Chapter 9. Kernel Security

Prof. Jaeseung Choi
Dept. of Computer Science and Engineering
Sogang University

Topics

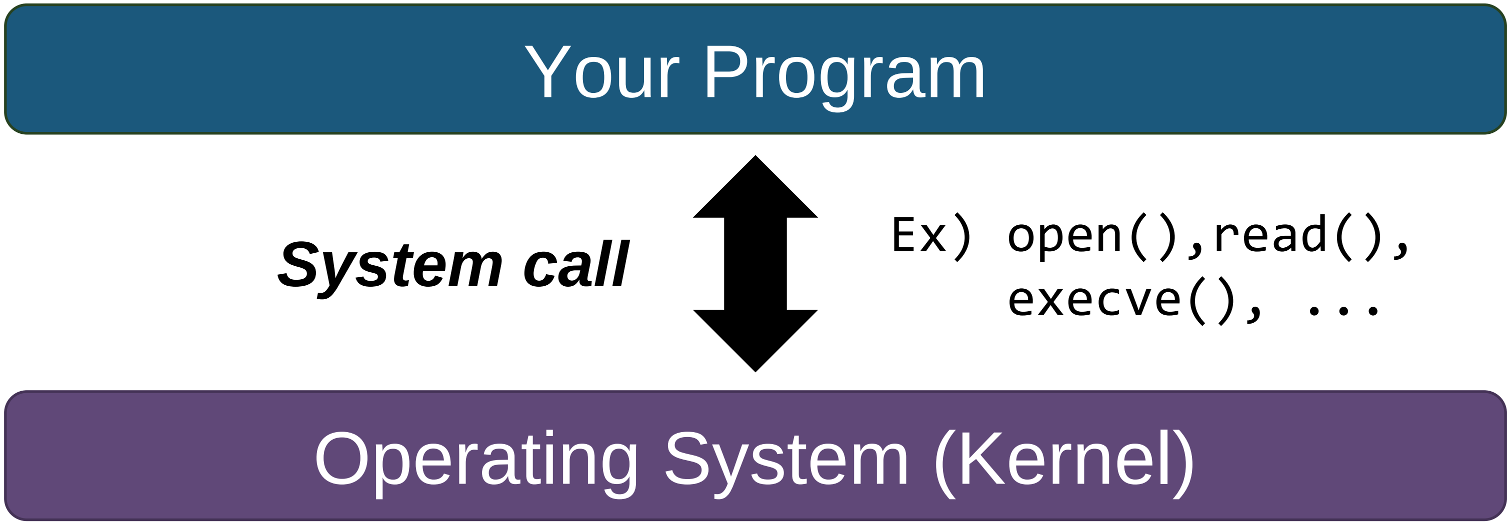
- **Software vulnerabilities in OS kernel**
  - Using user-provided pointer without check
  - Double fetch
  - Null dereference
  - Memory disclosure due to alignment

From User Code to Kernel Code

- **So far, we've discussed various software vulnerabilities**
- **We have assumed user code during this discussion**
  - Usually, what you write and run is user-level code
- **Now let's take a look into the kernel code**
  - The core part of operating system
  - Kernel code is executed in privileged mode
  - In **`PintOS`** project, you have written (or will write) kernel code
  - What kind of security issues can rise here?

Review: System Call

- **Recall that the program code you wrote is not allowed to access the system resources directly**
  - Ex) Reading a file or writing to a file
- **To do that, you must make a request via *system calls***
  - Then, the OS will do the task for you



