Robust Linux Memory Acquisition with Minimal Target Impact

By

Johannes Stuettgen and Michael Cohen

From the proceedings of

The Digital Forensic Research Conference

DFRWS 2014 EU

Amsterdam, NL (May 7th - 9th)

Digital Investigation 11 (2014) S112–S119

Contents lists available at ScienceDirect

Digital Investigation

journal homepage: www.elsevier.com/locate/diin

Robust Linux memory acquisition with minimal target impact

Johannes Stüttgen [a,*], Michael Cohen [b]

[a] Department of Computer Science, Friedrich-Alexander University of Erlangen-Nuremberg, Martensstraße 3, 91058 Erlangen, Germany
[b] Google Inc., Brandschenkestrasse 110, Zurich, Switzerland

Keywords:
Memory forensics
Linux memory acquisition
Memory acquisition
Incident response
Live forensics

ABSTRACT

Software based Memory acquisition on modern systems typically requires the insertion of a kernel module into the running kernel. On Linux, kernel modules must be compiled against the exact version of kernel headers and the exact kernel configuration used to build the currently executing kernel. This makes Linux memory acquisition significantly more complex in practice, than on other platforms due to the number of variations of kernel versions and configurations, especially when responding to incidents. The Linux kernel maintains a checksum of kernel version and will generally refuse to load a module which was compiled against a different kernel version. Although there are some techniques to override this check, there is an inherent danger leading to an unstable kernel and possible kernel crashes. This paper presents a novel technique to safely load a pre-compiled kernel module for acquisition on a wide range of Linux kernel versions and configuration. Our technique injects a minimal acquisition module (parasite) into another valid kernel module (host) already found on the target system. The resulting combined module is then relinked in such a way as to grant code execution and control over vital data structures to the acquisition code, whilst the host module remains dormant during runtime.

Introduction

Memory analysis has rapidly become a very powerful tool in the arsenal of incident responders and forensic examiners. Frameworks such as Volatility (Walters, 2007) or Second Look (Raytheon Pikewerks, 2013) allow an in-depth analysis of operating system data structures and can be used to gain a thorough understanding on a live systems state like running processes, network connections and mapped files.

For this to work it is necessary to acquire a memory image. While multiple methods to do this using physical access to the hardware exist (Carrier and Grand, 2004; Boileau, 2006), physical access is often not available in an incident response scenario. Thus, a software based approach is sometimes the only viable option. Because current operating systems operate in protected mode for security and safety reasons, acquisition of the entire physical address space can only be achieved in system mode. For Linux this typically requires the injection of a Linux kernel module into the running kernel. Since the Linux kernel checks modules for having the correct version and checksums before loading, the kernel will typically refuse to load a kernel module pre-compiled on a different kernel version or configuration to the one being acquired. This check is necessary since the struct layout of internal kernel data structures varies between versions and configurations, and loading an incompatible kernel version will result in kernel instability and a potential crash.

For incident response this requirement makes memory acquisition problematic, since often responders do not

[*] Corresponding author.
E-mail addresses: johannes.stuettgen@cs.fau.de (J. Stüttgen), scudette@google.com (M. Cohen).

http://dx.doi.org/10.1016/j.diin.2014.03.014

know in advance which kernel version they will need to acquire. It is not always possible to compile the kernel module on the acquired system, which may not even have compilers or kernel headers installed.

Some Linux memory acquisition solutions aim to solve this problem by maintaining a vast library of kernel modules for every possible distribution and kernel version (Raytheon Pikewerks, 2013). While this works well as long as the specific kernel is available in the library, it is hard to maintain and can not cover cases where the kernel has been custom compiled or just is not common enough to award a place in the library. This is especially the case on mobile phones. Often phone vendors might publish the kernel version they used, but the configuration and details on all vendor specific patches are often not known, severely impeding memory acquisition (Sylve et al., 2012).

Rootkit authors also have encountered the same problem when trying to infect kernels where the build environment is not available. Recent work for Android shows that while it is trivial to bypass module version checking, it is still a hard problem to identify struct layout in unknown binary kernels (You, 2012). In the Android case this problem is solved by restricting dependencies to very few kernel symbols and reverse engineering their data structures on the fly using heuristics (You, 2012).

A solution for data structure layout detection could be live disassembly of functions which are known to be stable and use certain members in these structs. Recent work showed that it's possible to dynamically determine the offsets of particular members in certain structs used in memory management, file I/O and the socket API (Case et al., 2010).

