Kernel Security: how2rootkit

Today's Agenda

Fun With Linking

Dynamic Linking, Symbol Interposition, and Hooking

- How shared libraries are built and loaded (review)
- How the dynamic linker resolves symbols at runtime (hopefully review?)
- How to abuse this to intercept function calls
- Three types of hooking and one bypass

Agenda

Part 1: Five Examples (hands-on)

- Ex1: Basic shared library
- Ex2: Constructor / destructor
- Ex3: dlopen / dlsym
- Ex4: RTLD_DEFAULT / RTLD_NEXT
- Ex5: LD_PRELOAD interposition

Part 2: PLT/GOT Internals

- How calls through shared libraries actually work on AArch64

Part 3: Hooking Techniques + HW Preview

- Three hooking levels + direct syscall bypass

Static vs Dynamic Linking

Static linking (`-static`)

- All library code copied into the binary at link time
- Larger binary, no runtime dependencies
- Harder to update (recompile everything)

Dynamic linking (default)

- Binary references shared objects (`.so` files)
- Smaller binary, libraries loaded at runtime by `ld-linux`
- Libraries shared across processes in memory
- Can be trivially intercepted... which is the whole point of today

Ex1: Basic Shared Library

Build a minimal `.so` and link a program against it

Files:

```
ex1_basic_so/
    mylib.h      # header with function declaration
    mylib.c      # library implementation
    main.c       # program that uses the library
```

flags: `-fPIC`, `-shared`, `-L`, `-l`, `LD_LIBRARY_PATH`

Ex1: mylib.c / mylib.h

mylib.h:

```
#ifndef MYLIB_H
#define MYLIB_H

void greet(const char *name);

#endif
```

mylib.c:

```
#include <stdio.h>
#include "mylib.h"

void greet(const char *name)
{
    printf("Hello, %s! Greetings from libmylib.so\n", name);
}
```

Ex1: Building and Running

main.c:

```c
#include <stdio.h>
#include "mylib.h"

int main(void)
{
    printf("ex1: Basic Shared Library\n");
    greet("Student");
    return 0;
}
```

Build and run :

```
gcc -fPIC -shared -o bin/libmylib.so ex1_basic_so/mylib.c
gcc -Iex1_basic_so -o bin/ex1_main ex1_basic_so/main.c -Lbin -lmylib

$ LD_LIBRARY_PATH=./bin ./bin/ex1_main
ex1: Basic Shared Library
Hello, Student! Greetings from libmylib.so
```

Ex1: What Each Flag Does

Flag	Purpose
`-fPIC`	Position-Independent Code -- required for shared libraries. Code uses relative addresses so it can be loaded at any address.
`-shared`	Produce a shared object (`.so`) instead of an executable
`-L<dir>`	Add `<dir>` to the library search path at **link time**
`-l<name>`	Link against `lib<name>.so` (e.g., `-lmylib` -> `libmylib.so`)
`LD_LIBRARY_PATH`	Colon-separated list of directories to search for `.so` files at **runtime**

Library Search path

Without `LD_LIBRARY_PATH` (or installing to `/usr/lib`), the dynamic linker won't find `libmylib.so` and the program fails with: (...unless of course it is in the same directory :)

```
error while loading shared libraries: libmylib.so:
cannot open shared object file: No such file or directory
```

Ex2: Constructor / Destructor

`__attribute__((constructor))` -- function runs automatically when the library is loaded, **before `main()`**

`__attribute__((destructor))` -- function runs when the library is unloaded, **after `main()` returns**

These are ELF `.init_array` / `.fini_array` entries -- the dynamic linker calls them, not your code.

