Ring Zero

Stuart Nevans Locke

Disclaimer

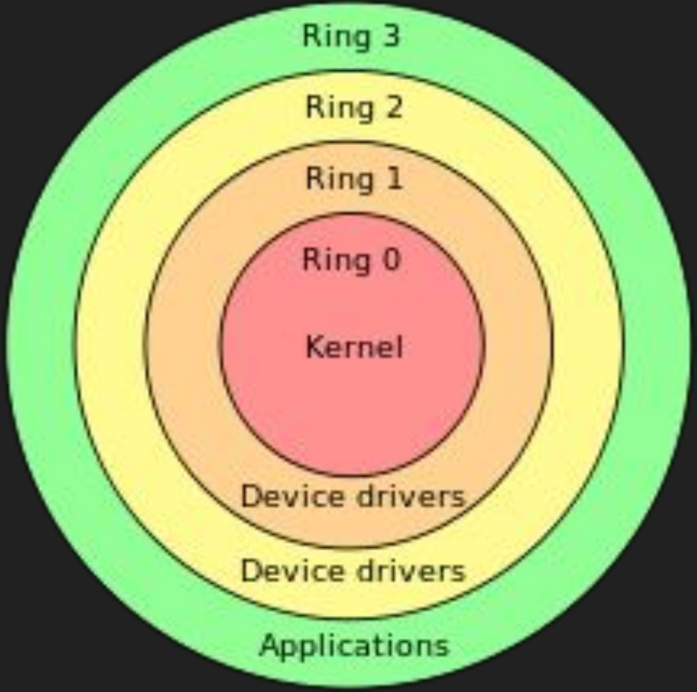
Much of this information is **very** specific to x86 based systems

Overview

- Protection Rings
- Virtual vs Physical Memory
- Pages
- KASLR vs ASLR (KAISER too)
- Userspace/Kernel Communication
- Kernel Security
  - Race Conditions
  - Infoleaks

Protection Rings

- Ring 0 - the kernel
  - All kernel code is executed in ring 0
  - Drivers generally run in ring 0
- Ring 1 and 2
  - Largely useless - Unused by mainstream windows and linux
- Ring 3 - Userspace
  - All normal code runs here
  - We’ve only looked at userspace exploitation so far

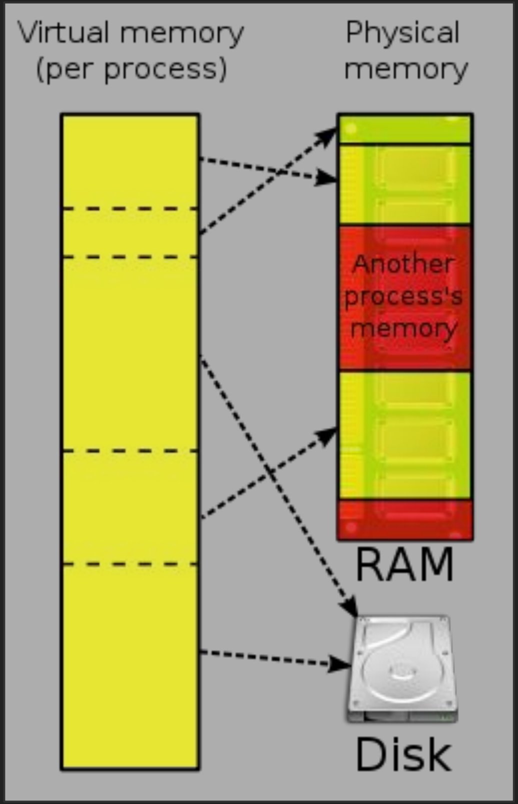


Protection Rings (Cont.)

- Note:
  - Ring 0-3 are the only real protection rings
- Ring -1 - **Hypervisor**
- Ring -2 - **SMM**
- Ring -3 - **IME**