Kernel integrity monitoring systems also have similar problems, as they have to monitor dynamic data and need to infer its type and structure to analyze it. Since this data layout changes with kernel version, these systems need to infer its data layout from external sources. The KOP (Carbone et al., 2009) and MAS (Weidong et al., 2012) frameworks, are exemplary systems designed to monitor integrity of dynamic kernel data structures. Their approach involves statically analyzing the kernel source code and debug symbols to infer type information for dynamic data. However, they rely on the kernel source-code and debug symbols for the exact running kernel being available in advance, which is exactly the dependency we can not guarantee in the incident response scenario.

Contributions. We have developed a method to inject a parasite kernel module into an already existing host kernel module as found on the running system. Most modern kernels have a large number of legitimate kernel modules, compiled specifically for the running kernel, already present on the system. Our approach locates a suitable existing kernel module (Host Module), injects a new kernel module into it (Parasite module) and loads the combined module into ring 0.

The resulting modified kernel module is fully compatible with the running kernel. All data structures accessed by the kernel are taken from the Host module, and were in fact compiled with compatible kernel headers and config options. However, control flow is diverted from the Host module to the Parasite module, by modifying static linking information. This allows the parasite module's code to use the hosts' structs for communication with kernel APIs.

Anatomy of a Linux kernel module

Linux kernel modules are relocatable ELF object files and not an executable. The obvious difference is that executable ELF files are processed by a loader, while relocatable objects are intended for a linker.

The loader relies on the ELF Program Headers to identify the file layout and decide which parts to map into memory with which permissions. The linker instead relies on ELF section headers for this, with special sections containing symbol string and relocation tables, to identify and resolve inter-section symbol references (Fig. 1).

Dependencies on other objects in an ELF executable are resolved by dynamic linking. In this process, external symbols are referenced through the Global Offset Table (.got) and Procedure Linkage Table (.got.plt), and resolved by the dynamic linker at runtime (Fig. 2).

In contrast to this, relocatable ELF objects are statically linked using relocations. Each section with references to symbols in other sections or objects has a corresponding relocation table. Entries in these tables contain information on the specific symbol referenced, and how to patch a specific code or data reference with the final address of the symbol after it has been relocated.

One or more of these relocatable objects can be linked together by placing them into their final position in the final executable or address space, after which the linker applies all relocations to patch the now final references directly into the code.

In the context of the Linux kernel this means that loading a kernel module is actually the same thing as linking an executable, but with the executable being the running kernel image.

How is a LKM loaded and linked

The actual loading process of a kernel module can be characterized by four steps, which begin in user space (Fig. 3):

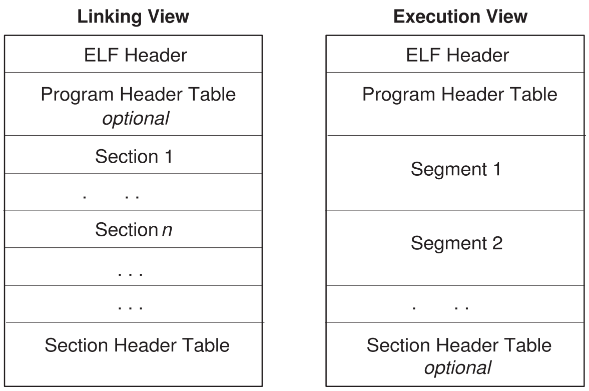


Fig. 1. ELF file layout (TIS Committee, 1995).

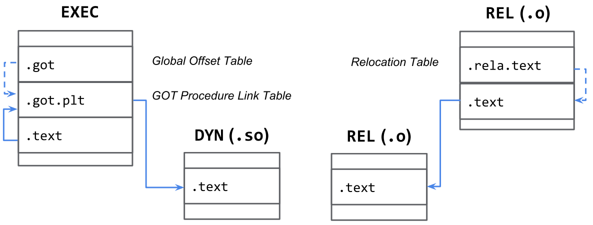


Fig. 2. Static vs. dynamic linking.

1. A user-mode process loads the kernel module image into memory and issues an `init_module` system call. This causes the kernel to dynamically allocate memory for the module and copy it into kernel space.
2. The kernel verifies the ELF headers and then starts to analyze two special sections in the module, `.modinfo` and `__versions`. These sections contain information on the exact version of kernel headers the module was compiled with. The kernel will refuse to load any modules that contain incompatible version magic.
3. After the version check, the kernel invokes its internal linker to resolve all relocations in the module. This will replace any inter-section or external symbol references in the module with the actual addresses of these symbols in the running kernel, basically assimilating it into the kernel image.
4. Finally, the kernel will link the *struct module* provided by the module into the module list and call the function pointer stored in `module.init`, which passes execution to the modules `init_module` function.

