A Framework for Prototyping and Testing Data-Only Rootkit Attacks

Ryan Riley
ryan.riley@qu.edu.qa
Qatar University
Doha, Qatar

Version 1.0
This is a preprint of the paper accepted in Elsevier Computers & Security

***Abstract*—Kernel rootkits—attacks which modify a running operating system kernel in order to hide an attacker's presence—are significant threats. Recent advances in rootkit defense technology will force rootkit threats to rely on only modifying kernel data structures without injecting and executing any new code; however these data-only kernel rootkit attacks are still both realistic and powerful. In this work we present DORF, a framework for prototyping and testing data-only rootkit attacks. DORF is an object-oriented framework that allows researchers to construct attacks that can be easily ported between various Linux distributions and versions. The current implementation of DORF contains a group of existing and new data-only attacks, and the portability of DORF is demonstrated by porting it to 6 different Linux distributions. The goal of DORF is to allow researchers to construct repeatable experiments with little effort, which will in turn advance research into data-only attacks and defenses.**

***Index Terms*—Malware, Rootkits, Data Attacks, Operating Systems, Security**

I. Introduction

Rootkits are advanced attacks designed to prevent an attacker from being discovered after compromising a system. Over the years they have evolved from simple replacements to key system binaries into complex programs capable of modifying an operating system kernel on the fly.

Research into defense techniques against such advanced threats frequently involves building a defense system prototype that is designed to protect a specific version of an operating system. The sample attacks and real-world rootkits that a researcher may want to test with; however, are frequently hard coded to function on a different version of the OS. As such, in order to test the effectiveness of a research prototype, the researcher is frequently required to "port" those attacks to operate on the same version of the OS used in the prototype. These changes are rarely publicly released, and even if they were they would be unlikely to be helpful to other researchers who would likely be using a different version of the OS for their testing. In addition, if a researcher is unable to get a specific attack to function, it is frequently just left out of testing. This methodology produces unrepeatable results that do not allow for direct comparisons between various defense techniques.

As an example of this problem, consider the variety of experiments performed in research focused on code injection based rootkits for Linux. In [1], the authors build a corpus of 25 rootkit attacks to test with, but end up stating, "Of the 25 that we acquired, we were able to install 18 in our virtual test infrastructure. The remainder either did not support our test kernel version or would not install in a virtualized environment." In [2], the authors test with 16 Linux rootkits, but only 12 of them overlap with [1]. In [3], the authors give no results that involved testing with rootkit attacks.

As another example, in our own work involving kernel rootkit defense [2], [4], [5], [6] we found ourselves locked in to an older version of Linux due to the rootkits we planned to test with. We have a corpus of rootkits which use a variety of attack mechanisms and techniques that we were using to test a variety of defense techniques. Our rootkits, however, were designed to run against a specific version of Linux (Redhat 8.0) and despite multiple attempts at porting them to more up-to-date versions of Linux, in the end we always ended up porting our new defense technique to Redhat 8.0 instead. It was simply less work.

Given the state of experimentation for research into code injection based rootkits, the goal of this work is to improve the quality and ease of testing research into a newer class of attack called data-only rootkits. Advances in kernel rootkit defense [2], [3] have created an environment where a rootkit is unable to inject and execute new code, effectively stopping the vast majority of rootkits. A rootkit attack which relies only on the modification of kernel data structures and not the execution of new code is called a "data-only rootkit attack". For example, a code injection based rootkit would hide a process by bypassing the `readdir()` system call and instead calling a malicious version that filters process entries from the `/proc` filesystem. A data-only attack, by contrast, would remove the target process from the linked list of all tasks located in kernel memory.

In this work we present DORF, the Data-Only Rootkit Framework, an object-oriented framework designed to allow for rapid prototyping and testing of data-only rootkit attacks on Linux. We demonstrate the usefulness of DORF by using it to implement four data-only attacks. We also show the portability of DORF by porting it to a total of 6 different Linux distributions/kernel versions. DORF has two primary goals: First, it allows researchers to focus on developing and testing data-only attacks without being concerned with the layout of kernel data structures or the mechanism used to

access kernel memory. Second, it allows them to test new defense systems with a variety of data-only attacks supplied by other researchers, without needing to re-implement those attacks to function against a different version of the Linux kernel. Instead, the researcher simply ports DORF (if a port is not already available) and then the attacks will "just work." It is our hope that researchers will use and contribute back to DORF in order to further facilitate research into detecting and preventing advanced data-only rootkit threats.

The contributions of this work are as follows:

1) We provide a definition of data-only rootkit attacks and further sub-divide such attacks into control-flow altering and non-control attacks depending on how they influence the control-flow of the kernel. These definitions can be used to help clarify the types of attacks and defenses needed for future work.
2) We present a new data-only attack that allows for file hiding. To our knowledge, this is the first release of such an attack. This attack demonstrates the potential power of data-only rootkit attacks.
3) We present (and release) DORF, the Data-Only Rootkit Framework. The purpose of the framework is to enable researchers to quickly prototype new data-only attacks as well as to allow them to test attacks against new defense systems they may develop.
4) We provide an outline for future research in the area in light of recent research results, including those in this paper. We emphasize four main areas: better semantic specification of kernel data structures, an investigation into the applicability of control- and data-flow integrity, kernel design that does not assume trusted data structures, and better testing of defense systems using a framework such as DORF.

II. History

In this section we will give a brief overview of the history of rootkit attacks.

A. Early Years

Early rootkit attacks against the integrity of the operating system did not focus on the OS kernel, but instead modified key system binaries in order to mask the presence of an attacker. A rootkit would contain modified versions of binaries such as `ls`, `ps`, `top`, `netcat`, and many others. By replacing these binaries, attackers could effectively make specific processes, files, and network connections invisible to an administrator.

