Robust Linux Memory Acquisition with Minimal Target Impact

By

Johannes Stuettgen and Michael Cohen

Presented At

The Digital Forensic Research Conference

DFRWS 2014 EU Amsterdam, NL (May 7th - 9th)

LMAP

Robust Linux Memory Acquisition with Minimal Target Impact

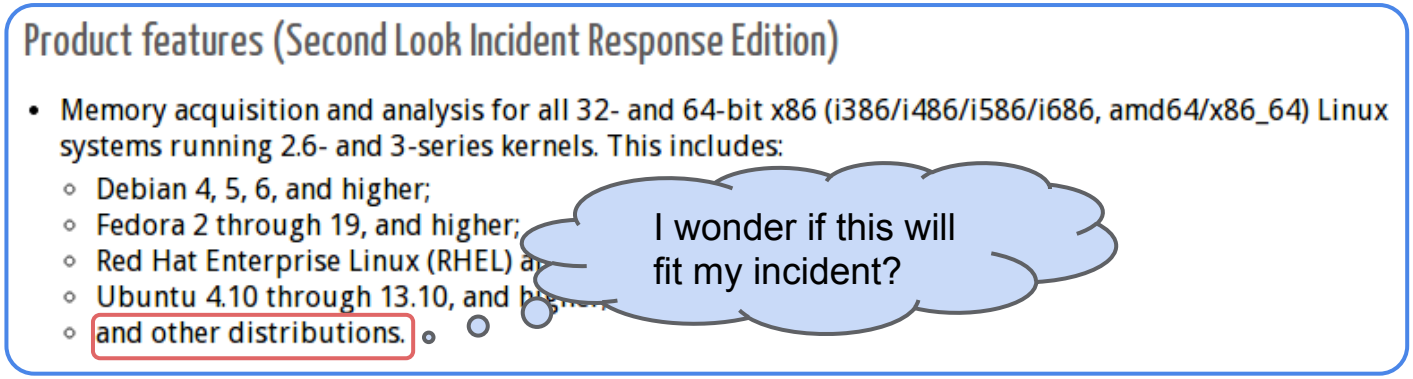
Johannes Stuettgen and Michael Cohen

Outline

- Problem statement
- Anatomy of a linux kernel module
  - How is a LKM loaded and linked
  - Why does it need to be compiled for a specific kernel config and version
- Existing techniques to load incompatible modules
- Technical details of a new, more stable approach
- Demo
- Challenges and Future work

Problem statement

- Current linux memory acquisition tools need to be compiled to the exact kernel version they need to run on.
  - This is difficult for incident response since we don't always know which kernel we will be responding to.
  - Some solutions pre-build a large library of commonly used standard distribution kernels.
    - But this does not solve the problem of customized or hand built kernels.
- The last thing you want to worry about during an incident is whether you have the exact kernel version in your library!



What is a Linux Kernel Module?

- Not an executable!
- Relocatable ELF object. Basically the same as an .o file
- Loading a kernel module is actually more similar to **static linking** of an object file - not dynamic linking.

/bin/ls

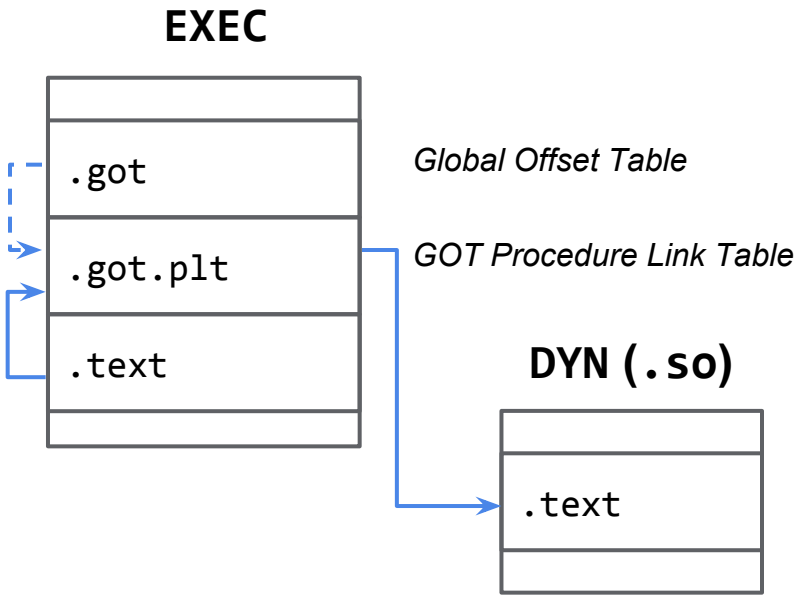
```
Class:                             ELF64
Data:                              2's complement, little endian
Version:                           1 (current)
OS/ABI:                            UNIX - System V
ABI Version:                       0
Type:                              EXEC (Executable file)
Machine:                           Advanced Micro Devices X86-64
Version:                           0x1
Entry point address:               0x4045a4
Start of program headers:          64 (bytes into file)
Start of section headers:          104048 (bytes into file)
```