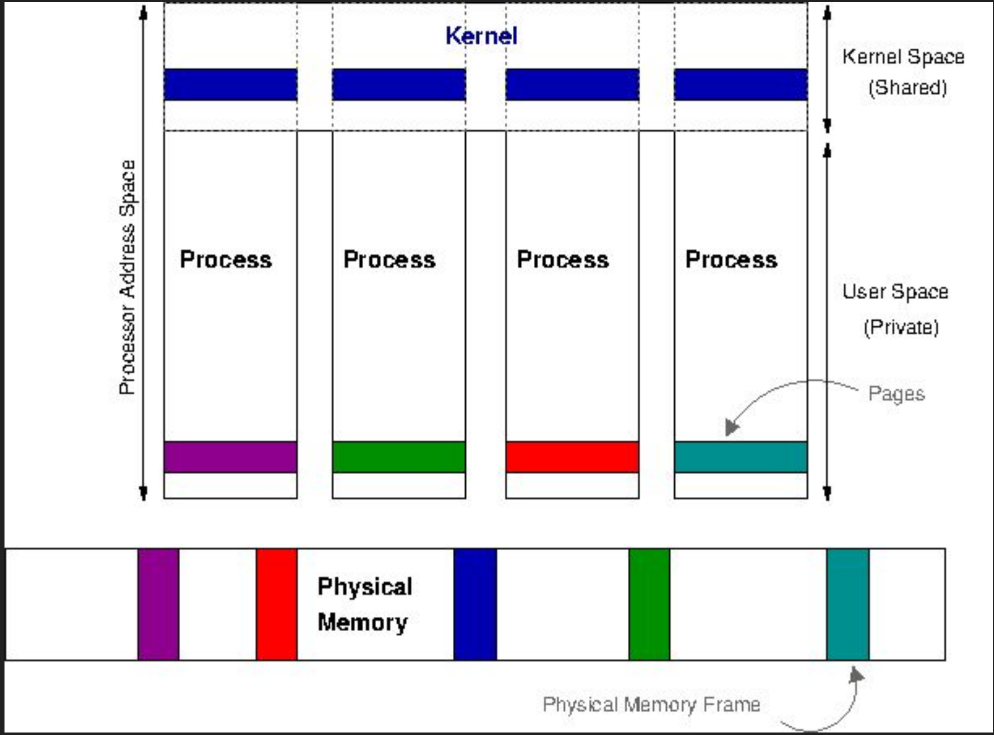
Virtual and Physical Memory

- Physical Memory
  - Exactly what it sounds like
    - Physical Memory directly corresponds to bytes in RAM or other storage
  - Shared by all processes
- Virtual Memory
  - Unique per process
  - Looks identical to physical memory to a process, but it can be stored anywhere.



Virtual Memory (Cont.)

- This is why all programs can have the same address space
    - Remember how ELFs use 0x400000
-



Pages

- Maps virtual memory to physical memory
- Pages also have permissions set, such as RWX
- Typical page size is 4096 bytes

```
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
    0x7f1df8b70000     0x7f1df8d25000 r-xp   1b5000 0      /usr/lib64/libc-2.27.so
    0x7f1df8d25000     0x7f1df8f25000 ---p   200000 1b5000 /usr/lib64/libc-2.27.so
    0x7f1df8f25000     0x7f1df8f29000 r--p     4000 1b5000 /usr/lib64/libc-2.27.so
    0x7f1df8f29000     0x7f1df8f2b000 rw-p     2000 1b9000 /usr/lib64/libc-2.27.so
    0x7f1df8f2b000     0x7f1df8f2f000 rw-p     4000 0
    0x7f1df8f2f000     0x7f1df8f56000 r-xp    27000 0      /usr/lib64/ld-2.27.so
    0x7f1df913c000     0x7f1df913e000 rw-p     2000 0
    0x7f1df9155000     0x7f1df9156000 r--p     1000 26000  /usr/lib64/ld-2.27.so
    0x7f1df9156000     0x7f1df9157000 rw-p     1000 27000  /usr/lib64/ld-2.27.so
    0x7f1df9157000     0x7f1df9158000 rw-p     1000 0
    0x7ffd8eadb000     0x7ffd8eafd000 rw-p    22000 0      [stack]
```

KASLR vs ASLR

- ASLR - Address Space Layout Randomization
    - Very, very good at what it does (Randomizing and HIDING where pages are mapped)
        - Microsoft has a bounty for a generic ASLR bypass
- KASLR - Kernel Address Space Layout Randomization
    - Very, very bad at what it does (Randomizing and HIDING where the kernel pages are mapped)
        - Hardware limits the amount of places kernel memory can be
        - No bounty for bypassing
    - 64 bit Linux KASLR gives **6 bits of entropy**
    - 64 bit Windows KASLR gives **13 bits of entropy**
    - Side channel attacks allowed KASLR to be trivially bypassed