To combat these types of attacks, administrators could simply store known good copies of system binaries on read-only media and execute them from there, bypassing the attacker's modifications. Another defense approach is periodic verification of all core system binaries. Systems such as Tripwire[7] create and use a database of file hashes to check the integrity of various files on the filesystem.

B. Modifying Static Kernel Data

Given that Tripwire was able to detect filesystem modifications, attackers needed to find a way to hide themselves without modifying core system files. This led to the creation of rootkits that modified statically allocated data structures in the operating system kernel itself. A common attack was to inject custom code into kernel memory and modify the system call table to cause it to be executed. For example, a modified version of the `readdir` system call could be created to remove references to specific files, hence making them invisible to any system binary that uses the system call. The system call table would then be modified to ensure the malicious system call was used instead of the original.

In response to this type of threat, a number of defense approaches were created. Some systems used rootkit like techniques to read and verify the integrity of the system call table. These approaches relied on an application running on the compromised system to be able to reliably retrieve kernel memory information; however, there was no way ensure the rootkit would not tamper with the rootkit checker. A more resilient approach involved the use of external hardware devices to grab snapshots of memory over the hardware bus and verify the data that way. Copilot[8], for example, uses a PCI card to inspect kernel memory over the PCI-bus. Given that the integrity verifier runs on separate hardware, the rootkit is unable to tamper with it. Other works, such as virtual machine introspection [9], [10], take a similar approach using virtualization.

C. Modifying Dynamic Kernel Data

Due to the fact that static kernel data rarely changes, it is trivial to verify its integrity. There are a large number of kernel data structures, however, which are dynamically allocated and subject to frequent change in both location and content at runtime. For example, the kernel filesystem code allocates dynamic data structures containing function pointers. The kernel networking code also allocates dynamic data structures containing function pointers. Rootkits such as adore-ng inject code into the kernel's memory space and modify these dynamically allocated function pointers in order to call it. Because the function pointers are contained in dynamically allocated data structures, traditional integrity checking approaches were unable to detect the attack.

In response to this threat there are two major classes of defense techniques. The first [11], [1] attempts to detect the attacks by finding a way to verify the semantic integrity of dynamic data structures. The second [4] attempts to validate that data structures are only modified by authorized code.

D. Moving Away From Injected Code

All of the kernel rootkit attacks described thus far focus on modifying kernel memory in order to alter function pointers to allow the execution of malicious rootkit code. SecVisor [3] and NICKLE [2] are two rootkit prevention systems that operate by stopping the execution of injected code at the kernel's privilege level. With this protection in place, existing rootkits are effectively disabled.

In order to overcome this limitation, attackers need to create rootkits that do not rely on running injected code.

III. Data Only Attacks

A rootkit attack that modifies kernel data structures without injecting new code is called a "data-only rootkit attack." Under a data-only attack the attacker has full access to read and write all kernel memory, but is unable to add new code or modify existing code that will be executed with the kernel's privilege level.

Inspired by prior work at the user-level [12], we will further classify data-only rootkit attacks into two types: Control-flow altering data-only rootkit attacks and non-control data-only rootkit attacks.

A *control-flow altering data-only rootkit attack* directly alters the control flow, even if it does not execute new code. Return-oriented [13], [14], [15] and jump-oriented [16] programming techniques, for example, could be used to construct this type of attack. For the rest of this paper we will refer to this type of attack as a CFA attack.

A *non-control data-only rootkit attack*, in contrast, does not directly alter the control-flow, but instead only manipulates data structures. This does not mean that control-flow is in no way altered by these attacks (altering a data structure can change control-flow), it simply means that control-flow is not directly manipulated through a means such as altering the return address of a function, altering a function pointer, etc. We will refer to this type of attack as an NC attack.

We use this classification because it is helpful when discussing attacks as well as potential defense techniques. For example, a defense technique that utilizes control-flow integrity [17] would be effective at stopping a CFA attack, but may be useless against an NC attack.

In this work we are primarily concerned with furthering research into NC attacks, although DORF could be adapted to be suitable for both types.

A. Related Work

There has been a limited amount of research into data-only kernel rootkit attacks from either the hacker or academic research community. We believe this is because there was no need for such attacks until recently. Why spend time doing a more complicated data-only attack when attacks involving injected code work just fine?

Control-Flow Altering Data-Only Rootkit Attacks
Existing work in CFA attacks focuses on altering control-flow in order to re-use existing code.

Initially, the work in this area centered around using existing library code to launch a shell or other simple exploit [18], [19] by placing valid function arguments on the stack, overwriting the return address to point to existing code (such as for the `system()` system call), and then allowing a normal `ret` instruction to cause it to execute. Shacham et. al. [13], [14] formalized and refined this idea and presented reliable and provably complete methods of return-oriented programming, a technique which allows an attacker to execute arbitrary functionality using only existing code. Their work involved placing a carefully crafted sequence of function arguments and return addresses on the stack, effectively creating a chain of functionality. Bletsch et. al. [16] and Checkoway et. al. [15] further expanded the technique and demonstrated a method that does not make use of the return instruction, hence defeating a number of proposed defense systems.

Up until this point, research in this area focused on applying return-oriented programming to processes at the user-level. Hund et. al. [20] applied it to create a return-oriented rootkit capable of modifying kernel data structures. This created a new classes of kernel rootkit, albeit one with limited functionality, that did not rely on the execution of new code. In an extension to that work, Hund [21] also demonstrated a data-only rootkit capable of hiding network connections by altering function pointers to point to existing code used to overwrite the stack pointer and initiate a return-oriented attack. Hund further hypothesized the possibility of using the technique to construct other persistence based attacks such as key logging and file hiding.