Why kernel modules need to be compiled for a specific kernel version

Linux kernel modules are object files and are linked directly into the running kernel. There is no protection of kernel memory from their actions, they run at the same privilege level and bugs can lead to kernel data corruption and thus to a kernel panic.

Furthermore, since it is directly linked with the module object file, the kernel actually uses some of the modules data structures. Each module contains a special section called `.gnu.linkonce.this_module`, which holds a static *struct module* generated in the compilation process. This struct is defined in the kernel headers, and is used by the kernel for bookkeeping and managing of the module. It is linked into the module list and the kernel will regularly access its members. For example the kernel directly dereferences the `module.init` member to call the modules initialization function (Fig. 5).

Forcing the module to be compiled with the exact same kernel version, configuration and compiler settings ensures that all APIs are compatible and structs have the exact same layout in both the module and the kernel. If the number of members, their order, the compilers padding settings or a conditional member are only present on certain configurations or differ from kernel to module, certain members (like for example the init pointer) will be at a different offset than the kernel expects. The call to `mod->init` might result in a call to something entirely different, uninitialized data or even unmapped memory. This can easily result in a kernel crash, forcing a reboot or leading to possible data loss or corruption (Fig. 4).

Bypassing module version checking

There are multiple ways to get around the version check and load a module even if it was compiled for a different kernel version. However, because of the reasons mentioned before this should only be a last resort as it can result in undefined behavior, data corruption or worse.

There is a kernel config option `“CONFIG_MODULE_FORCE_LOAD”`, which allows modules without valid vermagic to be loaded by using the `-force` option of modprobe. In many cases if the module was compiled on a very closely related kernel (e.g. only the last digit is

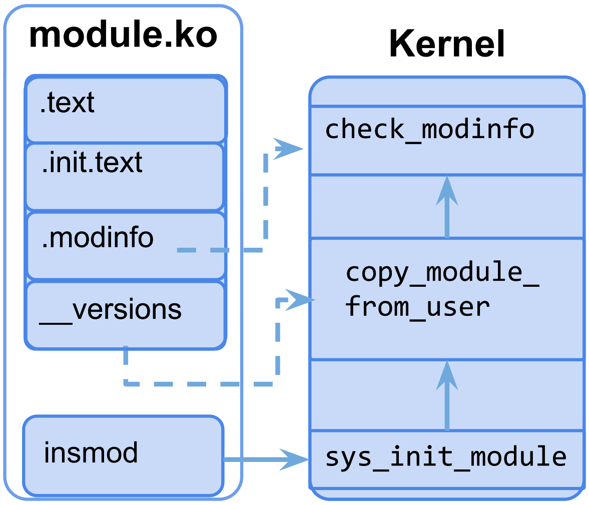
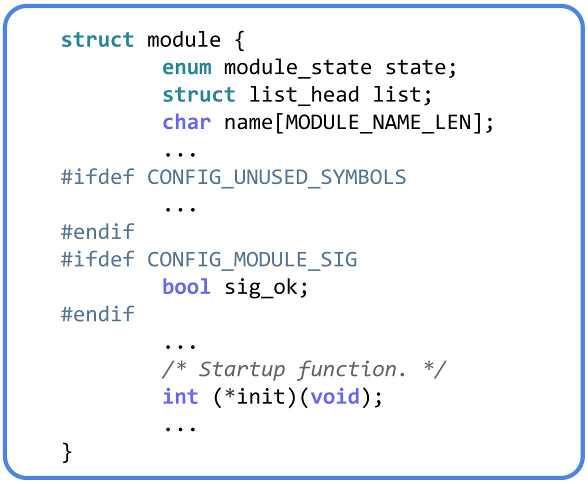


Fig. 3. Loading of a kernel module.



```
    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);
            ...
    }
```

Fig. 4. *struct module* in `kernel/module.h` (The linux kernel Archives, 2013).

different) for the same distribution (which hopefully uses a very similar configuration) this will work. For larger differences this technique could cause a kernel crash and is usually not recommended.

Because it is hard to verify if the versions are compatible without comparing the kernel headers and configuration, this option essentially allows for a gamble with the possibility of a very bad outcome. Documentation clearly states that "*Forced module loading sets the 'F' (forced) taint flag and is usually a really bad idea.*" (The linux kernel Archives, 2013, `init/Kconfig`), which is the reason few production kernels are compiled with this configuration option enabled.