Provides a mehcanism for automatically running code on load. This is how we install hooks before the target binary gets run

Ex2: initlib.c + main.c

initlib.c:

```c
#include <stdio.h>

__attribute__((constructor))
static void lib_init(void)
{
    printf("[initlib] constructor: library loaded!\n");
}

__attribute__((destructor))
static void lib_fini(void)
{
    printf("[initlib] destructor: library unloading!\n");
}

void do_something(void)
{
    printf("[initlib] do_something() called\n");
}
```

Ex2 Continued

main.c:

```c
extern void do_something(void);
int main(void) {
    printf("ex2: Constructor / Destructor\n");
    printf("[main] inside main()\n");
    do_something();
    printf("[main] leaving main()\n");
    return 0;
}
```

Output -- notice the execution order:

```
[initlib] constructor: library loaded!
ex2: Constructor / Destructor
[main] inside main()
[initlib] do_something() called
[main] leaving main()
[initlib] destructor: library unloading!
```

Ex2:

Constructors run **before any application code** -- this makes them the perfect mechanism for installing hooks.

The homework uses exactly this pattern:

```
__attribute__((constructor)) void install_hooks(void) {
    // Patch GOT entries, rewrite PLT stubs, etc.
    // All done before main() ever runs.
}
```

HINT: if you can get a library loaded (via `LD_PRELOAD`, `-l`, or `dlopen`), its constructors run automatically.

Ex3: Explicit Dynamic Linking

Instead of linking at compile time, you can load libraries **at runtime** with the `dlfcn.h` API:

Function	Purpose
`dlopen(path, flags)`	Load a shared library, returns a handle
`dlsym(handle, name)`	Look up a symbol (function/variable) by name
`dlclose(handle)`	Unload the library
`dlerror()`	Get a human-readable error string

Compile with `-ldl` to link against libdl.

This is the **plugin pattern** -- load code you didn't know about at compile time.

Ex3: The Plugin Library

mathlib.c -- a simple library with two exported functions:

```
#include <stdio.h>

int square(int x)
{
    printf("[mathlib] computing square(%d)\n", x);
    return x * x;
}

int cube(int x)
{
    printf("[mathlib] computing cube(%d)\n", x);
    return x * x * x;
}
```

Build (shared library only -- no `-l` at link time):

```
gcc -fPIC -shared -o bin/libmathlib.so ex3_dlopen/mathlib.c
```

Ex3: The dlopen/dlsym Pattern

```c
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

int main(void)
{
    printf("ex3: Explicit Dynamic Linking (dlopen / dlsym)\n");

    void *handle = dlopen("./bin/libmathlib.so", RTLD_LAZY);
    if (!handle) {
        fprintf(stderr, "dlopen failed: %s\n", dlerror());
        return 1;
    }

    dlerror();  /* clear any existing error */

    int (*square)(int) = dlsym(handle, "square");
    char *err = dlerror();
    if (err) { fprintf(stderr, "dlsym(square) failed: %s\n", err); ... }

    int (*cube)(int) = dlsym(handle, "cube");
    err = dlerror();
    if (err) { fprintf(stderr, "dlsym(cube) failed: %s\n", err); ... }

    int val = 7;
    printf("square(%d) = %d\n", val, square(val));
```

Ex3: Output

Output:

```
ex3: Explicit Dynamic Linking (dlopen / dlsym)
[mathlib] computing square(7)
square(7) = 49
[mathlib] computing cube(7)
cube(7)   = 343
Library closed.
```

RTLD_LAZY vs RTLD_NOW:

- `RTLD_LAZY` -- resolve symbols on first call (lazy binding)
- `RTLD_NOW` -- resolve all symbols immediately at `dlopen` time

Once you can look up any symbol by name at runtime, you can also **replace** them. `dlsym` gives you the address of any function -- and addresses can be overwritten.