Our current framework does not make use on control-flow altering attacks as an attack mechanism; however, their existence caused us to design DORF in such a way that this type of attack could be supported at a later date.

Non-Control Data-Only Rootkit Attacks
Existing work in NC attacks focuses on the details of how kernel data structures are used at runtime and produces techniques for modifying those data structures in order to meet an attacker's goals.

The initial work in NC attacks was done at the user-level by Chen et. al. [12], who demonstrated that NC attacks could still accomplish many of the same goals as their code injection counter-parts. Their work lays the foundation for similar work at the kernel level.

Ubra et. al. [22] analyzed the Linux scheduler and determined that the kernel queues used for process scheduling are not related to the kernel task list used to iterate through all running processes. They developed a data-only attack that hides a process by removing it from the kernel task list, but not removing it from the scheduling queues. They also provided an initial investigation into NC attacks involving the PID hash table (to be described in more detail later). They did not present a data-only implementation of their attacks, however, instead, they loaded a kernel module into memory and performed the attacks from there.

The initial implementation of non-control data attacks from the hacker community came in the form of the FU rootkit [23]. FU demonstrated two potential attacks against a Windows system: Process hiding as well as driver hiding. The implementation, however, does not operate in a data-only way (its loads a kernel driver that performs the attacks) but the attacks could be generalized to function without the loaded driver. In the academic community, Petroni et. al. [11] and Rhee et. al. [4] presented a variety of other attacks including altering and disabling SELinux, the process hiding attack described in [22], and an attack designed to modifying network statistics such as the amount of data transferred.

In this work we present a previously unseen file-hiding attack, and one of the goals of DORF is to provide a framework

that will allow other researchers to design more sophisticated, but currently unknown, advanced capabilities such as network connection hiding, key logging, etc.

B. Example Attacks

In order to demonstrate the potential power of NC attacks, we will now present four samples. Two were taken from some of the related work just described, and two were specifically developed for this paper. (The advanced process-hiding attack described below has been previously investigated in [22], but was not implemented as a data-only attack until now.)

1) Traditional Process Hiding: As described in [22], a traditional process-hiding attack functions by removing a process's `task_struct` from the kernel task list. In Linux kernel version 2.4 and early versions of 2.6 this caused a process to no longer appear to user-space processes such as `ps`.

A pointer to the head of the task list is stored in the static variable `init_task_struct`, and performing this attack is as simple as traversing that linked list, finding the structure representing the process you want to hide, and then removing it from the list. The process will still be scheduled for execution as usual, and can also still be found specifically by PID from user-space processes such as `kill`, however it will no longer appear in the `/proc` filesystem and hence will not be visible to programs like `ps` or `top`.

2) Disable SELinux: In [4] the authors presented an attack capable of disabling SELinux protection on a Redhat 8.0 host. SELinux is implemented in the kernel by making use of the Linux Security Modules (LSM) framework [24]. The LSM framework allows a security module (such as SELinux) to create an array of function pointers to various access control functions that will be called at critical times when accesses are being made.

A pointer to the data structure of function pointers is stored in the static variable `security_ops`. If SELinux is enabled, this points to a data structure containing pointers to SELinux functions. The kernel is also compiled with a static data structure of default function pointers that perform no special permission checks. These pointers are stored in the `default_security_ops` data structure. In order to disable SELinux, one simply needs to point the `security_ops` pointer to the `default_security_ops` data structure. This attack should also be effective at disabling other access control systems that make use of the LSM framework, such as AppArmor [25].

3) Advanced Process Hiding: The process-hiding attack described in Section III-B1 does not function against recent versions of the Linux kernel. Recent versions of the kernel do not derive the collection of running processes by analyzing the task list (although it is still there); instead, the list of running processes is gathered by iterating over the kernel PID hash table. (The PID hash table is a kernel data structure that allows for fast lookups of process table entries given the process's integer PID.)

The kernel exports information about running processes using the `proc` filesystem, a pseudo-filesystem that simply provides information about currently running processes. The `proc` filesystem systematically looks up entries in the PID hash table in order to construct the entries for the filesystem that are ultimately used by user-space tools to display information about currently running processes.

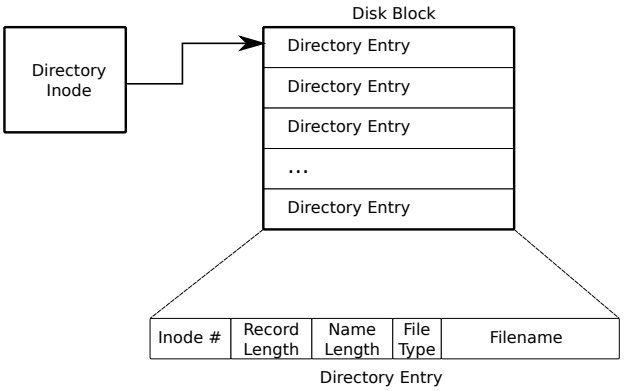

Fig. 1. Directory structure under ext3

In order to hide a process under this system, an attack can be constructed that searches the PID hash table data structures and removes the target process from the hash table.

This, however, has side-effects. When compared to the rarely used process list, the PID hash table is used frequently. This causes interference with some programs. Bash, for example, terminates once it is removed from the hash table due to the fact that kernel calls such as `find_task_by_pid()` will no longer function properly for the hidden process (`find_task_by_pid()` looks up a given process by searching the PID hash table, which now no longer contains the process) and this interferes with job control [22].