Even without the forced loading option enabled, the kernel can still be tricked into accepting an incompatible module by modification of the `.modinfo` and `__versions` sections. The version magic is not cryptographically signed, so it can simply be extracted from a valid module on the target system and replace the incompatible magic previously stored another module. Because the module now contains valid magic strings for kernel version and all its imported symbols the version check will pass and the kernel will allow the module to be loaded.

Nevertheless, the inherent danger with this is the same as with forced loading. It can result in undefined behavior, kernel crash and data loss.

Finally, the kexec system call offers another way to insert code into system mode. "*[K]exec is a system call that enables you to load and boot into another kernel from the currently running kernel*" (The Linux man-pages, 2012). This can be used to load a custom acquisition kernel, replacing the old one, similar to the approach taken by the Body Snatcher tool (Schatz, 2007). However, this will render the old kernel unusable and there is no way to recover from this into the state the system was in before. Additionally, this system call only exists on kernels compiled with `CONFIG_KEXEC` enabled, so there is no guarantee that it will be available.

Reliable loading of generic acquisition modules

A technique for loading a generic memory acquisition kernel module simplifies the acquisition process for incident responders of Linux systems. Investigators can concentrate on the incident and stop worrying about the exact kernel version of the target system, and prebuilding compatible kernel modules.

Requirements for a stable approach

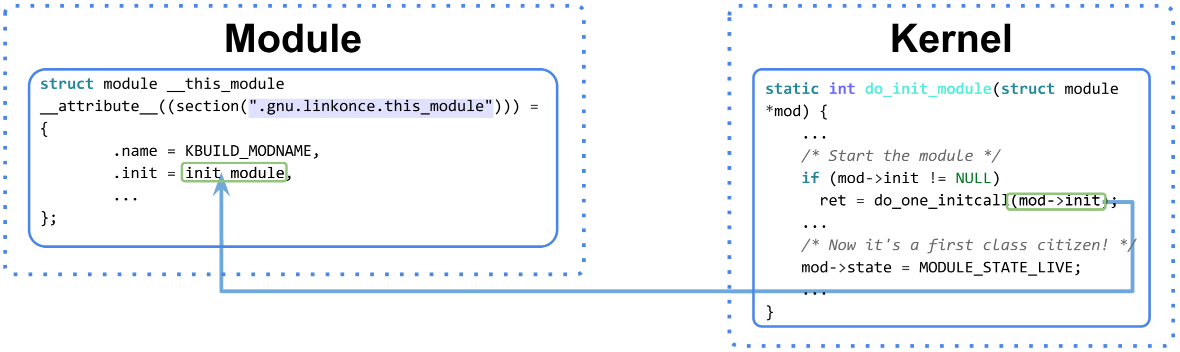
Multiple problems have to be solved to actually do this in a reliable manner without affecting system stability. The first is the matter of getting system mode code execution. We need the ability to insert arbitrary code into the running kernel and pass control to it. This involves bypassing the version check and handing the kernel a valid *struct module* with an `module->init` pointer under our control.

For this to work it is also necessary to predict the layout of the kernels data structures. Especially *struct module* is needed to get code execution in the first place, but usage of many kernel APIs also requires creation of specific structs with the correct layout. For example the creation of a device inode to communicate with user mode requires a kernel module to have a valid `struct file_operations` with correctly positioned pointers to the relevant driver functions (such as `read`, `write`, `llseek`).

The more APIs a kernel module wants to employ, the more data structures have to be used which increases the necessary knowledge on the running kernels struct layout. This implies that the problem becomes much easier to solve if the memory acquisition module uses as few APIs as possible. Some Linux memory acquisition solutions have a rich feature set, such as writing to disk from kernel mode or dumping memory over the network (Sylve, 2012). However, this requires knowledge of vfs and socket struct layout. Additionally, some existing tools parse the `iomem_resource` tree to enumerate physical memory mappings (so as to avoid acquiring DMA regions (Stüttgen and Cohen, 2013)). Kernel APIs mapping the virtual address space or even allocating memory can be difficult to use without detailed knowledge of the running kernels data structures and APIs. Ideally, an acquisition module for this use case should use as few kernel APIs as possible.

Parasitizing a compatible module

The first step in parasitizing a compatible module, is to locate a valid kernel module for the running kernel suitable for parasitizing. On most distributions the directory `/lib/modules/` contains a large number of kernel modules for different devices, which have all been compiled with the



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

```
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;
    ...
}
```

Fig. 5. Initialization of a kernel module.

correct headers and configuration and thus are compatible for linking into the running kernel. Code injection into one of these modules not only allows us to pass the kernel version checks but also ensures that the *struct module* linked into the kernel is compatible.

