Defeating Kernel Driver Purifier

Jidong Xiao[1] and Hai Huang[2] Haining Wang[3]

[1] College of William and Mary, Williamsburg VA 23185, USA,
[2] IBM T.J. Watson Research Center, Hawthorne NY 10532, USA,
[3] University of Delaware, Newark DE 19716, USA,

Abstract. Kernel driver purification is a technique used for detecting and eliminating malicious code embedded in kernel drivers. Ideally, only the benign functionalities remain after purification. As many kernel drivers are distributed in binary format, a kernel driver purifier is effective against existing kernel rootkits. However, in this paper, we demonstrate that an attacker is able to defeat such purification mechanisms through two different approaches: (1) by exploiting self-checksummed code or (2) by avoiding calling kernel APIs. Both approaches would allow arbitrary code to be injected into a kernel driver. Based on the two proposed offensive schemes, we implement prototypes of both types of rootkits and validate their efficacy through real experiments. Our evaluation results show that the proposed rootkits can defeat the current purification techniques. Moreover, these rootkits retain the same functionalities as those of real world rootkits, and only incur negligible performance overhead.

1 Introduction

Modern operating systems are often divided into a base kernel and various loadable kernel modules. Kernel drivers are often loaded into the kernel space as modules. The ability to quickly load and unload these modules makes driver upgrade effortless, as the new code can take an immediate effect without rebooting the machine. While the base kernel is trusted, kernel drivers are sometimes released by third-party vendors (i.e., untrusted) in binary format. This creates a problem as it is much more difficult to detect malicious code at the binary level than at the source level. Therefore, kernel drivers have been heavily exploited for hosting malicious code in the past. Sony's infamous XCP rootkit in 2005 [1, 22] and its USB device driver rootkit in 2007 [18] have exemplified this risk. In addition, kernel drivers, which constitute 70% of modern operating system's code base [16],are a significant source of software bugs [7, 10], making them substantially more vulnerable to various malicious attacks than the base kernel.

During an attack, once an attacker gains root access, rootkits are then installed to hide their track and provide backdoor access. Rootkits normally hook to the kernel and modify its data structures such as system call table, task list, interrupt descriptor table, and virtual file system handlers. Rootkits can be either installed as a separate kernel module, or injected into an existing kernel module. To protect against rootkits, different defense mechanisms have been proposed

and can be categorized into two basic approaches: kernel rootkit detection and kernel module isolation. In the former, various detection frameworks are created using either an extra device to monitor system memory [23] or virtual machine introspection techniques [9, 15]. And in the latter, strict isolation techniques are introduced to further isolate kernel modules from the base kernel [5, 30].

While the idea of enhancing the isolation of kernel drivers has been extensively studied in the past, it has not yet been widely adopted by mainstream operating systems. One of the key reasons is that it involves too much re-implementation effort. Instead of isolating kernel drivers, safeguarding a kernel driver itself looks more promising. As kernel drivers run at the same privilege level as the base kernel, one can achieve this goal by detecting and eliminating malicious code from kernel drivers before they are loaded into the kernel space. This technique is called kernel driver purification. Based on this design principle, Gu et al. [11] proposed and implemented a kernel driver purification framework, which aims to detect malicious/undesirable logic in a kernel driver and eliminate it without impairing the driver's normal functionalities. Their experimental results demonstrate that this technique can purify kernel drivers infected by various real world rootkits. However, we observe that there are two approaches which attackers can employ to defeat such a technique. The first approach uses self-checksum code to protect malicious kernel API calls, and the second approach is to simply avoid using kernel API calls altogether when writing a rootkit. We show that both approaches can effectively defeat current kernel driver purifiers.

The major contributions of our work are summarized as follows:

- We first present a self-checksum based rootkit that is able to evade the detection of current kernel driver purifiers. While self-checksum has long been proposed as a way to protect benign programs, as far as we know, we are the first to use it for hiding kernel rootkits. We also develop a compiler level tool, with which, attackers can automatically re-write existing rootkits and convert them into self-checksum based variants that are resistant to kernel driver purifiers.
- We present another approach of creating a more stealthy rootkit, which avoids using kernel API calls. While our first approach attempts to protect malicious kernel API calls from being removed by kernel driver purifiers, this new type of rootkit demonstrates that most kernel API calls can be avoided, and thus making the kernel driver purifier completely ineffective.
- We evaluate the functionality and performance of both rootkits. Our experimental results show that the presented rootkits maintain the same set of functionalities as most real world rootkits have and only incur minor performance overhead.