4) File Hiding: File hiding is a classic feature of rootkits, and until now a technique to hide files using only NC attacks has not been known. Traditionally, kernel rootkits hide files by hooking either the readdir system call or one of the function pointers in the dynamically allocated filesystem data structures. The rootkit will provide an alternative function that will execute the original call and then selectively filter the results before returning them to user-space. In this way, user-level programs such as `ls` will not see a file the attacker intends to hide.

When using an NC attack, however, we cannot install any alternate code. As such, we must find a way to hide files while only modifying kernel data structures.

Consider the ext3 file system as an example. A directory is represented by an inode, and that inode points to disk blocks that hold a listing of all files contained in that directory. Figure 1 illustrates the layout. If a file is deleted, the disk block containing the listing is modified to remove the entry for that file (in addition to the other things done to free the file, such as freeing its disk blocks and inode).

Due to the fact that disk reads and writes are slow, the Linux kernel maintains a cache of recently accessed disk blocks. If any modifications are made to a cached block then it is marked "dirty" in order to ensure it will get written back to disk later. (Blocks not marked dirty are not written back to disk.) This

cache can provide an avenue of attack for hiding a file. An attacker could locate the cached disk block entry that maintains the directory listing containing the file we wish to hide and remove that file from the listing. When a call to ls or a similar program is performed later, the cached version of the block will be used and the hidden file will not appear. In addition, as long as the dirty bit is not set for our modified block, the changes will not be made to the physical file system.

This approach, while simple in principal, is not without its challenges. First, searching the block cache by simply analyzing the contents of memory is difficult. The block cache is indexed in tree structures and integrated with the page cache of the Linux kernel. Second, if the block is modified in memory by our rootkit and is also later modified by a legitimate file operation, the block will be marked dirty and our temporary attack will become a permanent modification to the filesystem when then block is written back, such as during a sync operation. (Our hidden file would effectively be deleted.) Third, if the disk block is evicted from the cache due to memory pressure, then when the directory is read again a new cache item will be read from disk, and our changes will not be there.

IV. Experimental Framework

As can be seen in Section III-B, NC data-only attacks have the potential to be powerful. In addition, the related work of Section III-A reveals that there is very little research into more advanced attacks.

In order to promote further research into this area, we have created DORF, an object-oriented, extensible framework which allows for rapid prototyping and validation of these data-only attacks against Linux systems. DORF is written in Python and is available for download [26].

The goal of DORF is to allow researchers to quickly prototype and test a data-only attack without tying their attack to a specific attack mechanism regarding how kernel objects are found and manipulated. In addition, the framework will allow researchers to test new defense systems against a corpus of real attacks contributed by others without the need to port individual attacks to their testing environment. It is our hope that other researchers in this area will use DORF and contribute back to it.

A. Architecture Overview

The high-level architecture of DORF can be seen in Figure 2. The system contains four major components: The attacks, the kernel data structure module, the memory access module, and the symbol finder module. When porting DORF to a new distribution or kernel version, the kernel data structure module is modified to reflect any changes made to data structure definitions as well as ordering of members in memory. When altering the attack mechanism (such as changing the method by which the attacks access kernel memory) the memory access module and symbol finder module should be modified. The attacks themselves, however, do not required modification when porting or altering the attack mechanism. In this way a researcher can port DORF to a new platform and use a new memory access technique without needing to modify the existing attacks. These means that different researchers testing their defense systems with different versions of Linux can still test their systems using the same attacks with minimal effort.

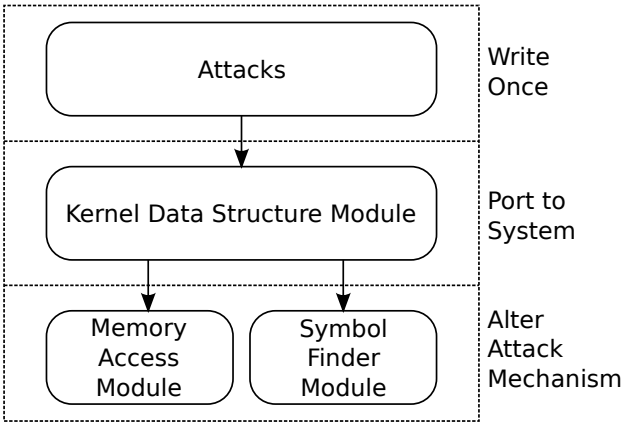

Fig. 2. Overview of DORF Architecture

B. Framework Modules

DORF is designed around three core modules that are meant to be interchanged based on the specific attack methods and targets.

1) Memory Access Module: A Memory Access Module (`MemAccess`) provides access to kernel memory. It can be used to read/write objects from memory based on the virtual address where those objects are located.

It has the following methods available:

- `read(dest, addr)`: Read a data object, `dest`, starting at address `addr`. `dest` should be a kernel data structure defined elsewhere in the framework (see Section IV-B3). `addr` should be a virtual address located in kernel memory.
- `write(src, addr)`: Write a data object, `src`, to address `addr`.

In the current version of the framework, a memory access module is provided that makes use of `/dev/mem` in order to provide kernel memory access. It also performs page table lookups to map virtual addresses to physical addresses.

2) Symbol Finder Module: A Symbol Finder Module (`SymbolFinder`) provides a mechanism for performing name-based lookups of statically allocated kernel variables. For example, the root of the PID hashtable is stored in a static variable named `pid_hash`. A data-only rootkit would need to find the address of this variable in order to start its search of that hash table.

It has the following methods available:

- `find(symbol)`: Return the address of the kernel symbol named by the string `symbol`.