KAISER

- Kernel Address Isolation to have Side-channels Efficiently Removed
  - Also called KPTI (Kernel Page Table Isolation)
- Essentially better KASLR
  - KAISER actually prevented Meltdown

Userspace/Kernel Communication

- The main method of communication (not only) is via syscalls
  - Syscall (0f 05) instruction
    - Basically jumps to kernel space
    - The kernel then figures out which syscall is being invoked and runs it (eax on linux)
  - Typically 100s of syscalls

Questions

The prior information is useful background for the rest of this, so ask any questions

After this is stuff more related to exploitation

Kernel Security

- What we **don't** want
  - Any information leakage
    - Could be used to defeat KASLR/KAISER
    - Could also just contain sensitive information
  - Any null pointers
    - It's not fun when a kernel dereferences an invalid pointer
  - Any unvalidated pointers
    - Corrupted pointers can lead to code execution
- What we will talk about
  - Race Conditions
  - Unvalidated Pointers
  - Infoleaks

Race Conditions

- Anyone see the issue in the following code?
- TOCTOU (Time of Check to Time of Use)

```
//This function can be called by any users. It executes only trusted binaries to run as root
//Trusted binaries are guaranteed to be safe to execute.
//filePath is a pointer to userspace memory that has the path of the file being executed.
int safeExecuteProgramAsRoot(char * filePath){
  if(!isValidFilePath(filePath){
    return INVALID_FILEPATH;
  }
  if(!isTrustedProgram(filePath)){
    return PROGRAM_UNTRUSTED;
  }
  executeProgramAsRoot(filePath);
  return SUCCESS;
}
```

Unvalidated Pointers

- Validate all Pointers before using them

```
//Takes a pointer provided by userspace to a buffer in userspace
void getKernelVersion(char * buffer){
    char[] version = “Stuart’s x86-64 Kernel Version 1.0131”;
    memcpy(buffer, version, sizeof(version));
}
```

Race Conditions

```
struct customString{
        char * buffer;
        int length;
}
```

- Read-After-Write

```
//Userspace provided output, bufferToUse pointers.
int getSystemVersion(customString * output, char * bufferToUse){
    if(!isSafePointer(output) && isSafePointer(bufferToUse)){
        return INVALID_PTR;
    }
    char[] version = “Stuart’s x86-64 Kernel Version 1.0131”;

    customString->buffer=bufferToUse;
    customString->length=strlen(version);

    memcpy(customString->buffer,strlen(version);
    return SUCCESS;
}
```

Infoleaks

```
int divide_numbers(int denom, int numerator, int * out){
    if(!isSafePointer(out)){
        return INVALID_PTR;
    }
    int result;
    if(denom != 0){
        result=denom/numerator;
    }
    *out=result;
    return SUCCESS;
}
```

Infoleaks

```
typedef struct resultStruct{
    uint8_t success;
    int result;
} resultStruct;
int divide_numbers(int denom, int numerator, resultStruct * out){
    if(!isSafePointer(out)){
        return INVALID_PTR;
    }
    resultStruct outStruct;
    outStruct.result=0; // No uninitialized memory!
    outStruct.success=0;
    if(denom != 0){
        outStruct.result=denom/numerator;
        outStruct.success=1;
    }
    memcpy(out,outStruct,sizeof(resultStruct));
    return SUCCESS;
}
```

Takeaways

- Unvalidated Pointers
  - Difficulty to spot: Easy
  - Difficulty to fix: Easy
  - Risk: Critical
- Race Conditions
  - Difficulty to spot: Medium
  - Difficulty to fix: Depends/Medium
  - Risk: High
- Infoleaks
  - Difficulty to spot: Hard
  - Difficulty to fix: Easy
  - Risk: Low (still an issue though)

Takeaways

- Kernel security is **really** hard.
- Linux example
  - Linux had a “put_user” function that copied data to userspace.
    - Same as isValidPointer in my code.
  - They also had “unsafe_put_user” which was a faster version.
  - In one of the syscalls (waitid), a developer accidentally just used “unsafe_put_user”.
    - Pretty easy to exploit vulnerability that was incredibly easy to access
- Windows example
  - One project (bochspwn reloaded) attempted to automate finding infoleak bugs
  - The project was able to find 29 separate infoleaks.
    - One of vulnerable functions leaked up to **6672** bytes