2 Background

Kernel drivers have always been a major source of kernel bugs and vulnerabilities, and improving their reliability has drawn significant attentions from the research community. The kernel driver purification technique has been shown to effectively sanitize kernel drivers infected by existing rootkits. In this section, we

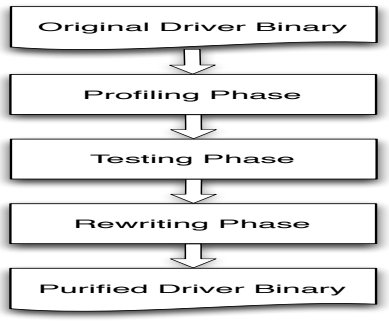

Fig. 1: Overview of kernel driver purification

present a brief overview of the kernel driver purification technique. The design principle of kernel driver purification is based on two observations. First, malicious/undesirable logic embedded within a kernel driver is normally orthogonal to the driver's base functionalities; second, its malicious goal is mainly achieved by interacting with the base kernel via kernel API invocations.

Figure 1 illustrates how a kernel driver purifier works at a high level with three different phases: profiling, testing, and rewriting.

- In the profiling phase, test cases are selected to exercise the common code paths of the kernel driver, where all kernel API invocations and return values are recorded. One key technique used in this phase is called driver-kernel interaction tracking. To record the kernel API invocations and return values, it is crucial to detect the transitions between the driver and the kernel code. This is achieved by monitoring the program counter: if the current basic block is within the driver's memory space but the previous basic block does not, it indicates that the control flow is transitioned from the kernel into the driver, and vice versa. Exploiting these observations, one can track all the transitions between the driver and the kernel code.
- In the testing phase, a subset of the kernel API invocations detected from the profiling phase are removed. If all the test cases complete successfully, these kernel API invocations can be viewed as not affecting the correct execution of the driver code, and therefore, they are marked as non-critical. If any test fails or the system crashes/halts/reboots, a divide-and-conquer approach is then used to narrow down the offending kernel API invocation. In the end, a list of non-critical kernel API invocations and their addresses are noted.
- In the rewriting phase, those non-critical kernel API invocations are removed from the binary kernel driver, and a new binary file, i.e., the purified driver binary file, is generated for use.

The above procedure ensures that malicious logic is removed while benign logic is maintained in the newly generated kernel driver. Although there are several limitations to this approach, including test coverage, false positives of the removed kernel API invocations, etc., the approach has been proven [11] to

be very effective in purifying trojaned drivers that have been infected by various real world rootkits.

However, the two fundamental assumptions of this approach, i.e., (1) the removal of kernel API calls made by the malicious code will not affect the base functionality of the kernel driver and (2) rootkits have to call kernel APIs to achieve its malicious goals, are challengeable. Although they might be true for existing rootkits, we will show that generic enhancements to these rootkits will void these assumptions, and thus, rendering the purification process ineffective.

3 Attack Model

In this section, we present our attack methods for defeating kernel driver purifier. We will use KBeast [14] as an example as it has been widely used in case studies in the past [32, 28]. KBeast is a Linux kernel rootkit that hijacks system calls, and it allows attackers to provide their own system call functions. By doing so, attackers can hide malicious kernel modules, files, directories, processes, sockets, and active network connections. Moreover, KBeast provides keystroke logging, anti-kill, anti-delete, and anti-remove functions.

Below we show a snippet of KBeast's code, which is the malicious version of the *unlink* system call. This function executes the real sys_unlink call when removing normal files and directories, but denies those requests that attempt to remove malicious files and directories.

```
1 asmlinkage int h4x_unlink(const char __user *pathname) {
2  int r;
3  char *kbuf=(char*)kmalloc(256,GFP_KERNEL);
4  copy_from_user(kbuf,pathname,255);
5  if(strstr(kbuf,_H4X0R_)||strstr(kbuf,KBEAST)){
6    kfree(kbuf);
7    return -EACCES;
8  }
9
10  r=(*o_unlink)(pathname);
11  kfree(kbuf);
12  return r;
}
```

The code above performs the following operations. Line 1: the start of the function definition. Line 2 to Line 4: allocate a memory buffer and use that memory buffer to store the pathname of a file. Line 5 to Line 8: compare the pathname with some predefined values to determine whether or not the file/directory in the request is malicious, and if so, return and release the memory buffer. Line 10: for any other ordinary files, invoke the original system call, i.e., sys_unlink. Line 11 to Line 12: free the memory buffer and return. A driver purifier would remove the kernel API calls such as kmalloc(), copy_from_user(), strstr(), and kfree() used by the malicious code. When these function calls are removed, the rootkit would lose its functionality partially or even completely. Therefore, from the attacker's perspective, it is important to protect these calls from being removed, or implement the rootkit without calling these kernel functions.

3.1 Self-Checksum Based Rootkit

We first present a self-checksum based approach to protect kernel API calls in a rootkit from being purified. To do so, there are at least two different strategies we can employ.

The first strategy is more straightforward, which is to add a conditional statement after each kernel API invocation. The conditional statement would test whether or not the preceding kernel API invocation is executed. It does nothing if the call is executed as expected. However, if the call is not executed (i.e., dynamically removed by a kernel driver purifier), it will trigger something abnormal, such as crashing the system or causing other types of failures that would result in the test cases to fail.