In the current version of the framework, a symbol finder module is provided that makes use of `/proc/kallsyms`, a kernel interface that allows kernel modules to find kernel objects. Other symbol finder modules could also be written that

would do lookups based on `System.map` (a file generated at compile time and commonly stored in `/boot`) or a detailed analysis of the kernel's binary code, similar to that performed by the SucKIT rootkit [27].

3) Kernel Data Structure Definitions: The binary layout of kernel data structures may vary between kernel versions and architectures. As such, a kernel data structure definition file is created to specify the layout of important data structures for the kernel version being attacked. The data structures are defined in Python using the `ctypes` library. Any data structure needed by a rootkit attack should be defined here. For example, Figure 3 is the definition of an `hlist_node`, a data structure embedded in other data structures in order to provide a doubly linked list. It defines the elements of the data structure using the `_fields_` member and also provides a `remove()` method to remove that entry from the list it is a part of.

Once a kernel data structure has been declared, it can also be used in other data types. For example, there is an `hlist_node` named `pid_chain` inside of the PID data structure in Figure 4 (`junk` entries in a definition are used to hold the place of data structure entries not presently needed for attacks).

C. Designing Attacks

In this section we will present DORF implementations of the attacks described in Section III-B. The purpose of these examples is to show the ease with which new attacks can be prototyped using the framework.

1) Traditional Process Hiding: Figure 5 shows the DORF code needed to hide a process by removing its entry from the kernel's task list. The code functions by using the Symbol Finder to find the pointer to the head of the task list (`init_task`) and then traverses the list searching for the process to remove. Once it finds the task structure of the target process, it calls the `remove()` method of the object's `tasks` element to remove it from the list.

```
class hlist_node(Structure):
    _fields_ = [     ('next', c_uint),
                     ('pprev', c_uint)
                ]

    def remove(self, dm):
        pprev = hlist_node()
        next = hlist_node()

        if self.pprev != 0:
            dm.read(pprev, self.pprev)
            pprev.next = self.next
            dm.write(pprev, self.pprev)

        if self.next != 0:
            dm.read(next, self.next)
            next.pprev = self.pprev
            dm.write(next, self.next)
```

Fig. 3. Definition of hlist_node

```
class pid(Structure):
    _fields_ = [     ('junk1', c_char * 28),
                     ('nr', c_int),
                     ('ns', c_void_p),
                     ('pid_chain', hlist_node)
                ]
```

Fig. 4. Definition of pid

```
# Removes an item from the task list.
# argv[1] contains the PID of the process
# to remove.

the_pid = int(sys.argv[1])

sf = SymbolFinder()
dm = DevMemReader(sf)

p = task_struct()

init_task = sf.find("init_task")
dm.read(p,init_task)
dm.read(p,p.next())

while p.pid != 0:
    dm.read(p,p.next())
    if p.pid == the_pid:
        p.tasks.remove(dm)
        break
```

Fig. 5. Process-hiding attack

2) Disable SELinux: Figure 6 shows the very small amount of DORF code required to disable SELinux. It first finds the locations of both `security_ops`, the pointer to the data structure of function pointers currently being used for access control checking, and `default_security_ops`, the default data structure that contains null function pointers. It then causes `security_ops` to point to `default_security_ops`, effectively disabling SELinux (or any other LSM based security module).

3) Advanced Process Hiding: Figure 7 demonstrates a more advanced process-hiding attack that removes an item from the PID hash table. It first instantiates a PidHash object, which automatically searches for and finds the kernel PID hash

```
# Disables SELinux.

sf = SymbolFinder()
dm = DevMemReader(sf)

secops = sf.find("security_ops")
def_secops = sf.find("default_security_ops")

dm.write(c_uint(def_secops), secops)
```

Fig. 6. Disabling SELinux

```
# Removes an item from the PID hash table
# argv[1] contains the PID of the process
# to remove.

the_pid = int(sys.argv[1])

sf = SymbolFinder()
dm = DevMemReader(sf)

ph = PidHash(sf,dm)
t = ph.find(the_pid)
if t != None:
    t.pid_chain.remove(dm)
```

Fig. 7. Advanced process-hiding attack

table. It then searches the table for the PID of the process to hide, returning a `task_struct`. It then removes that `task_struct` from the hash table using the `remove()` method. The framework handles the heavy lifting of searching the hashtable as well as removing an entry in it.

4) File Hiding: As mentioned in Section III-B4, file hiding involves manipulating the kernel's disk block cache in memory in order to remove the target file's entry from the disk block that stores its directory listing. As long as the block is not flushed back to disk, the hidden file is only temporarily hidden, and is not actually deleted. If the block is flushed to disk, then the file's data blocks and inode will remain allocated but inaccessible.

When the kernel adds a disk block to the memory cache it creates a `buffer_head` to store information like the disk block number, the dirty bit, and a pointer to the page in memory where the data is stored. Cached `buffer_head` objects are stored in the kernel's page cache and ultimately indexed using a tree structure. In order to optimize lookup times for frequently used `buffer_heads`, the last 8 used are stored in a statically allocated global array called `bh_lrus`.

Figure 8 shows the file-hiding attack. In order to quickly and easily find the `buffer_head` related to the directory information we are looking for, the attack code does a directory listing of the target file's parent directory (causing the disk block containing the directory information to be put to the front of `bh_lrus`) and then can immediately access the appropriate `buffer_head` by finding it at the head of `bh_lrus`.

After finding the appropriate disk block, the attack then searches the block for the file we wish to hide and removes it from the directory listing by modifying record-length indicators to "skip" the entry we wish to hide. DORF's kernel data structure module provides methods to search and manipulate directory entries, simplifying the attack code and enhancing its portability.