Ex4: Symbol Resolution

`dlsym` accepts special pseudo-handles instead of a `dlopen` handle:

Handle	Meaning
`RTLD_DEFAULT`	Search the **global** symbol table (all loaded libraries) in load order
`RTLD_NEXT`	Search libraries loaded **after** the caller -- skip yourself

- `RTLD_DEFAULT` lets you find any function currently loaded in memory
- `RTLD_NEXT` lets you **wrap** a function: provide your own version, then call through to the original

Ex4: Finding printf via RTLD_DEFAULT

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <dlfcn.h>

int main(void)
{
    printf("ex4: Symbol Resolution (RTLD_DEFAULT / RTLD_NEXT)\n\n");

    /* Use RTLD_DEFAULT to find printf in the global symbol table */
    int (*my_printf)(const char *, ...) = dlsym(RTLD_DEFAULT, "printf");
    if (!my_printf) {
        fprintf(stderr, "dlsym failed: %s\n", dlerror());
        return 1;
    }

    printf("Found printf via RTLD_DEFAULT at address: %p\n",
           (void *)my_printf);
    my_printf("Calling printf through function pointer: it works!\n");

    printf("\nputs() is intercepted by wraplib.so via RTLD_NEXT:\n");
    puts("Hello from main!");

    return 0;
}
```

`my_printf` is now a raw function pointer to libc's `printf` -- you can call it, store it, pass it around.

Ex4: Intercepting puts() with RTLD_NEXT

wraplib.c -- overrides `puts` and forwards to the original:

```
#define _GNU_SOURCE
#include <stdio.h>
#include <dlfcn.h>

int puts(const char *s)
{
    int (*real_puts)(const char *) =
        dlsym(RTLD_NEXT, "puts");
    if (!real_puts) {
        fputs("[wraplib] dlsym failed!\n",
              stderr);
        return EOF;
    }

    real_puts("[wraplib] intercepted puts:");
    return real_puts(s);
}
```

This works because `wraplib.so` is linked **before** libc, so its `puts` is found first. RTLD_NEXT skips `wraplib.so` and finds libc's `puts`.

Output:

```
ex4: Symbol Resolution
  (RTLD_DEFAULT / RTLD_NEXT)

Found printf via RTLD_DEFAULT
  at address: 0x400000910410
Calling printf through function
  pointer: it works!

puts() is intercepted by
  wraplib.so via RTLD_NEXT:
[wraplib] intercepted puts:
Hello from main!
```

The key: **link order determines which symbol wins**. The first library in the link map with a matching symbol name gets called.

Ex5: LD_PRELOAD Interposition

`LD_PRELOAD` is an environment variable that forces the dynamic linker to load a library **before all others**.

This means your library's symbols **override everything** -- including libc.

No recompilation needed. No source code needed. Just:

```
LD_PRELOAD=./hook.so ./target
```

The target program has no idea its functions have been replaced.

Ex5: The main.c

A completely normal program that calls `puts` three times:

```c
#include <stdio.h>

int main(void)
{
    puts("ex5: LD_PRELOAD Interposition");
    puts("This is a normal puts call.");
    puts("Nothing unusual here.");
    return 0;
}
```

Compiled normally -- no special flags, no awareness of any hook:

```
gcc -o bin/ex5_main ex5_preload/main.c
```

Ex5: preload.c

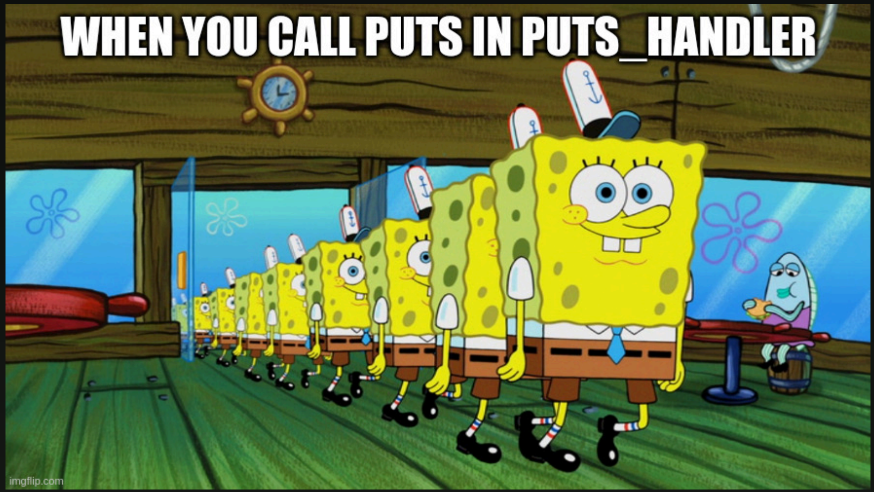
```c
#define _GNU_SOURCE
#include <stdio.h>
#include <dlfcn.h>