pmem.ko

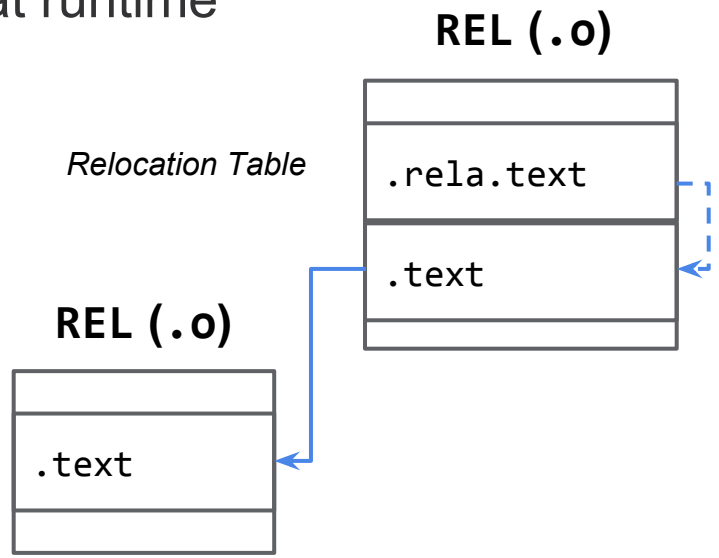
```
Class:                             ELF64
Data:                              2's complement, little endian
Version:                           1 (current)
OS/ABI:                            UNIX - System V
ABI Version:                       0
Type:                              REL (Relocatable file)
Machine:                           Advanced Micro Devices X86-64
Version:                           0x1
Entry point address:               0x0
Start of program headers:          0 (bytes into file)
Start of section headers:          11376 (bytes into file)
```

Dynamic vs. Static Linking

- Static linking usually performed when building an executable
- Dynamic linking performed by `ld.so` at runtime



- .got holds offsets of symbols resolved by linker at runtime.
- Jumps into the .got.plt use .got to jump to referenced sybol



- Static linking resolves addresses of all symbols referred to from other sections or objects.
- Linker uses `.rela` to change `.text` references to the final `.text` offsets in another object.