After the attack occurs, the file no longer appears in any directory listings; however, it can still be accessed directly using its complete path. This means an attacker can access the file despite the fact it cannot be seen through a file listing.

There are restrictions regarding how this attack should be used. In order to prevent our cached block from being marked

```
# Hide a file.  argv[1] stores the name+path
# of the file to hide

# Do some work on the pathname
full_path = os.path.abspath(sys.argv[1])
basename = os.path.basename(full_path)
dirname = os.path.dirname(full_path)
if os.path.exists(dirname) == False:
    print "Parent directory does not exist!"
    sys.exit(-1)

# Declare necessary pieces for this attack
sf = SymbolFinder()
dm = DevMemReader(sf)
bh = buffer_head()

# Find the recently used list
bh_lrus = bh.get_addr_first_bh(sf)

# Read the item's parent directory
print os.listdir(dirname)

# Read in the first buffer_head from the LRU
bh_addr = dm.read_int(bh_lrus)
dm.read(bh, bh_addr)

# Go through the directory block and find the
# entry, then remove it.
de = dir_entry(dm)

# Find the memory location of the dir_entry
# for the file we want.
loc = de.find_fname(bh.b_data, \
                    bh.b_data+bh.b_size, \
                    basename)

# Bypass that entry in the inode, then
# increase the usage count of the buffer it
# is stored in to prevent eviction.
if loc != 0:
    de.remove(bh.b_data, loc)
    bh.b_count = bh.b_count+1
    dm.write(bh,bh_addr)
```

Fig. 8. File-hiding attack

dirty, our hidden file should be stored in a directory that does not experience frequent changes. A directory such as `/usr/share/man` would be a good choice, as it is only updated when software is installed or removed. We also need to ensure that our cached block is not evicted and a new copy read in from disk. To do this we simply increment the `buffer_head`'s usage counter, making the kernel believe it is always in use and hence not available to be reclaimed.

Distribution	Release Date	Kernel	Lines of DORF Code Changed
Ubuntu 9.10	October 2009	2.6.31-20-generic	-
Ubuntu 10.04	April 2010	2.6.32-33-generic	5 / 530 (0.95%)
OpenSuse 11.4	March 2011	2.6.37.1-1.2	7 / 530 (1.3%)
Fedora Core 15	May 2011	2.6.40.4-5	7 / 530 (1.3%)
CentOS 6	July 2011	2.6.32-71.29.1.el6	6 / 530 (1.1%)
Ubuntu 11.10	October 2011	3.0.4	7 / 530 (1.3%)

TABLE I
RESULTS OF DORF PORTING

D. Implementation

The initial implementation of DORF was tested using Ubuntu 9.10 running kernel 2.6.31-20-generic and consisted of 530 lines of Python code. In order to demonstrate the portability of the system, it was then ported to five other Linux distributions and kernel versions, as shown in Table I. The kernels have all been recompiled to offer `/dev/mem` access to all of physical memory. (The kernel included with OpenSuse did not require this modification because access to `/dev/mem` is included by default.)

The different distributions used for testing were released over a two-year period from October 2009 to October 2011, and vary in Linux kernel versions from 2.6.31 to 3.0. Porting the DORF system from the reference implementation to the additional distributions required less than seven lines of change in each case. None of the changes required modifying the attacks themselves; instead, most of the changes were related to specifying the offsets in memory of members of kernel data structures. In one case the name of a kernel data structure changed. This demonstrates that DORF meets its goal of being easily portable to a variety of Linux distributions without requiring changes to the attacks themselves. Subjectively, the porting process was straightforward, and a new port required less than 30 minutes to create once the distribution was configured and running. A Linux kernel module created to aid in finding the order and offset of certain data structure elements is also included with the framework, and greatly speeds up porting efforts.

Future Extensions to DORF We anticipate future versions of the framework to be extended to include alternative memory access modules (such as by using return-oriented programming) as well as a broader array of attacks. We encourage researchers in this area to extend the framework by contributing their changes for the benefit of the research community.

V. DISCUSSION

In this section we will discuss the limitations of DORF, give an overview of the existing work in defense against data-only attacks, and give our recommendations for future work in the area.

A. Limitations

This work is not without its limitations. We will now discuss concerns raised by the authors and others during the development of DORF.

Alternate operating systems DORF has been designed and implemented specifically for Linux based attacks and does not currently support other operating systems. The concepts discussed here; however, are applicable to other OSes such as Windows. Various versions of Windows and service pack releases can alter the layout of kernel data structures, necessitating porting of data-only attacks. Due to the significant differences between data structures and basic OS design between Linux and Windows, we do not think that alternate OS support would be a simple extension to the existing framework. There simply would not be enough attacks that could function properly on both systems if the data-structure module is the only thing to distinguish them. Instead, we envision an alternate framework focusing on Windows itself.

Use of /dev/mem The current version of DORF directly manipulates kernel memory through the raw memory access device `/dev/mem`. Some may object that since this device is disabled by default in many distributions, DORF is irrelevant as the attacks are not realistic. There are two responses to this concern. First, and most importantly, DORF is primarily concerned with furthering research into what needs to be modified in kernel memory to perform NC attacks; it is not primarily concerned with the mechanism by which kernel memory is modified. In the same way, proper defense solutions should focus not on the mechanism, but the modifications that are enabled by them. This means DORF is still extremely useful for testing new defense systems. Second, the modular nature of DORF would allow someone to write a new memory access module (such as one making use of a return-oriented attack) that does not require `/dev/mem`. Once such a module has been written, existing attacks will function with it without modification, which is main purpose of DORF.