int puts(const char *s)
{
    /* Get the real puts -- do NOT call puts() here
       or infinite recursion! */
    int (*real_puts)(const char *) =
        dlsym(RTLD_NEXT, "puts");

    real_puts("I WAS HERE FIRST!");
    return real_puts(s);
}
```

```
gcc -fPIC -shared -o bin/libpreload.so ex5_preload/preload.c -ldl
```

Q: What happens if you call `puts` directly in our puts handler?

puts recursion

Warning: if your hook calls `puts()` without going through `real_puts`, it calls **itself** -- infinite recursion, stack overflow, segfault. Always use `RTLD_NEXT` to get the original.



Part 2: How Dynamic Linking Actually Works

We've seen that symbol interposition works. But **why** does it work?

The compiler can't know where `puts` lives in memory -- libc gets loaded at a different address every time (ASLR).

So the compiler generates **indirect calls** through two structures:

- **PLT** (Procedure Linkage Table) -- executable code stubs
- **GOT** (Global Offset Table) -- writable function pointers

Every call to a shared library function goes through PLT -> GOT -> actual function.

Understanding this mechanism is the essential to the homework.

What Happens When You Call puts()?

When you write `puts("hello")` in C, the compiler does **not** generate a direct call to libc's `puts`.

Instead it generates:

```
bl    puts@plt          ; branch-and-link to PLT stub
```

The `@plt` suffix means "go through the PLT stub, not the actual function."

The PLT stub is a small piece of code in **your** binary that knows how to find the real function via the GOT.

This is the indirection that makes dynamic linking (and hooking) possible.

PLT Stub on AArch64

Each PLT entry is a 3-instruction stub:

```
puts@plt:
    adrp   x16, GOT_PAGE        ; load high bits of GOT entry address
    ldr    x17, [x16, GOT_OFF]  ; load function pointer from GOT
    br     x17                   ; jump to whatever GOT points to
```

- The PLT stub contains **no function code** -- it just loads a pointer and jumps
- The pointer it loads comes from the **GOT**, which is a writable data section
- `x16/x17` are the intra-procedure-call scratch registers on AArch64 -- the linker owns them

GOT: Global Offset Table

The `.got.plt` section is an array of function pointers, one per imported function:

```
.got.plt:
    GOT[0]  = address of _dl_runtime_resolve  (resolver)
    GOT[1]  = link_map pointer
    GOT[2]  = address of _dl_runtime_resolve
    GOT[3]  = &puts  (or resolver stub, until first call)
    GOT[4]  = &printf (or resolver stub, until first call)
    ...
```

- The GOT is **writable** at runtime (so the resolver can patch it)
- Depending on mitigation, it might **not** be writable after being set (RELRO).
- Each entry is a **function pointer** that the PLT blindly jumps to
- Before first call: points back to resolver code
- After first call: points to the real libc function

Lazy Binding: First Call vs Second Call

First call to `puts()`:

```
.text           .plt              .got.plt         ld.so
  |               |                  |               |
  | bl puts@plt   |                  |               |
  |-------------->| adrp/ldr/br      |               |
  |               |----------------->| (unresolved)   |
  |               |                  |--- points to ->| _dl_runtime_resolve
  |               |                  |               | looks up "puts"
  |               |                  |<-- patches ---| GOT[n] = &real_puts
  |               |                  |               |
  |<------------------------------------------------ | jumps to real puts
```

Second call to `puts()`:

```
.text           .plt              .got.plt         libc
  |               |                  |               |
  | bl puts@plt   |                  |               |
  |-------------->| adrp/ldr/br      |               |
  |               |----------------->| (resolved!)   |
  |               |                  |--- points to->| real puts
  |<-------------------------------------------------|
```

No resolver involved on subsequent calls -- the GOT now has the real address.

Lazy Binding: Before and After

```
First call
+----------+
| .text    |
|  bl puts@plt ----+
+----------+       |
                    v
+----------+   +----------+
| .plt     |   | .got.plt |
| adrp     |   |          |
| ldr x17  |-->| GOT[n]:  |
| br  x17  |   | &resolver|----> ld.so
+----------+   +----------+      resolves
                                  + patches
```

```
After first call
+----------+
| .text    |
|  bl puts@plt ----+
+----------+       |
                    v
+----------+   +----------+
| .plt     |   | .got.plt |
| adrp     |   |          |
| ldr x17  |-->| GOT[n]:  |
| br  x17  |   | &puts    |----> libc puts
+----------+   +----------+      (direct)
```

PLT/GOT: Hooking

1. **The GOT is/can be made writable** -- it must be (at some point), so the resolver can patch it
2. **GOT entries are at known offsets** -- `readelf -r` shows every relocation
3. **The PLT implicitly trusts GOT **

Hooking Strategies

Level	What you change	Section modified
1. LD_PRELOAD	Symbol binding order	Nothing -- linker resolves your symbol first
2. GOT patching	Function pointer in `.got.plt`	`.got.plt` (data)
3. PLT patching	Instructions in `.plt`	`.plt` (code)

Part 3: Homework

1. **LD_PRELOAD / Binding Hook** -- override symbols by name
2. **GOT Patching** -- overwrite GOT function pointers at runtime
3. **PLT Stub Patching** -- rewrite PLT instructions to jump to your code
4. **Direct Syscall Bypass** -- skip all of the above with `svc #0`

Target: block access to `/tmp/protected/` by hooking `openat` and `unlinkat`

Bypass: write to `/tmp/protected/hacked.txt` despite hooks being loaded

Level 1: HW Part 1)

The simplest approach -- define your own version of each function:

```
/* binding_hook.c */
#define _GNU_SOURCE
#include <dlfcn.h>
#include <errno.h>

typedef int (*orig_openat_type)(int, const char *, int, mode_t);
static orig_openat_type real_openat = NULL;

#define PROTECTED_DIR "/tmp/protected"

__attribute__((constructor)) void install_hooks(void) {
    real_openat = dlsym(RTLD_NEXT, "openat");
    // ... same for open, open64, unlink, unlinkat
}

int openat(int dirfd, const char *pathname, int flags, ...) {
    if (strncmp(pathname, PROTECTED_DIR, ...) == 0) {
        errno = EACCES;
        return -1;
    }
    return real_openat(dirfd, pathname, flags, mode);
}
```

Easiest to implement. Also easiest to bypass (don't use libc).

Level 2: GOT Patching (HW Part 2)

Instead of relying on link order, **overwrite the GOT entry** directly:

1. `dl_iterate_phdr` -- iterate loaded ELF objects
2. Find the main executable's dynamic segment
3. Parse `DT_JMPREL` to find `R_AARCH64_JUMP_SLOT` relocations
4. Match symbol name (`openat`, `unlinkat`)
5. `mprotect` the GOT page as writable
6. Overwrite `GOT[n]` with your detour address

```
Before patching:
+----------+   +----------+
| .plt     |   | .got.plt |
| adrp     |   |          |
| ldr x17 -|-->| GOT[n]:  |
| br  x17  |   | &openat  |--> libc
+----------+   +----------+