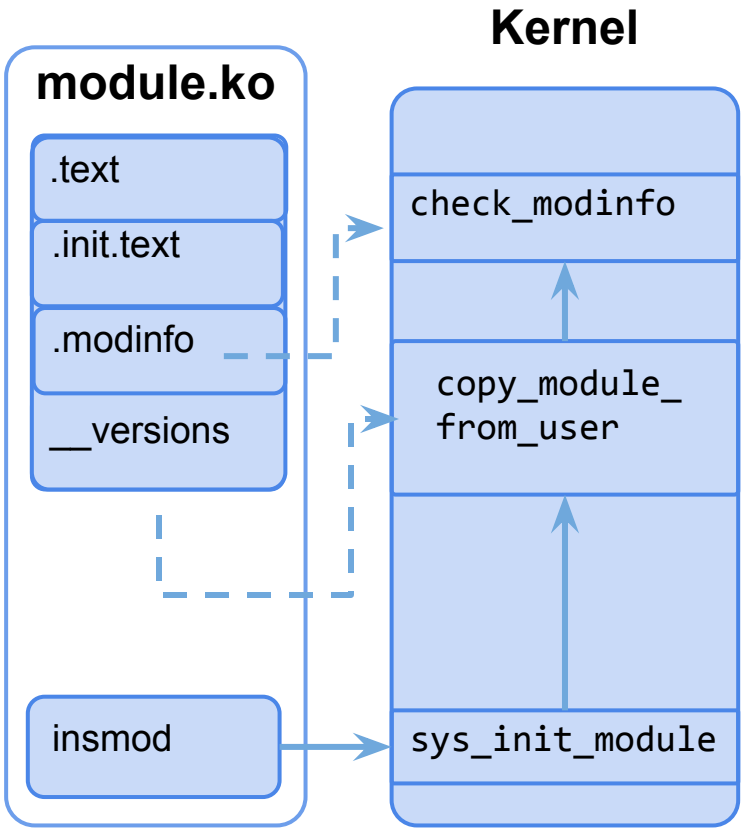
Static Linking

- Object contains list of relocation entries
- Linker iterates over relocation entries
  - Resolves relocated symbol
  - Patches object directly with address of symbol
- Relocations also used for data references

```
Relocation section '.rela.text' at offset 0x2c48 contains 134 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000005  004100000002 R_X86_64_PC32     0000000000000000 _cond_resched - 4
00000000005e  004800000002 R_X86_64_PC32     0000000000000000 __register_chrdev - 4
00000000008a  004d00000002 R_X86_64_PC32     0000000000000000 __unregister_chrdev - 4
00000000015b  003800000002 R_X86_64_PC32     0000000000000000 pte_mmap - 4
000000000166  003800000002 R_X86_64_PC32     0000000000000000 pte_mmap - 4
000000000188  00050000000b R_X86_64_32S      0000000000000000 .rodata + 18
```

Loading an LKM in Linux

1. `insmod` called from user-space
   - prepares module image
   - performs `init_module` system call
2. Kernel allocates memory and copies the module in
3. Kernel verifies module is compiled with identical config and version:
   - Checks module compatible kernel version in `.modinfo`