B. Existing Work

Given that one of the goals of this work is to promote better research into data-only attack prevention and detection techniques, we now present existing work in that area.

Most of the existing work focuses on techniques for specifying and validating the "correct" state for a dynamic kernel data structure. Works of this type can be useful for defending against both CFA and NC attacks. Petroni et. al. [11] present a specification language-based approach to to verifying the semantic integrity of dynamic kernel data structures. Their work provides a strong foundation for future work that needs to be done to formally specify semantic integrity rules for all kernel data structures. Hofmann et. al. [28], much like Petroni et. al., present a system to systemically verify the state of kernel data objects based on specified rules. They also present

a solution to the problem of verifying kernel memory while it is being used, even if it is temporarily inconsistent. This is an important improvement required for live-validation of a running system. Cozzie et. al. [29] and Dolan et. al. [30] take a slightly different approach and focus on producing signatures of kernel data structures in order to search for them in memory and detect rootkit attempts to hide them. The results are promising, but the effectiveness of the technique against a wide variety of kernel data structures has yet to be tested.

Another type of approach from Petroni et. al. focuses on detecting control flow modifications [1]. This type of approach would be useful for detecting CFA attacks, but not NC attacks. However, as part of their work they present a method for traversing dynamic kernel data structures that we believe will be useful for future research in both CFA and NC defense.

Rhee et. al. [4] demonstrate a system capable of monitoring dynamic data structures by intercepting writes to those memory addresses and verifying their validity. Effective use of the system, however, would require complete enumeration of all kernel data structures. This would be challenging to do by hand, but Ibrahim et. al. [31] propose a method to systematically provide a comprehensive kernel data definition for an OS kernel by only analyzing the source code. The combination of these two ideas could provide an interesting approach to defense. However, due to the way Rhee et. al. validate kernel writes, this combined technique would still only be helpful against NC attacks.

DORF is not a rootkit defense system, but it does aim to provide a framework that would allow researchers such as these to better test the effectiveness of their defense systems against a variety of attacks. Right now each of these works uses their own, unique testing methodology that would make direct comparisons difficult or impossible. DORF aims to change that for future work.

C. Applications for DORF

As mentioned previosuly, we envision DORF being used to further research into data-only rootkit attacks and defenses. For this to occur, we are relying on other researchers to make use of DORF and contribute their modifications back to the community. To facilitate this, the DORF source code is made available under an open source license and the development repository is publicly hosted [26].

Given DORF's modular nature, there are three types of contributions we envision researchers making as they use DORF for their testing:

1) Porting DORF to other versions of the Linux kernel by properly specifying data structure offsets and formatting for that specific version. Currently DORF has the files for compatibility with 6 different versions of the kernel. More are still needed.
2) Implementing existing, and designing new, data-only attacks using DORF. Currently DORF has four sample attacks. We would anticipate future work drastically increasing the number of attacks.
3) Providing new memory access modules that make use of alternative attack mechanisms such as return-oriented programming, traditional buffer overflows, etc. This will give DORF a wider variety of mechanisms, making it useful for testing a larger variety of defense systems.

As more researchers use and contribute to DORF, its effectiveness will continue to grow.

D. Recommendations for Future Work

Given all of this, new defense approaches are needed that focus not on the mechanism of how memory is manipulated (there is no reason to believe that new techniques will not soon emerge) but instead on detecting and preventing these attacks based on analyzing the kernel data structures themselves. This is especially difficult for dynamically-allocated, constantly changing data structures such as the disk block cache. Here we provide a few hints regarding research that we believe may help solve this problem.

First, the limited number of existing works that have applicability for prevention and detection of data-only attacks [11], [1], [4], [28] should be evaluated in light of a corpus of more complex attacks than those in existence when they were created. This will require additional research into more sophisticated data-only attacks, such as the file-hiding attack described in this work.

Second, work should be done to establish a method for defining the semantics and usefulness of all kernel data structures. These techniques should involve augmenting data structure definitions with information such as data structure relationships (e.g., all objects on list A should also be on list B) as well as describing ways to analyze the data structure to verify its integrity. Initial work in this area [11] is limited and requires a significant amount of manual intervention, or is simply not yet fully formed [31]. A full specification and automated verifier to detect integrity verifications would be ideal.

Third, work should be done to investigate the applicability of kernel level implementations of control-flow integrity [17] for CFA attacks, and data-flow integrity [32] for both CFA and NC attacks. Both techniques offer features and benefits that may be applicable to this space.

Finally, effort should be spent analyzing the possibility of "defensive coding" of OS kernels. Right now, the Linux kernel assumes that all data structures are trusted. Could it be modified to behave otherwise? For example, if the kernel verified that a process was part of the process list and the PID hash table prior to executing it, then process hiding attacks would not be possible (hidden processes would never run). This would relate strongly to the first recommendation above (better definitions of data structures) in order to ensure that all data structures storing a representation of an item are verified before that item is used/trusted.

References

[1] N. L. Petroni, Jr., M. Hicks, Automated Detection of Persistent Kernel Control-Flow Attacks, in: Proceedings of the 14th ACM Conference on Computer and Communications Security, 2007.
[2] R. Riley, X. Jiang, D. Xu, Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing, in: Proceedings of Recent Advances in Intrusion Detection, 2008, pp. 1–20.

[3] A. Seshadri, M. Luk, N. Qu, A. Perrig, SecVisor: A Tiny Hypervisor to Guarantee Lifetime Kernel Code Integrity for Commodity OSes, in: Proceedings of the ACM Symposium on Operating Systems Principles, 2007.

