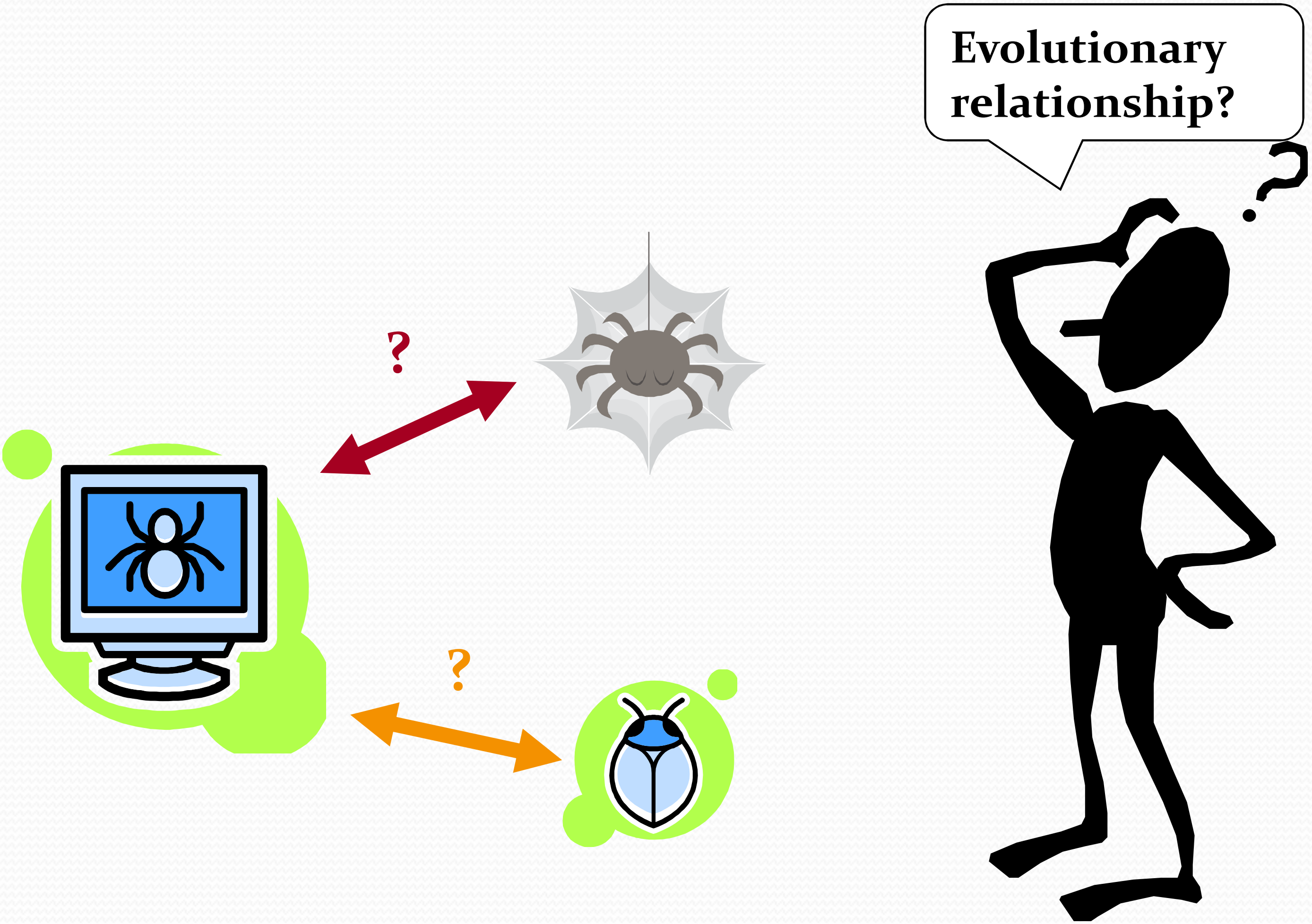




Practical considerations



Theoretical considerations



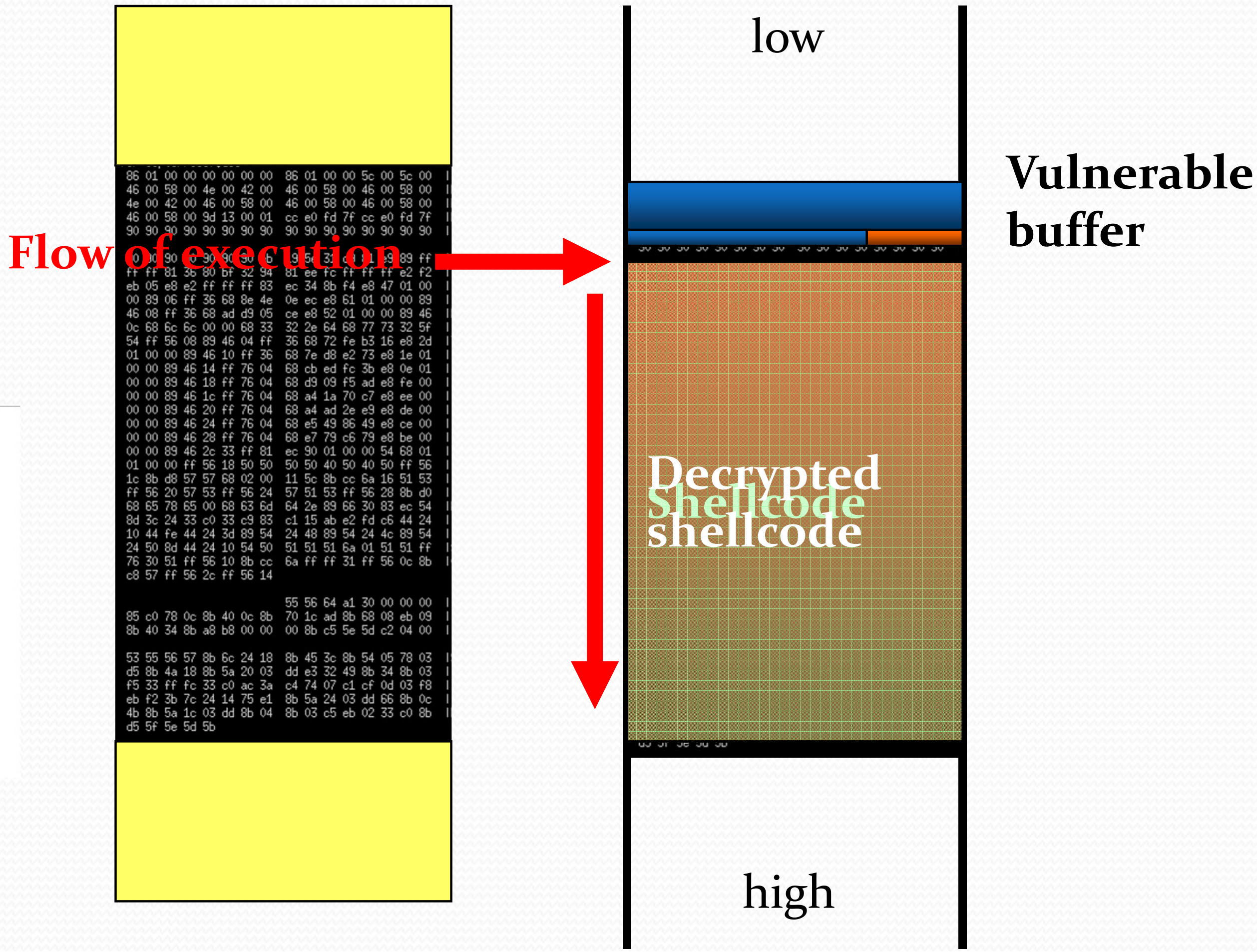
Why shellcodes?

- Our study focuses on exploits
- They are packaged with the exploit
  - First foreign code that executes on a newly infected machine
  - Part of exploit with most leeway for variation
- Primary challenge: collecting and analyzing shellcodes

Remote code injection attacks

MS RPC Request Exploit

Victim



Victim's stack memory

Why Remote Kernel Exploits?