However, we found that it is very challenging to track whether or not a kernel API function is called. This is because a kernel driver purifier can store the return value of each kernel API invocation, and for each removed API function invocation, it could fill the stored return value in the EAX register as if the function was invoked and returned. Thus, we cannot determine if the function is really called by checking the return value.

The key idea of our proposed approach is that when a kernel module is loaded into the kernel, a special module initialization function will automatically get called, and within this function, we embed the checksum code that computes the checksum for each of the module's functions. The pre-computed checksums are then stored in memory. When a malicious function is invoked, we can re-calculate the checksum and compare it with the stored value. By doing so, any modification against h4x_unlink() will be detected, and we can trigger a system crash (or some other abnormal behaviors). One might think that this is a denial-of-service attack, but in fact, it is not. The reason is that this attack is only mounted at the testing phase of a kernel driver purifier tool, not at a real production scenario.

Using the approach we presented, the re-generated h4x_unlink() function looks like the following:

```
1 asmlinkage int h4x_unlink(const char __user *pathname) {
2       int r;
3       char *kbuf=(char*)kmalloc(256,GFP_KERNEL);
4       copy_from_user(kbuf,pathname,255);
5       if(strstr(kbuf,_H4X0R_)||strstr(kbuf,KBEAST)){
6               kfree(kbuf);
7               if(compute_and_compare_checksum()==0)
8                       crash_the_kernel();
9               return -EACCES;
10      }
11
12      r=(*o_unlink)(pathname);
13      kfree(kbuf);
14      if(compute_and_compare_checksum()==0)
15              crash_the_kernel();
16      return r;
17 }
```

In the code snippet above, lines 7, 8, 14, and 15 are inserted in the form of pseudo code. The meaning of these lines of code are self-explanatory. One must be careful that on lines 8 and 15 when we intentionally crash the kernel, we

should not invoke any kernel functions such as panic() or die(), because these are also kernel API functions that could be removed by the purifier. In our implementation, instead of calling any kernel API functions, we use a simpler method of crashing the kernel by writing to the global NULL pointer so as to force a null pointer de-reference. One could also crash the kernel by overwriting the stack or performing badly-aligned memory operations [17]. Furthermore, as simply crashing the system when tampering is detected can raise a red flag, one can simulate different ways of code malfunctioning besides crashing the system, as long as it can still fail the tests of the purifier.

3.2 Kernel API Call Less Rootkit

We now present another approach for defeating the kernel driver purifier. As we described before, the kernel driver purifier assumes that rootkits have to invoke kernel APIs to achieve malicious intents. However, this is not necessarily true. To validate, we studied how many kernel APIs are used by real world rootkits and what they are. Our study chooses three rootkits: KBeast, Adore-ng [26], and DR [2]. These rootkits represent three different types of kernel rootkits.

KBeast uses the most straightforward approach. It achieves the malicious goals by hooking to the system call table so that it can redirect the code path to a malicious handler.

Adore-ng is more stealthy than KBeast. Instead of modifying the system call table, it uses the Virtual File System (VFS) intercept method. Especially, it intercepts functions at the VFS layer, which controls interactions to the ordinary file system as well as the /proc file system. By intercepting functions at the VFS layer and filtering malicious data, information can be hidden.

Among these three, DR is the most tricky one and is highly resistant to various detection tools. It is based on the attributes of the Debug Registers. The Debug Registers are special registers provided by IA32 processors used for supporting debugging operations. However, these registers can also be used in a malicious way. The DR rootkit sets a breakpoint at the system call handler, and replaces the do_debug() function, which is supposed to handle the debug operations, with its own function. In doing so, every time a system call is invoked, the control is first passed to the malicious function, in which malicious data are filtered.

A different kernel rootkit uses a different number of kernel API functions. This is because they exploit different parts of the system and support different auxiliary features. For example, there are 27 kernel API calls made by DR. These 27 API calls fall into five different categories, including memory operations, string operations, hijacked functions, debug purposes, and miscellaneous, for each of which, we will handle differently.

Strategy 1: Avoid using kernel API calls if possible. For example, the following piece of code is a part of the hook_getdents64() defined in the DR rootkit.

```
...
   struct dirent64 *our_dirent;
   struct dirent64 *their_dirent;
...
   their_dirent  = (struct dirent64 *) kmalloc(count, GFP_KERNEL);
```

```
    our_dirent    = (struct dirent64 *) kmalloc(count, GFP_KERNEL);

    /* can't read into kernel land due to !access_ok() check in original */
    their_len = original_getdents64(fd, dirp, count);

    if (their_len <= 0)
    {
        kfree(their_dirent);
        kfree(our_dirent);
        return their_len;
    }
...
```

To avoid using the functions *kmalloc* and *kfree*, we can use the struct variables directly, instead of using pointers as shown below.

```
    struct dirent64 our_dirent;
    struct dirent64 their_dirent;

    /* can't read into kernel land due to !access_ok() check in original */
    their_len = original_getdents64(fd, dirp, count);

    if (their_len <= 0)
        return their_len;
```

The major difference between these two code snippets is that, the former employs the kmalloc/kfree pair, which allocates/frees memory dynamically, while the latter allocates/frees memory in a static way, though they should have the same functionality.