[4] J. Rhee, R. Riley, D. Xu, X. Jiang, Defeating Dynamic Data Kernel Rootkit Attacks via VMM-Based Guest-Transparent Monitoring, in: Proceedings of the International Conference on Availability, Reliability and Security (ARES 2009), 2009.

[5] R. Riley, X. Jiang, D. Xu, Multi-Aspect Profiling of Kernel Rootkit Behavior, in: Proceedings of the 4th European Conference on Computer Systems, 2009.

[6] J. Rhee, R. Riley, D. Xu, X. Jiang, Kernel Malware Analysis with Un-tampered and Temporal Views of Dynamic Kernel Memory, in: Proceedings of the 13th International Symposium on Recent Advances in Intrusion Detection, 2010.

[7] G. H. Kim, E. H. Spafford, The design and implementation of tripwire: a file system integrity checker, in: CCS '94: Proceedings of the 2nd ACM Conference on Computer and communications security, ACM, New York, NY, USA, 1994, pp. 18–29.

[8] N. L. Petroni, T. Fraser, J. Molina, W. A. Arbaugh, Copilot: A Coprocessor-based Kernel Runtime Integrity Monitor, in: Proceedings of the 13th USENIX Security Symposium, 2004, pp. 179–194.

[9] T. Garfinkel, M. Rosenblum, A Virtual Machine Introspection Based Architecture for Intrusion Detection, in: Proceedings of the Network and Distributed Systems Security Symposium, 2003.

[10] X. Jiang, X. Wang, D. Xu, Stealthy Malware Detection through VMM-based "Out-of-the-Box" Semantic View Reconstruction, in: Proceedings of the 14th ACM Conference on Computer and Communications Security, 2007.

[11] N. L. Petroni, Jr., T. Fraser, A. Walters, W. A. Arbaugh, An Architecture for Specification-based Detection of Semantic Integrity Violations in Kernel Dynamic Data, in: Proceedings of the 15th USENIX Security Symposium, 2006.

[12] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, R. Iyer, Non-Control-Data Attacks Are Realistic Threats, in: Proceedings of the 14th USENIX Security Symposium, 2005.

[13] H. Shacham, The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (On the x86), in: Proceedings of the 14th ACM Conference on Computer and Communications Security, 2007.

[14] E. Buchanan, R. Roemer, H. Shacham, S. Savage, When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC, in: Proceedings of the 15th ACM Conference on Computer and Communications Security, 2008.

[15] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, M. Winandy, Return-oriented programming without returns, in: A. Keromytis, V. Shmatikov (Eds.), Proceedings of CCS 2010, ACM Press, 2010, pp. 559–72.

[16] T. Bletsch, X. Jiang, V. W. Freeh, Z. Liang, Jump-oriented programming: a new class of code-reuse attack, in: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ASIACCS '11, ACM, New York, NY, USA, 2011, pp. 30–40.

[17] M. Abadi, M. Budiu, Úlfar Erlingsson, J. Ligatti, Control-Flow Integrity: Principles, Implementations, and Applications, in: Proceedings of the 12th ACM Conference on Computer and Communications Security, 2005.

[18] Nergal, Advanced return-into-lib(c) exploits (PaX case study), Phrack 11 (58), article 4.

[19] Solar Designer, Getting around non-executable stack (and fix), Bugtraq Mailing List. August 10, 1997. http://seclists.org/bugtraq/1997/Aug/63.

[20] R. Hund, T. Holz, F. C. Freiling, Return-oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms, in: Proceedings of the 18th USENIX Security Symposium, 2009.

[21] R. Hund, Countering lifetime kernel code integrity protections, Diploma thesis, Universität Mannheim, http://pil.informatik.uni-mannheim.de/filepool/theses/diplomarbeit-2009-hund.pdf (May 2009).

[22] ubra from PHI Group, hiding processes (understanding the linux scheduler), Phrack 11 (63), article 12.

[23] P. Silberman, C.H.A.O.S., FUTo, Uninformed 3, http://uninformed.org/?v=3&a=7&t=sumry. Last accessed December 2011.

[24] C. Wright, C. Cowan, J. Morris, S. Smalley, G. Kroah-Hartman, Linux Security Modules: General Security Support for the Linux Kernel, in: Proceedings of the 11th Usenix Security Symposium, 2002.

[25] M. Bauer, Paranoid Penguin: An Introduction to Novell AppArmor, Linux Journale 2006 (2006) 13–, http://www.linuxjournal.com/article/9036.

[26] R. Riley, dorf - data-only rootkit framework https://www.github.com/rriley/DORF/. Last accessed October 2012.

[27] sd, devik, Linux on-the-fly Kernel Patching without LKM, Phrack 11 (58), article 7.

[28] O. S. Hofmann, A. Dunn, S. Kim, I. Roy, E. Witchel, Ensuring Operating System Kernel Integrity with OSck, in: Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, 2011.

[29] A. Cozzie, F. Stratton, H. Xue, S. T. King, Digging for data structures, in: Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2008.

[30] B. Dolan-Gavitt, A. Srivastava, P. Traynor, J. Giffin, Robust signatures for kernel data structures, in: Proceedings of the 16th ACM Conference on Computer and Communications Security, ACM, 2009, pp. 566–577.

[31] A. Ibrahim, J. Hamlyn-Harris, J. Grundy, M. Almorsy, Operating system kernel data disambiguation to support security analysis, in: Proceedings of 6th International Conference on Network and System Security (NSS), 2012.

[32] M. Castro, M. Costa, T. Harris, Securing Software by Enforcing Data-Flow Integrity, in: Proceedings of the USENIX Symposium on Operating Systems Design and Implementation, 2006.