   - Verifies each imported symbol's version in `__versions` if `CONFIG_MODVERSIONS` is enabled



Loading an LKM in Linux continued

4. Kernel resolves relocations by directly patching references to symbols
5. Kernel calls init_module function in the module

Kernel

```
int apply_relocate_add(...) {
  for (i = 0; ... ; i++) {
    ...
    val = sym->st_value + rel[i].r_addend;
    ...
    switch (ELF64_R_TYPE(rel[i].r_info)) {
      ...
      case R_X86_64_64:
        *(u64 *)loc = val;
      ...
    }
  }
}
```

module.ko

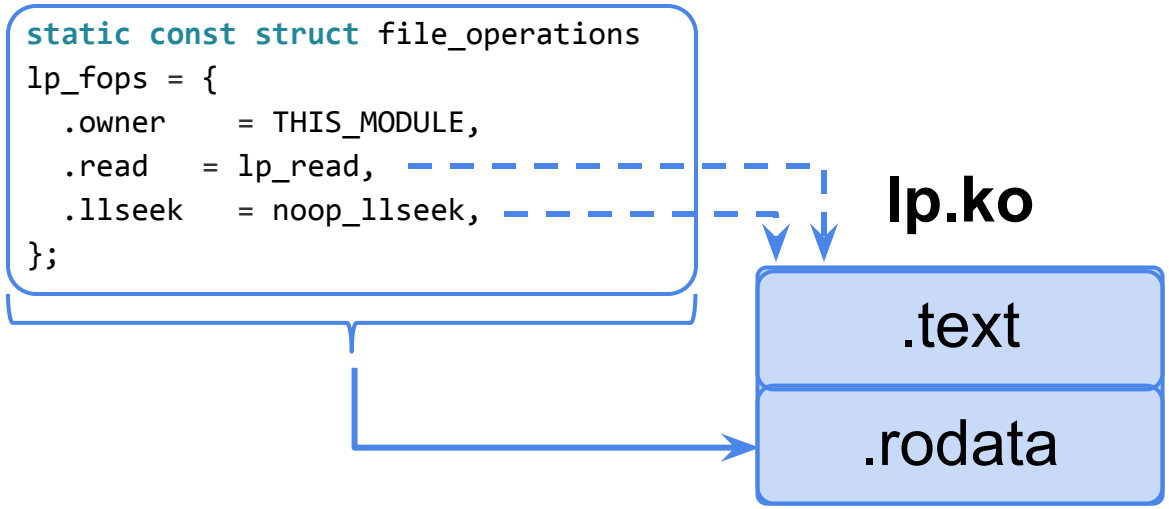
```
Relocation section '.rela.gnu.linkonce.this_module'
  Offset          Type     Sym. Name + Addend
000000000150 R_X86_64_64 init_module + 0
```

```
struct module __this_module
__attribute__((section(".gnu.linkonce.this_module"))) =
{
        .name = KBUILD_MODNAME,
        .init = init_module,
        ...
};
```

```
/* This is where the real work happens */
static int do_init_module(struct module
*mod) {
    ...
    /* Start the module */
    if (mod->init != NULL)
      ret = do_one_initcall(mod->init);
    ...
    /* Now it's a first class citizen! */
    mod->state = MODULE_STATE_LIVE;
    ...
}
```