- Instant root
  - No need to escalate privileg

- Remote userland exploitation
  - Full ASLR + NX/DEP
  - Sandboxing
  - Reduced privileges

Goals of This Talk

- Explore operating system internals from perspective of an attacker
- Discuss kernel data structures and subsystems
- Exploit development methodology
- Individual bugs vs. exploit techniques
- Discuss next steps for kernel hardening

Challenges of Remote Kernel Exploitation

- Consequence of failed remote userland exploit:
  - Crash application/service, wait until restarted
  - Crash child process, try again immediately

- Consequence of failed remote kernel exploit:
  - Kernel panic, game over

Linux Networking

- What happens when network data is received?
- Hardware magic happens, driver layer (linux/drivers/net) receives low-level frame
- Driver identifies "this is an IP packet", sends to network layer (linux/net/ipv{4,6})
- Network layer checks "what protocol is this" (TCP, UDP, ICMP, etc.) and dispatches to appropriate protocol handler (linux/net/*)

What Can We Achieve?

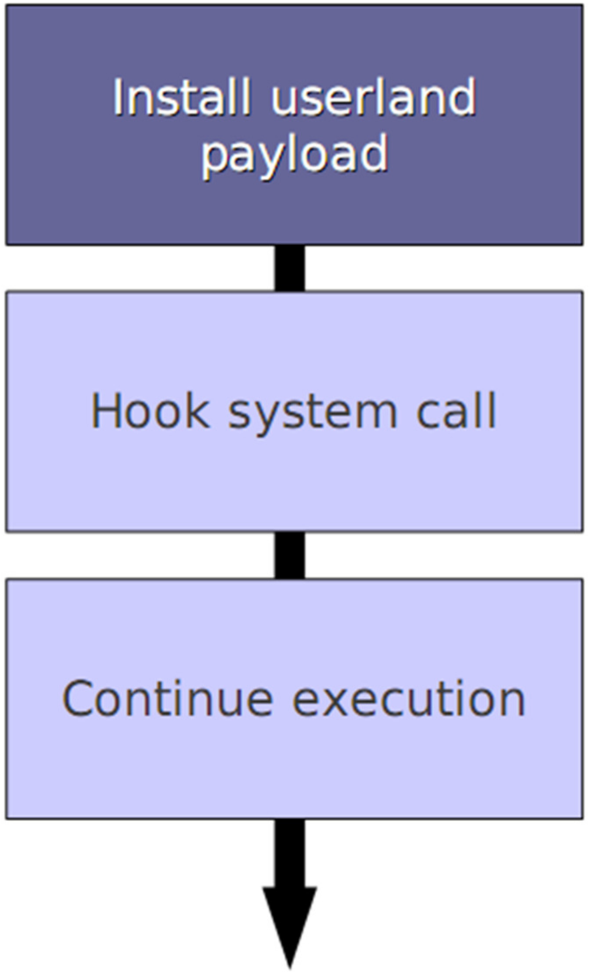
- Trigger the overflow, gain control of EIP
- Leverage ROP to mark softirq stack executable, jump into shellcode
- Search for intact ROSE frame on kernel heap, mark executable, jump into it
- Install kernel backdoor by hooking ICMP handler
- Do some necessary cleanup and unwind stack for safe return from softirq

What About That Backdoor Part?

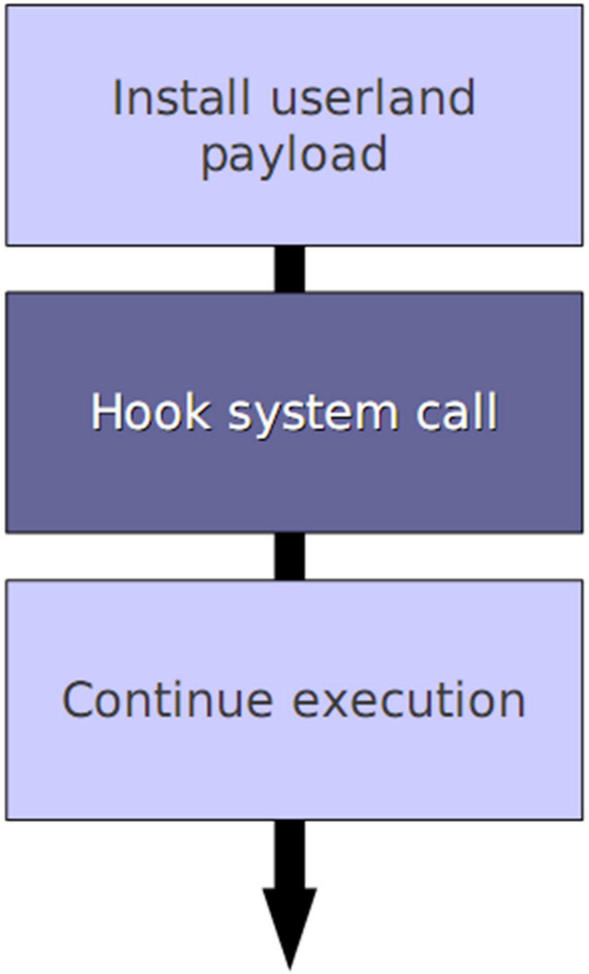
- Whenever an ICMP packet is received, our hook is called
- Check for magic tag in ICMP header
- Two distinct types of packets
  - “Install” packets contain userland shellcode
  - “Trigger” packets cause shellcode to execute
- May be sent independently
  - Install payload, trigger it repeatedly at later date