Parasitizing an existing kernel module is not a novel technique. The technique has been employed by malware authors previously as a stealthy persistence technique (Truff, 2003). Because a kernel module is a relocatable object, it is easy to add new code and data to it using standard tools. It can be essentially relinked with another module to combine both into a single object file. This can be done using the linker `ld` or by copying individual sections using `objcopy`. The Adore-ng rootkit (Stealth, 2004) for example uses this technique to hide its kernel module inside a legitimate one on infected systems and gain code execution when the host is loaded on startup. The method is documented to work on a wide variety of kernels, from the 2.4 series (Truff, 2003) to more current 2.6 and 3.0 kernels (Styx, 2012).

To divert control flow in the infected module, malware usually rewrites the symbol names of initialization functions. By renaming `init_module` to something else and changing the name of the injected initialization routines to `init_module`, the kernel linker will insert the address of the injected routine into the struct `module->init` member on relocation. When the kernel initializes the loaded module it will thus call the malware's code, not the hosts.

While this technique provides a stable method to solve the first problem of getting code execution in a stable manner, it does not address the problem of learning the struct layout of the running kernel. For our use case, we are interested in other structs a host module has to offer. If we can find a kernel module on the target system that contains all necessary structs which the parasite kernel module needs in order to use the kernel APIs, we can parasitize this module and make use of these structs ourselves.

The ELF relocation tables in the host module can then be exploited to patch these structs on module load to suit our needs, without having to know anything about their layout.

Code injection into kernel modules

Previous work used the linker `ld` to link code into the host module (Truff, 2003; Styx, 2012). However, this complicates the build process because it either needs the linker available on the target system, or it is necessary to first copy a suitable module from the target to a system with a suitable build environment, infect it there and then copy the result back. This is both undesirable when responding to an incident, as it changes the target's state and increases forensic impact.

Therefore it is prudent to implement a custom linker that can perform this process on the fly in memory when executed on the target. The linker has to be able to insert entries into section header, symbol and string tables and add sections to the binary.

Redirection of control flow

Once we are able to inject code into a kernel module, we need to divert the control flow away from the host to the parasite. This can be performed by using a technique we call "*Relocation Hooking*". This is commonly used to manipulate entries in the Procedure Linkage Table to hook calls to dynamic libraries in ELF executables (Shoumikhin, 2010). The general idea is that the linker will use information in the relocation tables to patch the programs control flow, thus manipulation of these tables can exploit the linker to patch a program for us.

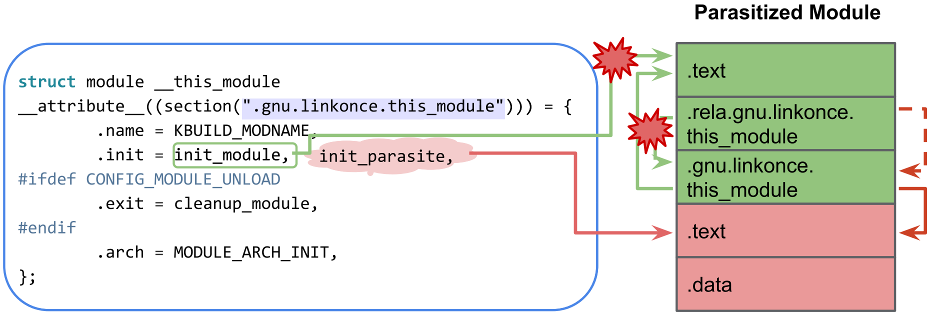
Relocation Tables are an array of relocation entries, each describing the use of a symbol in a specific location of the program. They provide information on how this code needs to be patched to reference the actual address of this symbol, as soon as it has been loaded and its address is known. Because references and addressing are highly architecture dependent, a large number of different types of relocations exist. On x86-64, a relocation table is an array of `struct ELF64_Rela`, storing the offset in the code where the relocation will be performed, information on the type of relocation, the index of the referenced symbol and an addend. Depending on the type of relocation, the addend has to be added to the symbol offset, for example when patching an RIP relative reference in position independent code. There are 37 different types of relocation on x86-64 (Matz et al., 2012), of which only 5 are actually used in kernel modules (The linux kernel Archives, 2013, `arch/x86/kernel/module.c`).

Hooking Module Initialization. Each kernel module contains a *struct module* called `__this_module`, which is automatically generated from the module source code at compile time by expansion of some macros. The resulting definition is available in the generated .mod.c file as seen in Fig. 6, and is linked into the module using the relocation table for its section (`.gnu.linkonce.this_module`). This struct is then used by the kernel to call the initialization code pointed to by `__this_module->init`. The relocation table for this section has an entry that instructs the kernel to patch the address of the `init_module` function into this member of the struct. By modifying the symbol index in that relocation entry we can make the linker patch any symbol we want into the struct when the module is loaded. Thus it is sufficient to find this relocation entry and change its symbol index to the one of the parasites initialization function to get code execution.