How do LKMs communicate with user-space?

- Typical LKMs use file operations to communicate with user-space:
  - Create a `file_operations` struct with function pointers to implemented functions registered with this device.
  - Register a device inode under `/dev/`



Modules have a static fops struct in the .rodata segment containing function pointers to `.text`.

How to load LKM into different kernel?

1. The kernel config option "CONFIG_MODULE_FORCE_LOAD" allows modules without valid vermagic to be loaded.
   a. In practice few kernel enable this because it is quite dangerous.
   b. *“Forced module loading [...] is usually a really bad idea.”*
2. It is possible to trick the kernel into loading arbitrary modules by patching the .modinfo and __versions sections
   a. “Steal” valid signatures from other existing modules.
   b. Copy them into arbitrary module to make the version check pass

```
Contents of section .modinfo:
 0000 6c696365 6e73653d 47504c00 7665726d  license=GPL.verm
 0010 61676963 3d332e38 2e302d33 332d6765  agic=3.8.0-33-ge
 0020 6e657269 6320534d 50206d6f 645f756e  neric SMP mod_un
 0030 6c6f6164 206d6f64 76657273 696f6e73  load modversions
 0040 20006465 70656e64 733d0073 72637665   .depends=.srcve
 0050 7273696f 6e3d3631 32323832 38414436  rsion=6122828AD6
 0060 32414537 43413544 36364241 3100      2AE7CA5D66BA1.
```

So we have execution, now what?

- By cheating the version check, we can get execution in the init_module code. However, this is not enough
- We still depend on the struct layout of the kernel which we used to compile the module.

```
/* This is where the real work happens */
static int do_init_module(struct module *mod)
{
    ...
    /* Start the module */
    if (mod->init != NULL)
      ret = do_one_initcall(mod->init);
    ...
    /* Now it's a first class citizen! */
    mod->state = MODULE_STATE_LIVE;
    ...
}
```

```
struct module {
      enum module_state state;
      struct list_head list;
      char name[MODULE_NAME_LEN];
      ...
#ifdef CONFIG_UNUSED_SYMBOLS
      ...
#endif
#ifdef CONFIG_MODULE_SIG
      bool sig_ok;
#endif
      ...
      /* Startup function. */
      int (*init)(void);
      ...
}
```

What actually needs to be done

- We need to be able to insert arbitrary code into the running kernel
  - Pass version check.
  - Get code execution through a valid `struct module->init`
- We need to predict the layout of the kernel's data structures
  - We definitely need `struct module`
  - We also need `struct file_operations` for communication with user-space.
  - Finally, we need other API's structs as needed by the acquisition tool
    - The more complex work we do in the driver, the more APIs we will use which makes this harder (e.g. copy to network, copy to disk etc).
- Ideally we want to use as few APIs as possible in the driver!
  - No disk I/O in the driver
  - No usage of memory mapping and enumeration API’s
  - No memory allocation!