Strategy 2: Re-implement kernel functions in the rootkit. Most of string operations can be implemented in a few lines of C code, such as `strcmp` and `strstr`. In fact, since Linux kernel has implemented these functions, we can reuse the code; however, instead of calling these kernel defined functions, we can include these functions as part of the rootkit.

Strategy 3: Do nothing. Some kernel API calls are inherently critical calls, e.g., those original system calls (hijacked by the rootkit). Apparently, the kernel driver purifier would not eliminate these calls, as that would always cause the system to crash. Some calls, such as *printk*, are only used for debugging purposes. Thus, we do not need to do anything here as it does not matter even if they are eliminated by a kernel driver purifier.

4 Implementation

4.1 Self-Checksum Based Rootkit

To implement this attack and evaluate its effectiveness in injecting real world rootkits, we first add the checksum code to ensure that the injected rootkit is not tampered with by a purifier. Any checksum algorithm would suffice (e.g., CRC32, MD5, SHA1). These algorithms are not new, and using one of these algorithms to generate a checksum for a block of C code has been well studied by Viega and Messier [29]. Our implementation is built on their work, and the checksum code in our scenario consists of less than 200 lines of C code.

There are two possible ways to calculate the checksum of a rootkit: either compute one checksum for the whole rootkit, or compute one checksum for each

function. The former is simpler to implement, but it makes the rootkit more detectable. This is because computing one checksum for the whole rootkit means, any changes to the rootkit would lead to checksum mismatch, and therefore might crash the system or trigger some other pre-defined behaviors. Such a property makes the rootkit suspicious, i.e., when a purifier detects that all attempts to remove API invocations fail, it will realize that something is wrong. In contrast, we use the latter approach, namely, for each function, we compute a checksum, and compare all these checksums during runtime with their initial values. To obfuscate the code, some obfuscating functions which also contain a certain number of API invocations can be added. For instance, we can add some functions that do nothing but just call the kernel print API to print some bogus information. The following shows an example, where we include a function called h4x_bogus between two malicious functions: h4x_unlink and h4x_rmdir.

```
asmlinkage int h4x_unlink(const char __user *pathname) {
...
}

asmlinkage int h4x_bogus (){
       printk("some bogus information\"n);
}

asmlinkage int h4x_rmdir(const char __user *pathname) {
...
}
```

In this example, we do compute the checksums for h4x_unlink and h4x_rmdir, respectively, but we do not compute the checksum for h4x_bogus.

In addition, we compute the initial checksum during the module initialization stage. One might think that the computation code in the module initialization area could also be removed by the kernel driver purifier. This can be easily coped with; if for some reason, the checksum has not been initialized properly, we can assume the computation code has been removed and we will simply crash the kernel. An alternative approach is, instead of computing the checksum during the module initialization stage, we can compute it offline and store it on the disk, and then read it into memory when the module is loaded. If this read operation fails, we will also trigger a kernel crash.

To convert existing rootkits such that they are protected by the checksum code and are resistant to kernel driver purifiers, we develop a compiler-like tool to perform the insertion of the checksum code described above automatically. Figure 2 illustrates the basic flowchart of this automation tool. The tool is implemented as a perl script, and it includes less than 250 lines of Perl code. It takes the original rootkit as input, inserts the checksum logic into the rootkit, and outputs the modified rootkit. Basically, it first parses the source file to get the list of all the malicious functions defined in the rootkit. Next, for each malicious function, it computes its checksum. Such a computation logic is included in the module initialization code. Finally, the same computation logic, as well as a comparison logic, is inserted into each function. After these steps, the modified rootkit is generated.

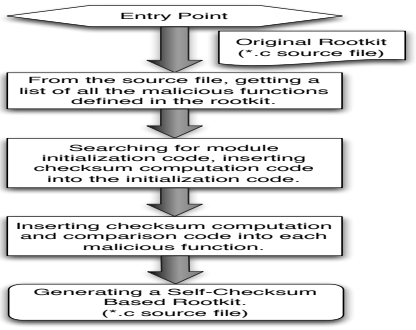

Fig. 2: Flowchart of the automation tool

4.2 Kernel API Call Less Rootkit

Using the strategies described in Section 3, we have implemented a kernel-API-call less rootkit. For the sake of comparison, we develop our rootkit based on the DR rootkit. This rootkit consists of 1298 lines of C code (including comments), and our rootkit consists of 1381 lines of C code. Our rootkit does not call the printk statements and any kernel functions except the original system calls. A kernel driver purifier that removes the original system calls would certainly crash the system, thus, it would mark them as critical and not remove them. As for the printk statements, it is irrelevant whether or not they are removed as they do not impact the the essential functions of the rootkit.