Note that we don't need to know anything about the layout of *struct module* at all to do this, all information needed to patch this struct is available in the relocation entry and the patch itself is performed by the linker.

Communication with User Mode. Even after we have code execution, we still lack a method of communicating with user space. A memory acquisition driver needs to receive instructions from user space on which physical pages to acquire and needs to pass these pages back to user space.

One of the simplest and most commonly used methods for system- to user-mode communication in Linux is the character device. A kernel module can create a struct `file_operations`, which contains function pointers for operations like `read`, `write`, `llseek` and such. The module then registers a major number with the kernel, which will link the struct to any inode referencing that major number. The system call `mknod` can be used from user space to



```
struct module __this_module
__attribute__((section(".gnu.linkonce.this_module"))) = {
        .name = KBUILD_MODNAME,
        .init = init_module, init_parasite,
#ifdef CONFIG_MODULE_UNLOAD
        .exit = cleanup_module,
#endif
        .arch = MODULE_ARCH_INIT,
};
```

Fig. 6. Relocation hook of `module->init`.

create such an inode. Any file operations on this inode will be dispatched to the functions referenced in the corresponding `struct file_operations`.

If the host module implements a character device, it must already have a compatible version of this struct in its `.data` or `.rodata` section (Usually kernel modules initialize their `struct file_operations` statically at compile time). To populate the function pointers in this struct there have to be relocation entries for this section, because the functions are placed in another section whose address is not known until loading time. When the kernel loads the module, the linker relocates the sections and then places the addresses of all relevant functions into the `struct file_operations`, by parsing the corresponding relocation table.

We can exploit this process by modifying the relocation table of the host to point to a symbol of our choice instead of the original `read` and `llseek` functions exported by the Host Module (Fig. 7).

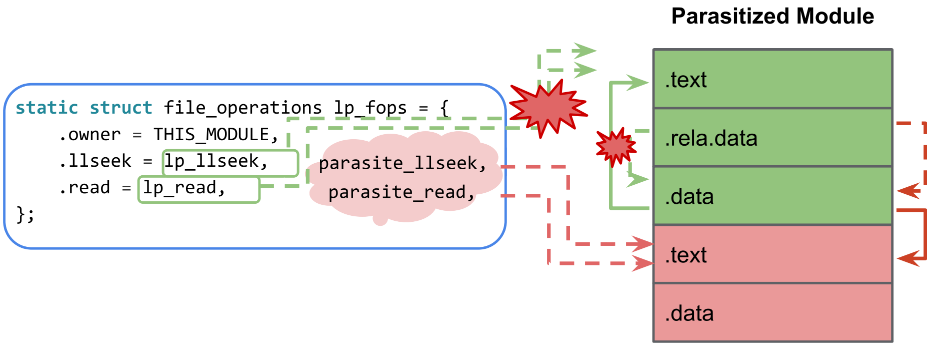
When the parasitized module is loaded, the kernel linker will patch the struct with function pointers to the parasites' `read` and `llseek` functions instead. The parasite can then call the `register_chrdev` API in the kernel with a pointer to this struct, which is guaranteed to be compatible with the running kernel. Thanks to the relocation entries we don't need to know the layout of `struct file_operations` to do this. Our pointers will be placed at the correct offsets by the linker and any `read` or `llseek` calls to a device inode with our major number will be dispatched to the parasites' `read` or `llseek` functions.

Selection of suitable host

Due to the need for certain symbols and structs, this approach won't work with arbitrary kernel modules. However, most distributions ship with a large number of modules to handle many different hardware devices, which are found in `/lib/modules/‘uname -r’`. We can scan this directory and select a host module that satisfies the following criteria:

- It contains a symbol with an `_fops` suffix in the `.data` or `.rodata` section, which indicates it has a `struct file_operations` available.
- It contains symbols with `_read` and `_llseek` suffixes, with relocation entries into the `struct file_operations`. This is necessary for us to successfully patch the `struct file_operations`.
- It imports the symbols `register_chrdev` and `copy_to_user`, which the parasite needs to register the file operations struct with a major number and copy data to user buffers when called for `read`.