LKM infection to the rescue

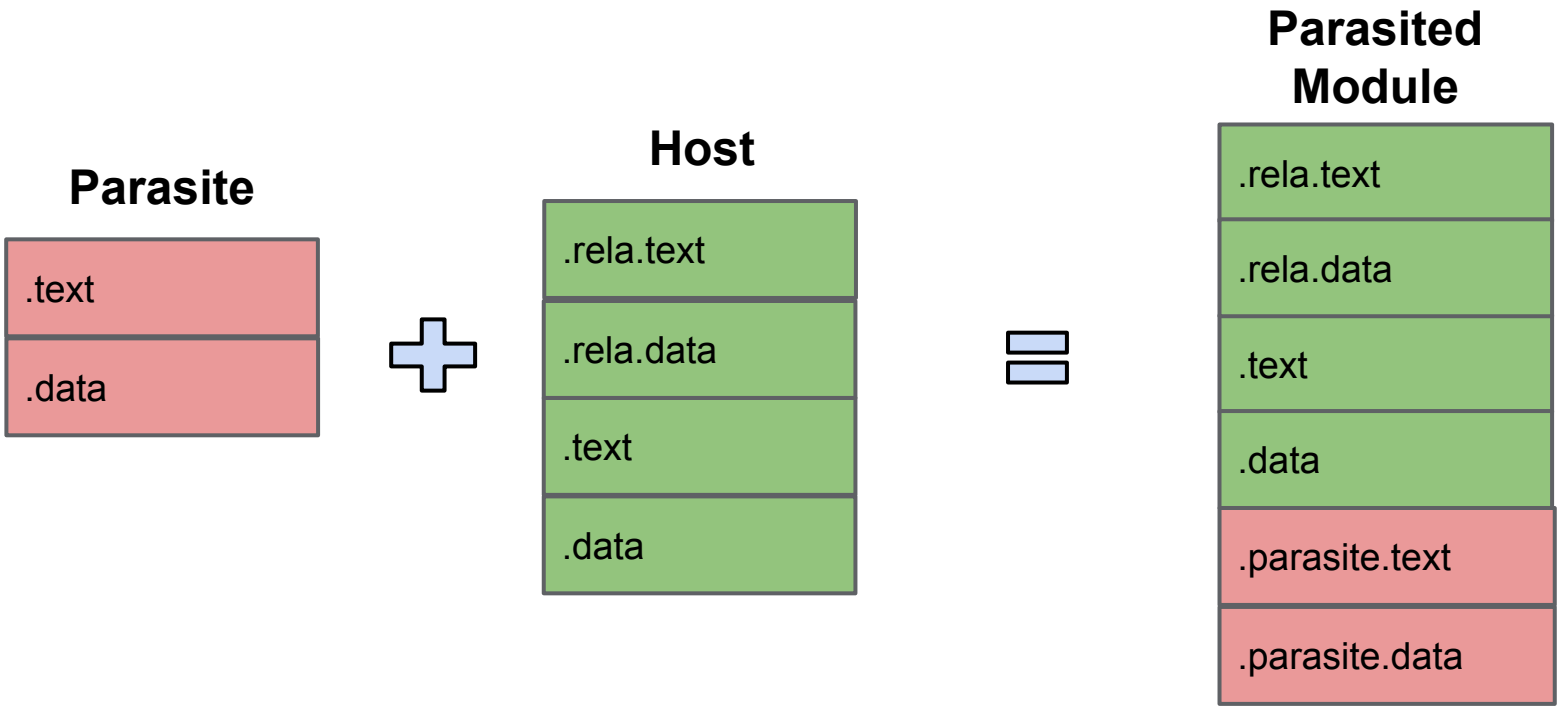
- An LKM is a relocatable ELF and can be linked to other objects
- Easy to add code and data to it
  - ‘`ld -r parasite.ko host.ko`’
- Known and stable rootkit persistence technique
  - Phrack #61 (2003) File `0x0a` by Truff: “*Infecting loadable kernel modules*”
  - Phrack #68 (2012) File 0x0b by Styx: *“Infecting loadable kernel modules kernel versions 2.6.x/3.0.x”*
  - Used by the Adore-ng rootkit (and probably many others)
- Several ways to divert execution flow to injected code
  - Most commonly done by renaming symbols
- We’re more interested in the structs a host module has to offer
  - Contains valid `struct module` for running kernel
  - Has relocations that “*know*” about struct layout