5 Experimental Evaluation

We use several real world rootkits in our evaluation. To demonstrate the effectiveness of our proposed method, especially to verify that the attack maintains the rootkit's functionalities, we use KBeast. To measure the performance overhead and the automation of our attack, we choose KBeast, Adore-ng, and DR as our rootkits. The experiments are carried out in four steps. First, we evaluate the effectiveness of our rootkit injection mechanism. For the checksum based rootkits, we first use our compiler tool to automatically inject the checksum code into a kernel driver infected by the rootkits. By simulating kernel driver purification, we can verify if the modified rootkits (i.e., rootkits with checksum logic) can evade detection and elimination. To verify the effectiveness of kernel-API-call less rootkit, we manually examine its source code and the generated binary, ensure that it does not contain any calls to kernel API functions (except the original system calls and debug statements).

Second, to confirm our rootkits still retain all the malicious functionalities, we manually perform test cases to validate. Third, we evaluate the performance impact of our rootkits. For the checksum based rootkit, we evaluate the kernel driver in various formats: the vanilla uninfected driver, the infected driver without the checksum code, and the infected driver with the checksum code. As a large overhead caused by performing checksum could easily lead to its discovery, we tune our checksum code so this does not happen. Finally, we validate the effectiveness of our automation tool, which can automatically transform exist-

ing rootkits into a form that is resistant to kernel driver purifiers. We run our experiments on a Linux Qemu-KVM based virtual machine. Our test machine is a Dell Desktop (with Intel Xeon 3.07GHz Quad-Core CPU and 4GB memory) running OpenSuSE 12.3. We used OpenSuSE 11.3 as the guest OS.

5.1 Effectiveness

We choose E1000 NIC driver as our target driver and inject the KBeast rootkit into it. The injection method we use is described in Phrack issue 68 [27]. The basic idea is, by redirecting the initialization function pointer to the malicious init function rather than the original init function, the malicious init function will be invoked when the module is loaded into the Linux kernel; therefore, the malicious functions will be registered into the kernel and the original system calls will be hijacked.

As mentioned before, the kernel driver purifier consists of three phases: profiling phase, testing phase, and rewriting phase. The focus of our attack is on the second phase, i.e., the testing phase. In the testing phase, the kernel driver purifier attempts to eliminate all kernel API invocations that do not affect the correct execution of the test suite.

To verify that our checksum based rootkit is indeed immune to such purifiers, we conduct our experiments on a QEMU+gdb platform. By using gdb, we can figure out the address of each function. Then we can disassemble each function, and thus figure out the address of each kernel API invocation. We replace these kernel API invocations with NOP instructions. We want to verify that when we replace the kernel API invocation in a malicious logic, the system would crash; however, when we replace the kernel API invocation in a benign logic with NOP instructions, the system may not crash, especially, if it is a printk statement or some other non-critical kernel API invocations.[4] By doing so, we can prove that our rootkit is resistant to kernel driver purifiers, because kernel driver purifiers would attempt to remove every kernel API invocation, but if the removal of certain invocations causes system reboot or crash, such invocations are marked as *critical* and would be retained. Since we use the checksum based code, any removal of the malicious invocation would cause system to crash; therefore, all of the malicious kernel API invocations will be kept.

Case Study: The following is the disassembled code of the malicious function h4x_unlink in KBeast (with checksum code embedded). Compared with the C code of h4x_unlink(), we know that the following kernel APIs are invoked: kmem_cache_alloc_notrace, slab_buffer_size, _copy_from_user, strstr, and kfree. To verify that our checksum code makes the malicious logic resistant to kernel driver

[4] In fact, the replacement of a kernel API invocation is more complicated than one would expect. When the kernel API invocation has a return value and it is used later, replacing such an invocation with a series of NOPs would cause an undefined situation. However, since our goal is to detect code changes, we do not have to resolve this issue. By adding some debugging statements, when system crashes, we can tell whether it is caused by the checksum mis-match or by the "undefined situation" problem.

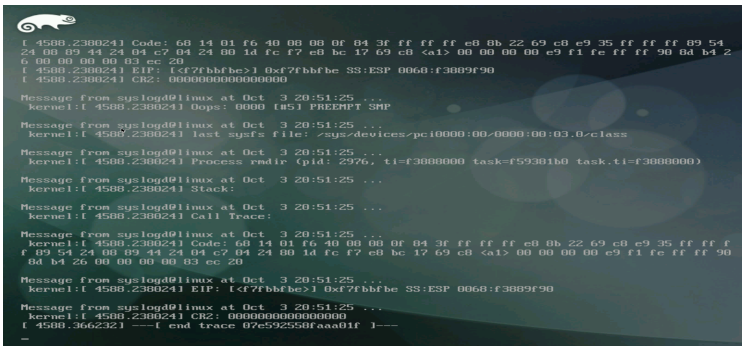

Fig. 3: System crashes when we replace the call of strstr with NOP instructions

purifiers, we attempt to replace any of these kernel API invocation instructions with NOP instructions. And we expect that such a replacement would cause a system crash.