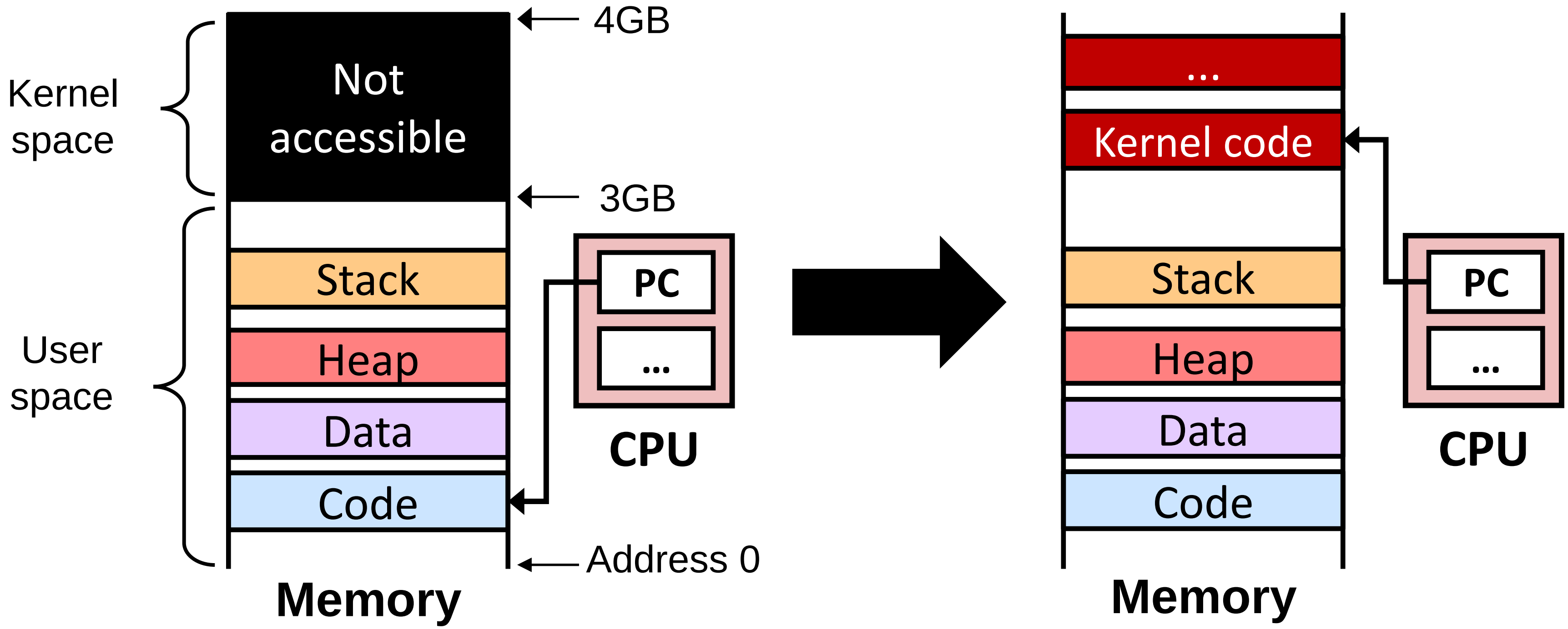
Review: `syscall` Instruction

- **At assembly level, we must use `syscall` instruction**
- **Similar to function call, but the function is in kernel**
- **When the code sets particular registers properly and executes this instruction, a system call is invoked**
  - Now, let's dig a little bit deeper about the system call

```
...
mov    $0, %rsi       # %rsi must contain flag (option)
mov    ..., %rdi      # %rdi must point to filename string
mov    $0x2, %rax     # System call ID of open() is 2
syscall
```

What happens at this moment?

- **Upon the execution of `syscall` instruction:**
  - The CPU mode changes to privileged mode (kernel mode)
  - Kernel-level memory space becomes accessible
  - Program counter moves to pre-defined address in kernel code



Note: These address range values are used for **32-bit Linux**

System Call Handler in Kernel

- **Now it's the turn of system call handler in kernel code**
  - It examines the system call arguments passed from the user
  - Ex) **`open`** system call: What is the name of file that the user requests to open? Does this user have a proper permission?
  - Ex) **`read`** system call: What is the input file descriptor? Where does the user want the file content to be stored?
- **Note: To do these things, system call handler code will have to access both the user space and kernel space**

Vulnerabilities in Kernel

- **Most of the software vulnerabilities that we learned until now can also occur in the kernel code**
- **For example, buffer overflow can occur and corrupt the critical data stored in the kernel memory**
  - Ex) **`strcpy()`** below copies string without any length check
- **Kernel must not trust user-provided arguments**
  - An attacker may invoke malicious system calls and exploit kernel vulnerabilities, in order to obtain the **`root`** privilege

```
// User-provided arguments 'filename' and 'flags'
int open_handler(char *filename, int flags) {
  char buf[32];
  strcpy(buf, filename); // Buffer overflow
  ...
```

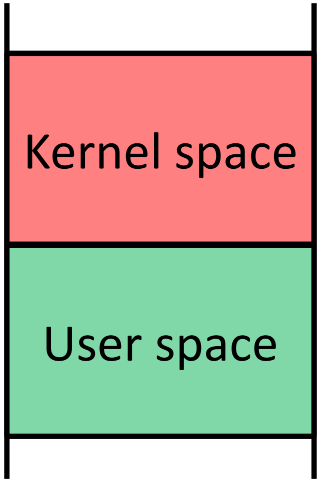
OK, then is this all? (Buffer overflow, format string bug, use-after-free, …)