Selection of Host Module

- Scan `/lib/modules/`uname -r`/` for suitable modules.
- What makes a module suitable?
  - Needs to contain the following symbols:
    - `struct file operations`
    - `read` and `llseek` methods
  - Has to import `register_chrdev` to register it's fops struct
    - legacy method for registering character devices
    - More widely supported on older kernels than `cdev_init`
  - needs to have relocations from `read` and `seek` functions into `static struct file_operations`
- In practice there are many suitable modules on most systems:
  - `lp.ko`, `irnet.ko` etc.

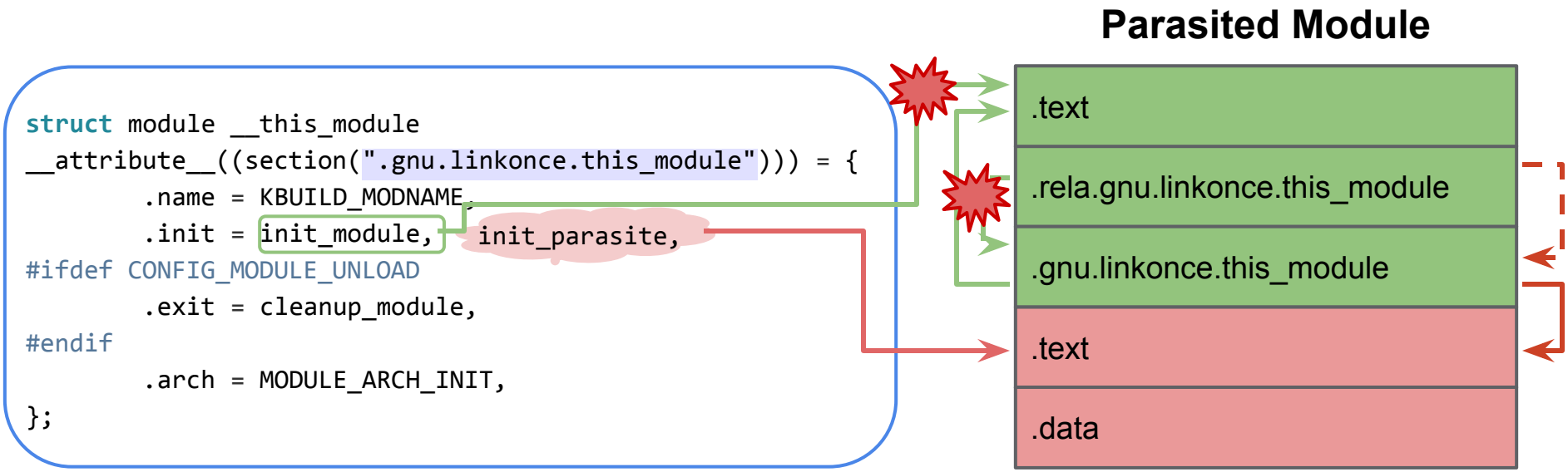
Inject our code into the module

- Copy `.text` and `.data` sections from the parasite into the host



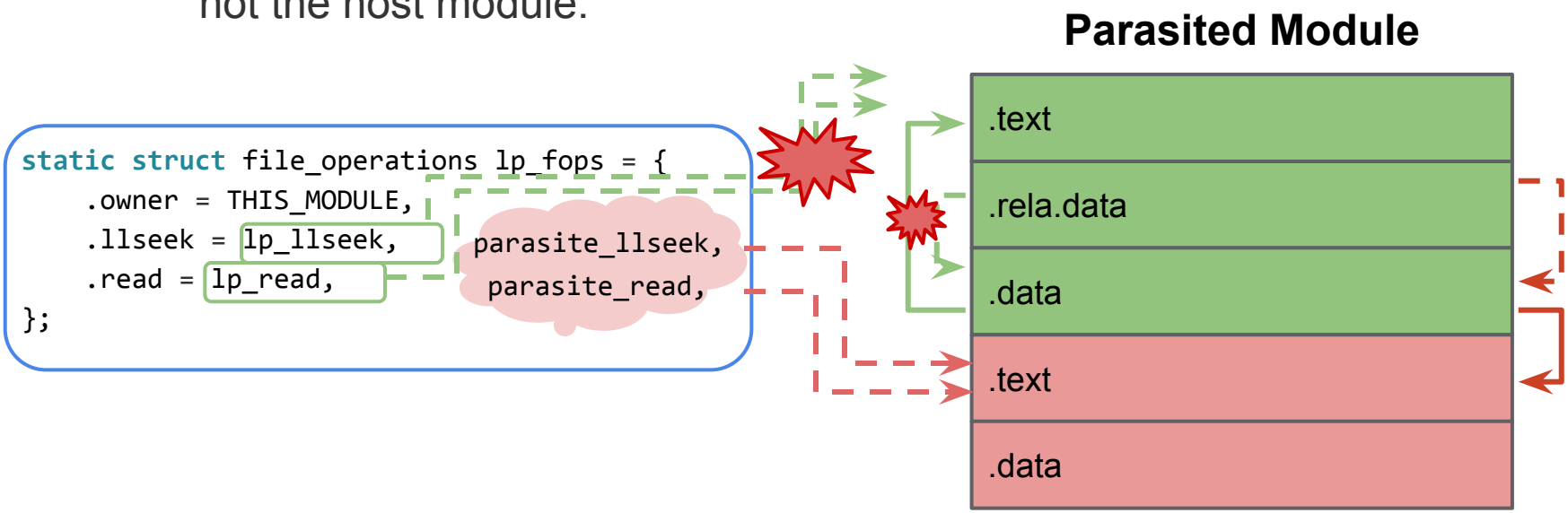
Redirect execution to parasite code

- Fix up relocations to put the location of our code into `__this_module->init`, instead of the location of `module_init`.
- We don't need to know anything about the layout of struct module, the kernel linker will handle it for us



Redirect File Operations

- Fix up the relocations to point at our code for `struct file_operations`.
  - All read and seek operations on the device file call the parasite now, not the host module.