The disassembled code of the function h4x_unlink includes nearly 90 lines of x86 assembly instructions. To save space, we omit most of the instructions, but we do show those that call the kernel API functions _copy_from_user and strstr.

```
1 gdb> disassemble h4x_unlink
2 Dump of assembler code for function h4x_unlink:
...
30 0xf7fbc039 <+105>: call 0xc03fa250 <_copy_from_user>
31 0xf7fbc03e <+110>: mov $0xf7fc2174,%edx
32 0xf7fbc043 <+115>: mov %ebx,%eax
33 0xf7fbc045 <+117>: call 0xc03f9b70 <strstr>
34 0xf7fbc04a <+122>: test %eax,%eax
35 0xf7fbc04c <+124>: jne 0xf7fbc088 <h4x_unlink+184>
36 0xf7fbc04e <+126>: mov $0xf7fc216d,%edx
37 0xf7fbc053 <+131>: mov %ebx,%eax
......
......
90 End of assembler dump.
```

As an example, we show the disassembled code after we have replaced the strstr function call (i.e., line 33 of the above disassembled code), with five NOP instructions. Since the original call instruction has five bytes, and each NOP instruction occupies one byte, to ensure the alignment, we use five NOP instructions to fill up one call instruction.

```
1 gdb> disassemble h4x_unlink
2 Dump of assembler code for function h4x_unlink:
...
30 0xf7fbc039 <+105>: call 0xc03fa250 <_copy_from_user>
31 0xf7fbc03e <+110>: mov $0xf7fc2174,%edx
32 0xf7fbc043 <+115>: mov %ebx,%eax
33 0xf7fbc045 <+117>: nop
34 0xf7fbc046 <+118>: nop
35 0xf7fbc047 <+119>: nop
36 0xf7fbc048 <+120>: nop
37 0xf7fbc049 <+121>: nop
38 0xf7fbc04a <+122>: add %al,(%eax)
39 0xf7fbc04c <+124>: add %bh,(%edx)
40 0xf7fbc04e <+126>: mov $0xf7fc216d,%edx
```

```
41 0xf7fbc053 <+131>: mov %ebx,%eax
......
......
94 End of assembler dump.
```

Figure 3 illustrates that the system indeed crashes when we replace the call to strstr() with NOP instructions.

Similarly, we also verify that replacing the other kernel APIs invocations, kmem_cache_alloc_notrace, slab_buffer_size, _copy_from_user, or kfree, with a series of NOP instructions, has the same effect. In other words, they all crash the system. Therefore, such a checksum based code is capable of protecting kernel API calls from being eliminated by kernel driver purifiers.

5.2 Functionality

Next, we verify that all the malicious functions are still retained. According to its user guide, KBeast has the following features:

- Hiding this loadable kernel module.
- Hiding processes/files/directories/sockets/connections.
- Keystroke logging to capture user activity.
- Remote binding backdoor hidden by the kernel rootkit.
- Anti-kill process.
- Anti-remove files.
- Anti-delete this loadable kernel module.
- Local root escalation backdoor.

We manually verify that all these functionalities are preserved after we add the checksum code into the rootkit and inject such a rootkit into the E1000 NIC driver. Similarly, we have also verified that our kernel-API-call less rootkit have all the features enabled as the DR rootkit.

5.3 Performance Overhead

We first evaluate the CPU overhead of the checksum code. We use the CPU intensive benchmark Cuadro [8] to measure the CPU performance overhead. Cuadro is a benchmarking program that measures CPU performance by numerically seeking a solution and measuring the runtime of a two-dimensional heat equation in Cartesian coordinates. We compare the CPU performance for the original E1000 NIC driver, the E1000 NIC driver infected by the original rootkit, and the E1000 NIC driver infected by our self-checksum based rootkit. The results are presented in Figure 4. For each group, the blue bar (leftmost) denotes the original driver, the red bar (middle) depicts the trojaned driver without the checksum code, and the green bar (rightmost) represents the trojaned driver with the checksum code. From the results, we observe that the checksum code induces negligible computation overhead to the trojaned driver: the checksum code only incurs 1% to 2% additional overhead. One might wonder why we use a CPU intensive benchmark to measure NIC driver. The reason is that the rootkit code, which is injected into the NIC driver, includes various intercepted system

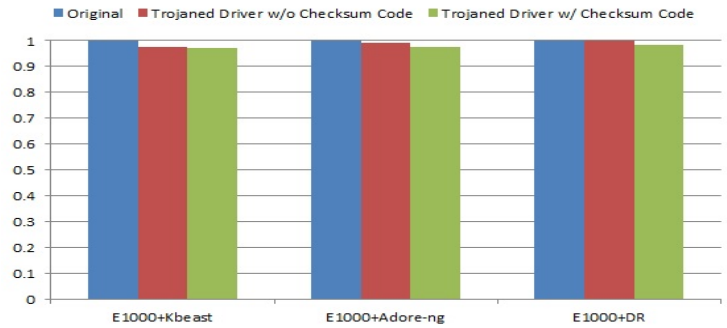

Fig. 4: Comparison of normalized CPU performance among original driver, driver infected by rootkit without the checksum code, and driver infected by rootkit with the checksum code