Backdoor Strategy

- Problem: ICMP handler also runs in softirq context
  - Want userland code execution
- Phase 1: transition to kernel-mode process context
- Phase 2: hijack userland control flow



- Check for magic tag and packet type
- If “install” packet, copy userland payload into safe place (softirq stack)



- If “trigger” packet, need to transition to process context

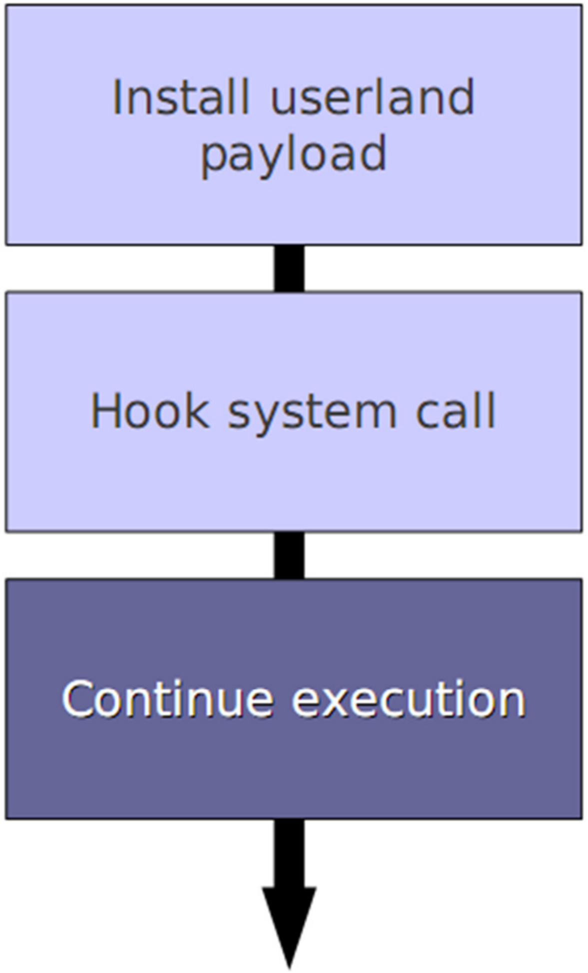
- Easiest way: hook system call

System Calls

- Userland process invokes a system call (read, write, fork, etc.)
- Traditional mechanism is int 0x80 (more recently everything uses systenter/syscall)
- Index into Interrupt Descriptor Table, check privileges
- Invokes handler specified by IDT (syscall entry point)
- Syscall entry point parses arguments, indexes into syscall table, and calls appropriate system call handler

System Call Hijacking

- How to find system call table at runtime?
  - sidt instruction retrieves IDT address
  - Find handler for INT 0x80 (syscall)
  - Scan function for byte pattern calling into syscall table
- Read-only syscall table
  - More flipping write-protect bit in %cr0
- Store original syscall handler for later, write address of hook into syscall table



- Want working ICMP stack
- Call original ICMP handler

Conclusion: Local vs Global Solutions

- Systematic method for classifying exploits
  - Exploit collection
  - Shellcode extraction and decryption
  - Shellcode comparison using exedit distance
  - Group exploits with clustering
- Similarity between samples in computed phylogenies corresponded well with observed differences
- Useful step toward automating malware classification

- Teşekkürler
- Thank you
- Efcharisto Poly
- Muito Obrigado
- Danke Schön
- Bedankt
- Labai Aciu