No, there are other unique kind of bugs that can occur in the kernel code

Danger in User-Provided Pointer

- **Let's consider the system call handler for `read()`**
  - Assume that the file content is first loaded into kernel space
  - In the end, this content must be copied to the user's buffer (**`buf`**)
  - In normal case, **`buf`** must be an address in user space
- **But what if the provided `buf` is a kernel-space address?**
  - **`memcpy()`** may overwrite and corrupt kernel memory

```
read_handler(int fd, void *buf, size_t n) {
  ... // Read in the file content
  memcpy(buf, file_content, n);
  return 0;
}
```



Review: `PintOS` Manual

- **In fact, if you have taken the *Operating System* course, you must be already familiar with all these issues**
- **The `PintOS` manual also explains this vulnerability**
  - Section 3.1.5 Accessing User Memory
  - It also gives you a solution: check whether the user-provided pointer belongs to user space (address below 3GB)

3.1.5 Accessing User Memory

As part of a system call, the kernel must often access memory through pointers provided by a user program. The kernel must be very careful about doing so, because the user can pass a null pointer, a pointer to unmapped virtual memory, or a pointer to kernel virtual address space (above `PHYS_BASE`). All of these types of invalid pointers must be rejected

The second method is to check only that a user pointer points below `PHYS_BASE`, then dereference it. An invalid user pointer will cause a "page fault" that you can handle by

Real-world Example

- **CVE-2020-0792 found in *Windows* kernel**
  - System call handler performs a series of check on user inputs
  - **`IsKernelSpace()`** checks the range of a pointer
  - But this check can be skipped if odd-number length is provided

```
SyscallHandler(struct Str* arg) {
  wchar_t *buf = arg->buf;
  ushort len = arg->len;
  if (len & 1) {
    LogError(...);
  }
  else if (IsKernelSpace(buf)) { // Why "else-if"?
    return;
  }
  ... // Access 'buf' here.
```

Quiz

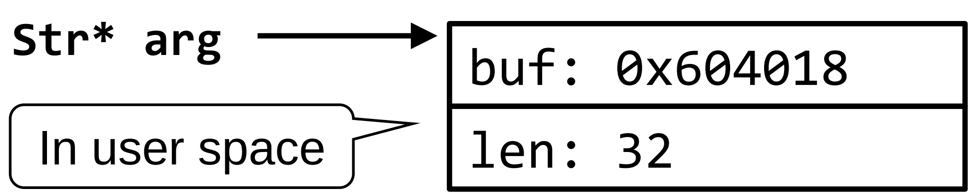
- **Consider the following system call handler in kernel**
  - At first glance, this code seems to be safe
  - But there is a potential vulnerability here: what is it?

```
SyscallHandler(struct Str* arg) {
  char kbuf[128];
  if (IsKernelSpace(arg->buf) || arg->len >= 128) {
    return;
  }
  memcpy(kbuf, arg->buf, arg->len);
  ...
```

Double Fetch (Race Condition)

- **The code fetches the `buf` (and `len`) field twice**
  - **`arg->buf`** or **`arg->len`** may change between the TOC and TOU
  - Assume that an attacker creates two threads **`t1`** and **`t2`**
    - First, **`t1`** invokes system call and passes **`IsKernelSpace()`**

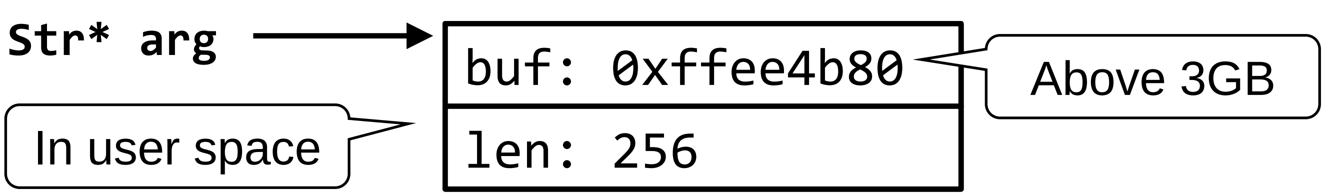
```
if (IsKernelSpace(arg->buf) || arg->len >= 128) {
  return;
}
memcpy(kbuf, arg->buf, arg->len);
```



Double Fetch (Race Condition)

- **The code fetches the buf (and `len`) field twice**
  - **`arg->buf`** or **`arg->len`** may change between the TOC and TOU
  - Assume that an attacker creates two threads **`t1`** and **`t2`**
    - First, **`t1`** invokes system call and passes **`IsKernelSpace`**`()`
    - Then the execution switches to thread **`t2`**, which updates **`arg`**
    - Finally, **`t1`** resumes and uses the updated field in **`arg`**

```
if (IsKernelSpace(arg->buf) || arg->len >= 128) {
  return;
}
memcpy(kbuf, arg->buf, arg->len);
```