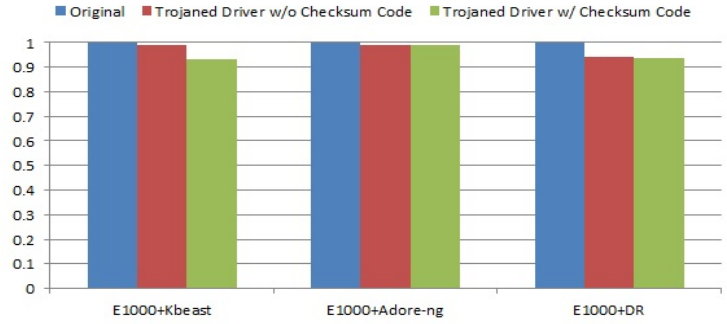

Fig. 5: Comparison of normalized network throughput among original driver, driver infected by rootkit without the checksum code, and driver infected by rootkit with the checksum code

call code, and system calls are frequently used by various workloads; therefore, such interception would incur CPU overhead. That is why, even without the checksum code (the red bar), the infected driver performs worse than the vanilla driver when running CPU intensive workload.

We also evaluate the network performance overhead of the checksum code. We use Iperf [13] to measure the network throughput of the E1000 NIC driver infected by various rootkits. Iperf is a commonly used network benchmark tool and can create TCP and UDP data packets to measure the throughput between two endpoints. Similarly, we compare the network throughput for the original E1000 NIC driver, the E1000 NIC driver infected by the original rootkit, and the E1000 NIC driver infected by our self-checksum based rootkit. The results are presented in Figure 5. The blue bar (leftmost) denotes the original driver, the red bar (middle) depicts the trojaned driver without the checksum code, and the green bar (rightmost) represents the checksum based trojaned driver. From the results, we observe that the checksum code induces negligible network overhead to the trojaned driver: the checksum code only incurs about 1% to 5% additional overhead.

Finally, we evaluate the CPU overhead of the kernel-API-call less rootkit. Since this rootkit is built on the DR rootkit, we compare its CPU overhead with that of the DR rootkit. We still use Cuardo as the CPU intensive benchmark. Since we do not add any extra logic to the DR rootkit, we do not expect our kernel-API-call less rootkit to cause any noticeable performance overhead, and indeed, our experimental results show that our rootkit incurs less than 1% of CPU overhead compared to the DR rootkit.

5.4 Automation

We choose KBeast, Adore-ng, and DR as our target rootkits, as they represent three different types of kernel rootkit implementations.

Although the above rootkits make use of different techniques to accomplish their malicious purposes, we have verified that, our compiler-like tool, written in Perl script, can automatically convert all these rootkits to a format that is resistant to kernel driver purifiers.

6 Defense Mechanisms

In this section, we first compare the two proposed rootkit injection methods, and then we discuss the defense mechanisms against them.

6.1 Comparison between Our Two Methods

The checksum based rootkit has the advantage that it does not require an attacker to be familiar with a specific rootkit and the Linux kernel. The attacker could simply download a rootkit from the Internet, and uses our compiler tool to inject the rootkit along with the checksum code so it can circumvent kernel driver purifiers. In contrast, kernel-API-call less rootkit would require an attacker to be familiar with both the rootkit and various aspects of the kernel. The more complex a rootkit is, the more effort it would require to inject such a rootkit.

The kernel-API-call less rootkits can completely and cleanly bypass a purifier's detection logic, and thus, they are more resistant to being detected. Moreover, as it does not introduce extra checksum logic, it incurs even less performance overhead, which could further lower its chance of being detected by tools that monitor performance anomalies.

6.2 Defense Mechanism

There are several ways to defend against the checksum based attack we have proposed. A purifier can potentially add a NOP operation before each kernel API call. If the kernel driver fails the test or crashes the kernel, it is an indication that a piece of the driver code is performing suspicious activities. The kernel driver purifier could use this approach to detect such an attack, which is shown in Figure 6. A limitation of this approach is that it can detect whether or not the checksum logic is present in the kernel driver, but its presence does not equate to the presence of malicious code. The checksum code could very well be a security mechanism to prevent malicious code from tampering with legitimate code. This implies that system administrators would need to manually verify the legitimacy of the kernel driver code by other means, e.g., check the MD5 value