After patching:
+----------+   +----------+
| .plt     |   | .got.plt |
| adrp     |   |          |
| ldr x17 -|-->| GOT[n]:  |
| br  x17  |   | &hacked  |--> your
+----------+   +----------+
```

got hook

Before:

```
GOT[n]: 0x7fff13e7a230  (libc openat)
```

After:

```
GOT[n]: 0x7fff20001040  (hacked_openat)
```

The PLT stub is unchanged -- it still does `ldr x17, [GOT+off]` and `br x17`. But now the pointer leads to your function.

Part 3: PLT Stub Patching

Instead of changing where the GOT points, **rewrite the PLT stub instructions** themselves:

Original PLT stub (3 instructions, 12 bytes):

```
adrp   x16, GOT_PAGE       ; 4 bytes
ldr    x17, [x16, GOT_OFF] ; 4 bytes
br     x17                  ; 4 bytes
```

Replaced PLT stub (3 instructions + embedded address, 16 bytes):

```
ldr    x16, #8             ; 4 bytes -- load from PC+8
br     x16                  ; 4 bytes -- jump to detour
.quad  &hacked_openat       ; 8 bytes -- embedded address
```

hw3 hints

Helper functions you need to implement:

- `encode_ldr_literal_x16()` -- encode `ldr x16, #8` as a 32-bit instruction
- `encode_br_x16()` -- encode `br x16` as a 32-bit instruction

Requires `mprotect` on `.plt` to make it writable (it's normally read-execute).

ELF Structures You Need to Parse

Structure	What it describes	Key fields
`Elf64_Dyn`	Dynamic section entries	`d_tag` (DT_JMPREL, DT_SYMTAB, DT_STRTAB), `d_un.d_ptr`
`Elf64_Rela`	Relocation entries	`r_offset` (GOT address), `r_info` (symbol index + type)
`Elf64_Sym`	Symbol table entries	`st_name` (offset into strtab), `st_value`
`struct dl_phdr_info`	Per-object info from `dl_iterate_phdr`	`dlpi_addr` (base), `dlpi_phdr`, `dlpi_phnum`

helper commands

```
readelf -W -d bin/tester | grep -E 'JMPREL|SYMTAB|STRTAB|PLTRELSZ'
readelf -W -r bin/tester | grep -E 'JUMP_SLOT'
readelf -W -S bin/tester | grep -E '\.plt'
aarch64-linux-gnu-objdump -d -j .plt bin/tester
```

The Bypass: Direct Syscalls (HW Part 4)

All three hooking techniques intercept at the **userland library level**. The kernel doesn't know or care.

To bypass all hooks, issue the syscall instruction directly:

```c
/* bypass_syscall.c */
#include "syscall_utils.h"

#define TARGET_FILE "/tmp/protected/hacked.txt"
#define PAYLOAD "1337h4x0r"

static int write_payload(void) {
    int fd = sys_openat(AT_FDCWD, TARGET_FILE,
                        O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (fd < 0) return 1;
    sys_write(fd, PAYLOAD, sizeof(PAYLOAD) - 1);
    sys_close(fd);
    return 0;
}
```

Symbol vs syscall

- note that `openat` often points to a syscall wrapper
- if we implement our own syscall wraper we can bypass the hook

```
mov   x8, #__NR_openat    ; syscall number
svc   #0                   ; supervisor call -- trap to kernel
```

No PLT. No GOT. No libc. No hooks. The kernel handles it directly.

Capstone 0


handle_getdents64
handle_fstat
handle_write
handle_open
./a.out
hook.so
handle_read
handle_openat
imgflip.com

Userland Rootkit that protects

First part of the capstone is to implement a userland shared object that

- 1. protects files from writing/reading
- 2. hides the existence of files
- 3. protects itself from being deleted
- 4. Allows "special" processes to access protected resources

Rootkit Discussion