Preventing Double Fetch

- **How can we avoid double fetch bug?**
- **Copy the user-provided inputs to the kernel space first (e.g., to local variables), and then perform validation**
  - Now, the attacker cannot change the value of **`buf`** and **`len`** after passing the validation (TOCTOU not possible anymore)

```
SyscallHandler(struct Str* arg) {
  char kbuf[128];
  char *buf = arg->buf; // Copy arguments to local variable
  int len = arg->len;
  if (IsKernelSpace(buf) || len >= 128) { // Validation
    return;
  }
  memcpy(kbuf, buf, len);
  ...
```

NULL Dereference in User Code

- **Now, let's forget about the kernel for a while and think about this question:**
  - **In user code**, can NULL dereference raise a security issue?
- **If your application uses NULL pointer, it will crash**
  - So this may result in a denial-of-service attack (if the application was running an important service)
- **Can hackers do something more than denial-of-service?**
  - In most cases, NULL dereference cannot lead to arbitrary code execution or memory disclosure
  - NULL is just an invalid memory address, so using such pointer will immediately raise a crash

NULL Dereference in Kernel

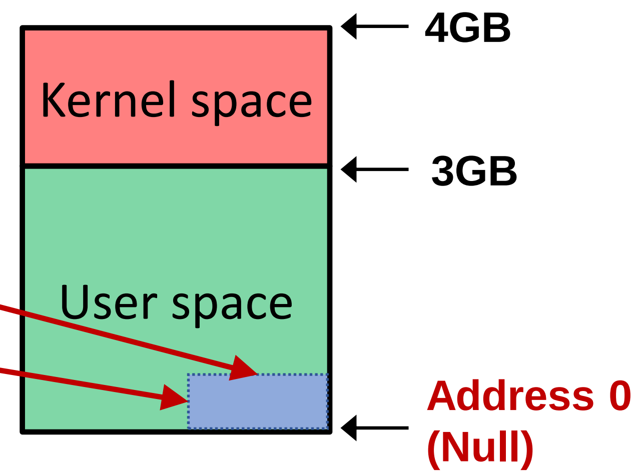
- **NULL dereference in kernel code a totally different story**
- **Assume that the kernel code is doing certain operation**
  - Ex) Writing a secret key value to a pointer
  - Ex) Fetching and calling a function pointer
  - What happens if these pointers (**`p`**, **`fptr_table[i]`**) are NULL?
  - Note that these pointers are ***not*** provided from user (via syscall)

```
int x, *p = NULL;
if (...) {
  p = &x;
}
*p = secret_key; // What if p is NULL?
...
fptr_table[i](100); // What if fptr_table[i] is NULL?
```

NULL Pointer ∈ User Space

- **In kernel security, NULL pointer is not just an *invalid address*: it is an address that belongs to the user space**
  - What if the user can allocate memory at address 0 (zero)?
  - You cannot do this with **`malloc()`**, but it is possible with **`mmap()`**
    - To be precise, it ***was*** possible (details in the next page)
  - Then the kernel will store the secret key to address 0 (user space)
  - Or the kernel may even execute the code in the user space

```
*p = secret_key;
...
fptr_table[i](100);
```



Preventing NULL Dereference

- **At software-level**
  - Prevents memory allocation at address 0
  - Configuration parameter **`mmap_min_addr`**
    - User cannot allocate memory at address below this range
- **At hardware-level**
  - SMEP (Supervisor Mode **Execution** Prevention): prevents CPU running in the kernel mode from **executing** user-space code
  - SMAP (Supervisor Mode **Access** Prevention): prevents CPU running in the kernel mode from **accessing** user-space data
    - Temporarily disabled when a system call handler has to fetch data from the user space memory

Lessons

- **The complex threat model of kernel security gives a rise to new interesting types of vulnerabilities**
  - Ex) NULL dereference is not a serious vulnerability in user application code, but it is a critical vulnerability in kernel code
- **Understanding certain kind of security issues requires a deep understanding on the internals of computer**
  - Ex) To understand the memory disclosure due to alignment, you should even know the behavior of a compiler

Appendix:

1 more subtle kernel vulnerability

Quiz

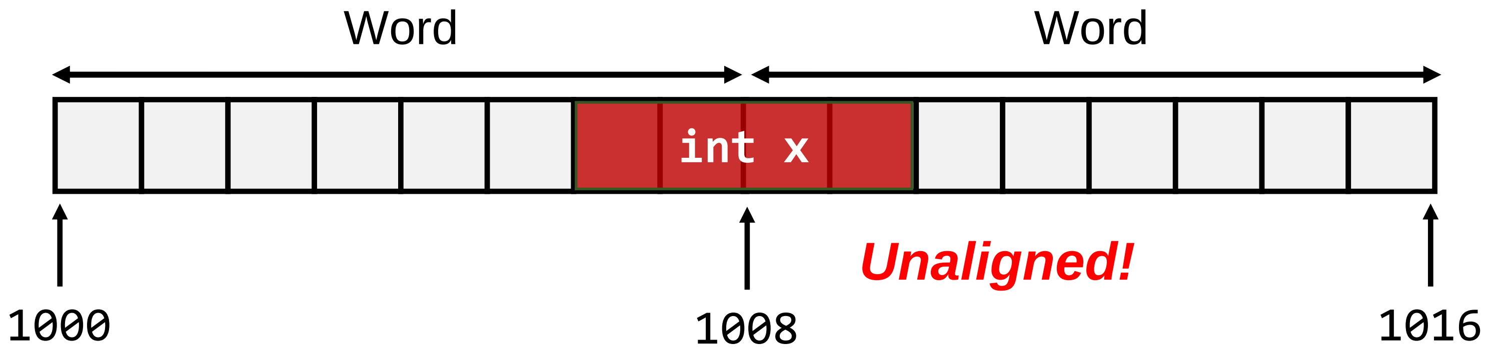
- **Consider the following system call handler in kernel**
  - At first glance, this code seems to be safe
  - But there is a potential vulnerability here: what is it?

```
gettime_handler(struct Time *arg) {
  struct Time t;
  if (is_kernel_space(arg)) {
    return;
  }
  // Set each field of the local struct
  t.second = ...;
  t.nano_sec = ...;
  memcpy(arg, &t, sizeof(struct Time));
}
```

```
struct Time {
  int second;
  long nano_sec;
};
```

Review: Alignment

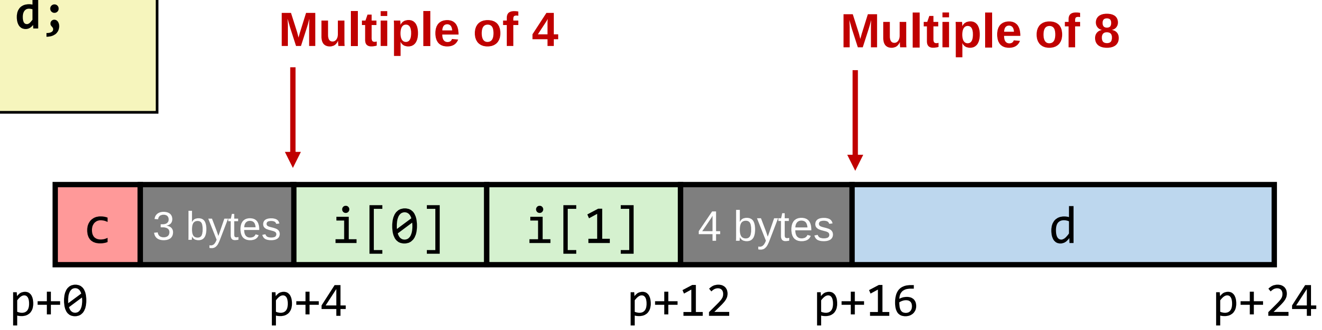
- **Assume a data type that requires *K* bytes of memory**
  - "Aligned" means that its memory address is multiple of *K*
  - Ex) In order to be aligned, **`int`** type (4-byte) variable must be placed at an address that is multiple of 4
- **Motivation for alignment**
  - At hardware level, memory is accessed by chunk of bytes (word)
  - Inefficient to load or store data that spans multiple words



Review: Alignment of Structure

- **Alignment within the structure**
  - Each ***field*** must be placed at an ***aligned offset***
- **Example:** `struct S1`
  - Compiler will insert ***padding*** (unused space) between the fields

```
struct S1 {
  char c;
  int i[2];
  double d;
};
```



Alignment and Memory Disclosure

- **Let's return to the example code in the previous quiz**
  - The padding bytes between the two fields are not initialized
  - Uninitialized bytes in the stack will be copied to user space
  - If unlucky, it will disclose sensitive data in kernel memory

```c
gettime_handler(struct Time *arg) {
  struct Time t;
  if (is_kernel_space(arg)) {
    return;
  }
  // Set each field of the local struct
  t.second = ...;
  t.nano_sec = ...;
  memcpy(arg, &t, sizeof(struct Time));
}
```

```c
struct Time {
  int second;
  long nano_sec;
};
```