If we find such a module on the target we can load it into memory, inject the acquisition module, hook the relocations and then pass it to the `init_module` system call for linking into the kernel.



```
static struct file_operations lp_fops = {
    .owner = THIS_MODULE,
    .llseek = lp_llseek,   parasite_llseek,
    .read = lp_read,       parasite_read,
};
```

Fig. 7. Relocation hook of `struct file_operations`.

Implementation of acquisition module

As mentioned in Section Requirements for a Stable Approach, it is important that the memory acquisition module imports as few kernel symbols as possible. While it is possible to employ the same technique for other data structures as used on struct module and struct file_operations, this increases the requirements on the host module.

For each additional API we want to use we add a dependency that must be satisfied by some module on the target. This decreases the number of suitable modules, reducing the chance of finding a suitable host.

We have developed a minimal physical memory acquisition module, which only relies on the `register_chrdev` and `copy_to_user` symbols. The module is based on the `pte_mmap` library for mapping memory without kernel support (Stüttgen and Cohen, 2013). This is accomplished by directly editing the page tables and manually remapping parts of the modules data segment to the desired physical page. A few other tweaks were necessary to remove symbol dependencies.

Commonly, memory acquisition modules perform I/O range checks in kernel mode by parsing the `iomem_resource` tree (Sylve, 2012; Cohen, 2011). However, this requires knowledge of the `struct resource` layout. We removed this functionality from the kernel module, and leave the detection of physical memory layout to the user space imaging tool. Typically this can be determined for example by parsing `/proc/iomem` or using pci introspection (Stüttgen and Cohen, 2013).

The original `pte_mmap` also used the `preempt_disable` and `preempt_enable` symbols to ensure the modules thread can't be interrupted and resumed on another CPU. Because the TLB of another CPU might still contain the old mapping for the remapped page, this could result in a corrupted image. Use of these symbols implies we would have to find a valid version magic on the target, which we don't want to rely on. We have replaced them by simply using the `cli`/`sti` instructions to disable interrupts for the brief period of remapping and copying a page.

We also removed debug logging from the module, as not every suitable host module might import `printk`.

Furthermore, we removed all dynamic memory allocation from the pmem module, and placed all data structures into the data segment. This even allows us to get rid of the `kmalloc`, `vmalloc`, `kfree` and `vfree` symbols, as each module might use a different memory allocation API and we don't want to limit our selection in target modules this way.

Another important detail we discovered when trying to make a module as version independent from the running kernel as possible is config options affecting APIs. For example the `copy_to_user` API is an inline function calling `_copy_to_user` after performing some debug bookkeeping on kernels with config option `CONFIG_DEBUG_ATOMIC_SLEEP` enabled (on kernels newer than 3.0, older kernels have `CONFIG_DEBUG_SPINLOCK_SLEEP`).

Compiling in an environment where this option is enabled will result in a symbol dependency that kernels compiled without it can not satisfy, thus limiting the scope where the module can be successfully loaded. Also this causes problems when scanning for suitable hosts, as they import `_copy_to_user` when this option is enabled and `copy_to_user` when compiled without it. We have solved this problem by explicitly calling `_copy_to_user` in our module, and modifying the symbol table to use the correct one depending on what the host uses. Since `copy_to_user` essentially calls `_copy_to_user`, this is doesn't affect the codes correctness or stability.

Finally, the build environment needs to be slightly tweaked, because some configuration options trigger dependencies on symbols that might not be available on the target system. For example, if the `CONFIG_FUNCTION_TRACER` option is enabled, all functions will call the symbol `__fentry__` at the beginning to enable `ftrace` functionality in the kernel. Any module compiled with this will depend on the `__fentry__` symbol which is not available on kernels without `ftrace`.

Evaluation

We have evaluated on multiple Linux distributions and kernel versions to provide data on how big the difference in kernel version can actually be while still being able to obtain a physical memory image. We compiled our parasite module on an Ubuntu system with kernel 3.8.0-34. We do not believe this technique will work on 2.4 kernels due to massive changes in LKM loading and relocation architecture, so we did not test these.

We have tested our module on six different kernels and distributions as shown in Table 1. The number of available modules was always quite large. All tested systems had a number of suitable modules available, with newer kernels providing 14–15 different suitable host modules.

Our technique was successful in acquiring memory from all tested systems without crashes or any other major problems.

Conclusion and future work

We have developed a physical memory acquisition kernel module and an LKM infection engine that, once built in our environment, can be loaded on any Linux kernel between `2.6.38` and `3.10`, regardless of configuration or compiler options. Testing shows our approach has very little impact on system stability and provides reliable access to physical memory. This simplifies memory forensic procedures significantly and allows for physical memory acquisition even on systems where kernel headers are not available. It also minimizes the impact on the target system,

Table 1
Host modules by kernel version.