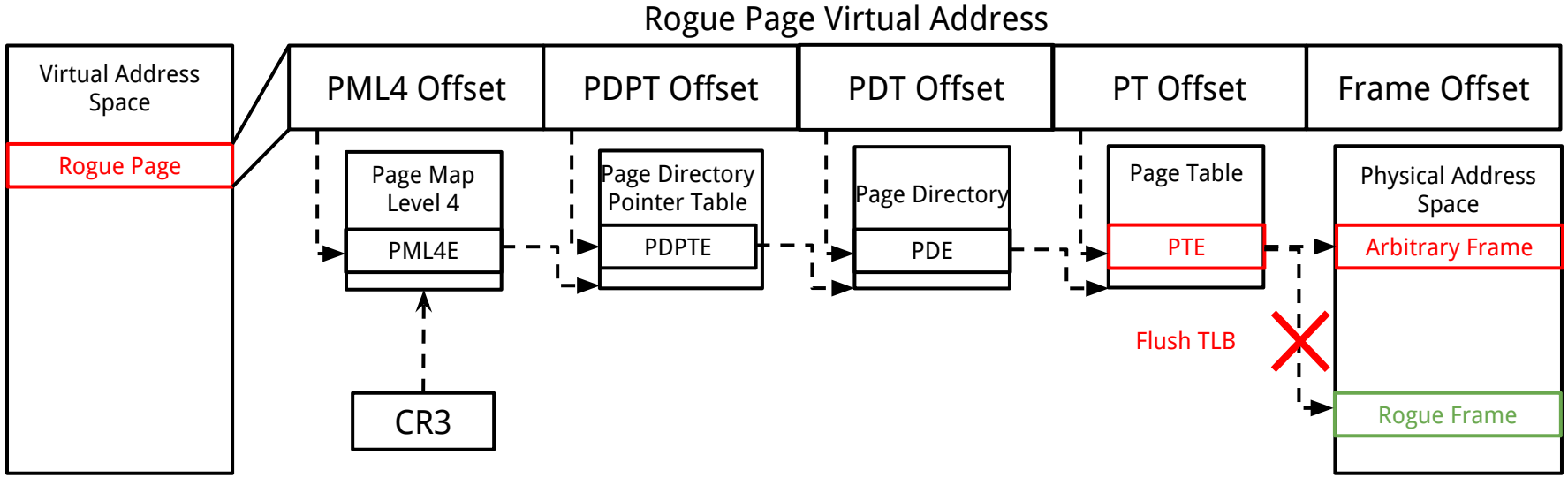


API Independent Acquisition Code.

- Even after getting code execution we need to ensure we are not using the wrong APIs:
  - Internal kernel struct layouts could change between versions.
  - Kernel APIs can change between versions.
- If we declare a dependency on an exported symbol we need to also declare a version magic for it.
  - This means we need to search existing modules for this symbol. (To copy the magic hash from their `__versions` section.)
  - It may not exist on the running system.
- Ideally a memory acquisition module should not depend on any kernel APIs at all!

PTE based acquisition methods.

- Manually parse and manipulate the page tables.
  - No dependencies on mapping API's.
- Remap page in data segment to any physical address as needed.
  - No need for memory allocation.
- Get data to user-space via read calls on device inode.
  - User supplied buffer, no memory allocation in driver.



Demo

- Minimal version of `pmem_pte`
- Packaged with a statically linked loader (`LMAP`)
- Build on an Ubuntu 12.10 system with kernel `3.5.0-41-generic`
- Dynamically finds, injects and patches a suitable host
- In this example a Debian 7.0 with Kernel `3.2.0-4-amd64`

```
root@victim:~# ./lmap -a dump
[+] Scanning modules in /lib/modules/3.2.0-4-amd64 for suitable host
[+] Found suitable host /lib/modules/3.2.0-4-amd64/kernel/drivers/char/lp.ko
[+] Using underscore prefixes and noop_llseek (kernel 3.x) relocation hooking method (2)
[+] Successfully injected parasite into /lib/modules/3.2.0-4-amd64/kernel/drivers/char/lp.ko
[+] Injected pmem module has been loaded
[+] Starting to dump memory
[+] [0000000000010000 - 000000000009efff] [WRITTEN]
[+] [0000000000100000 - 000000007ffeffff] [WRITTEN]
[+] Acquired 524159 pages (2146955264 bytes)
[+] Size of accessible physical address space: 2147418112 bytes (2 segments)
[+] Successfully wrote elf image of memory to dump
```

Challenges

- Some APIs and interfaces change over time
  - Character devices don't use `llseek` in `struct file_operations` on kernels < 2.6.37.
    - Breaks random access on these systems.
  - Kernel symbol names change sometimes
    - `copy_to_user` vs `_copy_to_user`
- Code stability is a big issue
  - Crashes have fatal consequences
- 32-Bit support
  - Different relocations and data types, should be pretty straight forward.

Future Work

- Android Memory Acquisition.
  - Cellphone vendors often don’t release their kernel sources.
  - Adapt existing implementation to ARM style relocations.
- Relocations contain implicit information about struct layout.
  - This is also very interesting for analysis.
  - Can derive config settings from relocation offsets.
  - Build a partial Volatility profile from this.

```
000000000080  004300000001 R_X86_64_64       0000000000000000 __this_module
000000000088  005700000001 R_X86_64_64       0000000000000000 noop_llseek
000000000090  000100000001 R_X86_64_64       0000000000000000 lp_read
000000000098  000100000001 R_X86_64_64       0000000000000000 lp_write
0000000000c0  000100000001 R_X86_64_64       0000000000000000 lp_ioctl
0000000000c8  000100000001 R_X86_64_64       0000000000000000 lp_compat_ioctl
0000000000d8  000100000001 R_X86_64_64       0000000000000000 lp_open
0000000000e8  000100000001 R_X86_64_64       0000000000000000 lp_release
```

The End

Slides: `http://goo.gl/4hbPCJ`

Code: `http://rekall-forensic.com`