Takeaways

- Kernel security is **really** hard.
- Linux example
  - Linux had a “put_user” function that copied data to userspace.
    - Same as isValidPointer in my code.
  - They also had “unsafe_put_user” which was a faster version.
  - In one of the syscalls (waitid), a developer accidentally just used “unsafe_put_user”.
    - Pretty easy to exploit vulnerability that was incredibly easy to access
- Windows example
  - One project (bochspwn reloaded) attempted to automate finding infoleak bugs
  - The project was able to find 29 separate infoleaks.
    - One of vulnerable functions leaked up to **6672** bytes

Windows Kernel Information Disclosure Vulnerability	CVE-2017-8479	Mateusz Jurczyk of Google Project Zero
Windows Kernel Information Disclosure Vulnerability	CVE-2017-8480	Mateusz Jurczyk of Google Project Zero
Windows Kernel Information Disclosure Vulnerability	CVE-2017-8481	Mateusz Jurczyk of Google Project Zero
Windows Kernel Information Disclosure Vulnerability	CVE-2017-8482	• fanxiaocao and pjf of IceSword Lab , Qihoo 360 • Mateusz Jurczyk of Google Project Zero
Windows Kernel Information Disclosure Vulnerability	CVE-2017-8483	Mateusz Jurczyk of Google Project Zero
Win32k Information Disclosure Vulnerability	CVE-2017-8484	Mateusz Jurczyk of Google Project Zero
Windows Kernel Information Disclosure Vulnerability	CVE-2017-8485	• fanxiaocao and pjf of IceSword Lab , Qihoo 360 • Mateusz Jurczyk of Google Project Zero
Windows Kernel Information Disclosure Vulnerability	CVE-2017-8488	Mateusz Jurczyk of Google Project Zero
Windows Kernel Information Disclosure Vulnerability	CVE-2017-8489	Mateusz Jurczyk of Google Project Zero
Windows Kernel Information Disclosure Vulnerability	CVE-2017-8490	
Windows Kernel Information Disclosure Vulnerability	CVE-2017-8491	
Windows Kernel Information Disclosure Vulnerability	CVE-2017-8492	

Windows Kernel Information Disclosure Vulnerability	CVE-2017-0175
Windows Kernel Information Disclosure Vulnerability	CVE-2017-0220
Win32k Information Disclosure Vulnerability	CVE-2017-0245
Windows Kernel Information Disclosure Vulnerability	CVE-2017-0258
Windows Kernel Information Disclosure Vulnerability	CVE-2017-0259

Windows Kernel Information Disclosure Vulnerability	CVE-2017-0167

Windows Kernel Information Disclosure Vulnerability	CVE-2017-0299	Mateusz Jurczyk of Google Project Zero
Windows Kernel Information Disclosure Vulnerability	CVE-2017-0300	Mateusz Jurczyk of Google Project Zero
Windows Kernel Information Disclosure Vulnerability	CVE-2017-8462	Mateusz Jurczyk of Google Project Zero
Windows Kernel Information Disclosure Vulnerability	CVE-2017-8469	Mateusz Jurczyk of Google Project Zero
Win32k Information Disclosure Vulnerability	CVE-2017-8470	• fanxiaocao and pjf of IceSword Lab, Qihoo 360 • Mateusz Jurczyk of Google Project Zero
Win32k Information Disclosure Vulnerability	CVE-2017-8471	Mateusz Jurczyk of Google Project Zero
Win32k Information Disclosure Vulnerability	CVE-2017-8472	Mateusz Jurczyk of Google Project Zero
Win32k Information Disclosure Vulnerability	CVE-2017-8473	Mateusz Jurczyk of Google Project Zero
Windows Kernel Information Disclosure Vulnerability	CVE-2017-8474	• fanxiaocao and pjf of IceSword Lab , Qihoo 360 • Mateusz Jurczyk of Google Project Zero
Win32k Information Disclosure Vulnerability	CVE-2017-8475	Mateusz Jurczyk of Google Project Zero
Windows Kernel Information Disclosure Vulnerability	CVE-2017-8476	• fanxiaocao and pjf of IceSword Lab , Qihoo 360 • Mateusz Jurczyk of Google Project Zero
Win32k Information Disclosure Vulnerability	CVE-2017-8477	Mateusz Jurczyk of Google Project Zero

Questions?

- Looking for input on what to cover in the future
  - Binary Exploitation (Heap)
  - Low Level Stuff (Like this!.) (Maybe talk about pipelining and CPUs.)
  - Reverse Engineering (Hard to create a lot of content for.)