Kernel version	Modules available	Modules suitable
2.6.27 (fedora 10)	1746	4
2.6.38 (fedora 15)	2280	14
3.1 (fedora 16)	2384	14
2.6.24 (ubuntu 8.04)	1939	6
3.8 (ubuntu 12.10)	3708	14
3.11 (ubuntu 13.10)	3957	15

as there is no need to install a build environment and compile software on the system that is to be analyzed.

However, one problem remains. For sophisticated analysis of the acquired memory dump we need to gather information on symbols and data structures. The Volatility project for example refers to this as a profile. This profile is usually built by compiling a module with DWARF information for the exact kernel version on the target, which is then parsed to extract struct layout and symbol information (Hale, 2013).

When the kernel version and configuration is not known or available, this is not possible. Further work can utilize the information gathered from existing Linux kernel modules' relocation information to build a partial profile for the target system. By extracting and parsing this information we can get an understanding of the layout at least parts of certain structs. With this knowledge we can build a partial profile without having access to kernel headers and configuration files.

We also want to port the relocation hooking library to ARM and do some tests on Android. Because phone vendors usually don't publish their exact kernel config and sources this is a very interesting use case and more work is certainly required (Sylve et al., 2012).

Acknowledgments

We would like to thank Darren Bilby and Heather Adkins for reading a previous version of this paper and giving us valuable feedback and suggestions for improvement.

References

Boileau A. Hit by a bus: physical access attacks with Firewire. Ruxcon; 2006.

Carbone M, Cui W, Lu L, Lee W, Peinado M, Jiang X. Mapping kernel objects to enable systematic integrity checking. In: Proceedings of the 16th ACM conference on Computer and communications security. ACM; 2009. pp. 555–65.

Carrier B, Grand J. A hardware-based memory acquisition procedure for digital investigations. Digit Investig 2004;1(1):50–60.

Case A, Marziale L, Richard GG. Dynamic recreation of kernel data structures for live forensics. Digit Investig 2010;7:S32–40.

Cohen M. PMEM – physical memory driver http://code.google.com/p/volatility/source/browse/branches/scudette/tools/linux; 2011.

Hale M. Linux support in volatility http://code.google.com/p/volatility/wiki/LinuxMemoryForensics; 2013.

Matz M, Hubicka J, Jaeger A, Mitchell M. System V application binary interface http://refspecs.linuxfoundation.org/elf/x86-64-abi-0.99.pdf; 2012.

Raytheon Pikewerks. Linux incident response with second look http://secondlookforensics.com/linux-incident-response/; 2013.

Schatz B. BodySnatcher: towards reliable volatile memory acquisition by software. Digit Investig 2007;4:126–34.

Shoumikhin A. Redirecting functions in shared ELF libraries http://www.apriorit.com/our-experience/articles/9-sd-articles/181-elf-hook; 2010.

Stealth. The Adore-ng rootkit http://packetstormsecurity.com/files/32843/adore-ng-0.41.tgz.html; 2004.

Stüttgen J, Cohen M. Anti-forensic resilient memory acquisition. Digit Investig 2013;10:S105–15.

Styx. Infecting loadable kernel modules, kernel versions 2.6.x/3.0.x. Phrack 2012;0x0e(0x44):0x0b.

Sylve J. LiME – Linux Memory Extractor. In: ShmooCon' 12; 2012.

Sylve J, Case A, Marziale L, Richard GG. Acquisition and analysis of volatile memory from android devices. Digit Investig 2012;8(3):175–84.

The Linux kernel Archives. The Linux kernel source code https://www.kernel.org/pub/linux/kernel/v3.x/linux-3.12.tar.xz; 2013.

The Linux man-pages. kexec_load – load a new kernel for later execution http://man7.org/linux/man-pages/man2/kexec_load.2.html; 2012.

TIS Committee. Tool Interface Standard Executable and Linking Format (ELF) Specification v1.2 http://refspecs.linuxbase.org/elf/elf.pdf; 1995.

Truff. Infecting loadable kernel modules. Phrack 2003;0x0b(0x3d):0x0a.

Walters A. Volatility: an advanced memory forensics framework https://code.google.com/p/volatility/; 2007.

Weidong C, Zhilei X, Marcus P, Ellick C. Tracking rootkit footprints with a practical memory analysis system. In: Proceedings of the 21st USENIX conference on Security symposium. USENIX Association; 2012. p. 42.

You D-H. Android platform based Linux kernel rootkit. Phrack 2012; 0x0e(0x44):0x06.