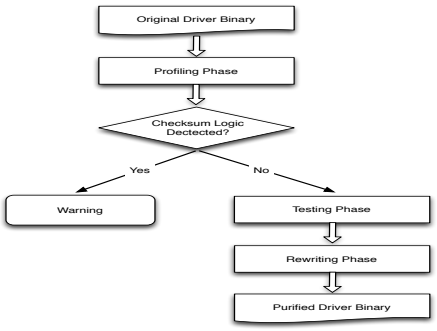

Fig. 6: Kernel Driver Purifier with Checksum Detection

of the binary driver against the published value released by an authoritative source. An alternative defense is to do fine-grained testing and filtering in the purifier. Current kernel driver purifiers only deal with kernel API calls, but one could potentially perform the same trial-and-error steps at the binary instruction level. With a sufficient amount of time, the checksum code embedded into the kernel driver could all be removed, thus leaving only the original driver code along with any injected malicious code. From there, the malicious code can be detected using the same approach as the existing kernel driver purifier employs. The main challenge of this approach is time complexity, as generating all the combinations of code at a granularity of the binary instruction level to test would take an astronomical amount of time. One heuristic to make this approach more practical is to perform trial-and-error steps at a granularity of basic blocks. This could improve time complexity by at least several orders of magnitude. We have not yet started working on the defense mechanisms and will consider this as our future work.

While the above approaches might be effective in defending against the checksum based rootkit, they are certainly ineffective against our kernel-API-call less rootkit. The kernel-API-call less rootkit actually subverts the fundamental assumption of the kernel driver purifier, and therefore we believe, any change in the kernel driver purifier would just be in vain. To detect the kernel-API-call less rootkit, defenders have to resort to some other approaches, for example, the virtualization introspection based approaches.

7 Related Work

7.1 Kernel Protection from Buggy or Malicious Drivers

We first briefly summarize previous work related to kernel protection from buggy or malicious drivers. Most existing research efforts can be divided into three categories — isolation based, hypervisor based, and offline testing based.

The key idea of isolation based systems is to isolate the presumably malicious or buggy drivers from the core part of the kernel. Nexus [30] and SUD [5] both propose to move device drivers into user space. By introducing the notion of API integrity and module principals, and using a compiler plugin, LXFI [21] can

isolate kernel drivers from the core kernel. Hypervisor based approaches such as [31] and [25] require operating systems to run on top of a hypervisor, from which, malicious behaviors can be monitored, tracked, and constrained.

While the above two approaches work in a runtime environment, some other research projects attempt to handle buggy drivers offline. SDV [4] presents a static analysis engine. By analyzing the source code, SDV locates kernel API usage errors in drivers. In contrast, DDT [20] utilizes a symbolic execution technique to pinpoint bugs in a closed source binary device driver. Although these offline testing based systems do not incur any runtime overhead, they target bugs in drivers, instead of the malicious logic.

Note that the kernel driver purifier we target is orthogonal and complementary to all these previous projects mentioned above. The kernel driver purifier is also an offline based method and does not incur any runtime overhead. In comparison to existing offline systems, the kernel driver purifier aims to eliminate the malicious logic in the drivers.

7.2 Self-checksumming

Self-checksumming has been proposed as a technique for verifying or ensuring software integrity. One of the earliest such proposals was made by Aucsmith [3] in 1996, and he used self-checksumming as a key part of building tamper resistance software. Based on his work, other researchers have then proposed some alternative approaches [6, 12], which mainly focus on improving the performance and making it easy to be included into existing software. Since then, self-checksumming has become a popular tamper-resistance strategy, and it has mostly been used in a defensive manner. For example, in the Pioneer system [24], by using self-checksumming code, one can verify code integrity and enforce untampered code execution; and based on this technique, the authors successfully built a kernel rootkit detector. As another example, a self-checksumming algorithm is proposed to build a timing-based attestation system for enterprise use [19]. While these systems make use of self-checksumming to protect benign code, in this paper, we demonstrate that such a technique can also be used for malicious purposes.

8 Conclusion

In this paper, we have presented two offensive approaches to defeating kernel driver purifiers. In the first approach, by incorporating a self-checksumming algorithm into an existing rootkit and injecting the rootkit into a kernel driver, we can successfully evade the detection of a kernel driver purifier. In the second approach, the proposed kernel-API-call less rootkit avoids using kernel APIs and hence subverts the fundamental assumption of a kernel driver purifier, making the purifier completely ineffective. We have implemented the prototypes of the proposed rootkits. Through real experiments, we have shown that both rootkits preserve all the malicious functionalities as the original, with negligible performance overhead. Finally, to highlight the advantage of the checksum based rootkit technique, we have developed a compiler-like tool that automatically

transforms the existing rootkits into their variants that are resistant to kernel driver purifiers.

We conclude that current kernel driver purification technique, though promising and effective in handling existing rootkits, is still too fragile to cope with more sophisticated rootkits. In our future work, we will develop more advanced kernel driver purifiers to defend against those self-checksum based rootkits. But we believe that more fundamental changes are needed in the kernel driver purification technique to defend against the kernel-API-call less rootkits.

References

1. Sony bmg copy protection rootkit scandal. `http://en.wikipedia.org/wiki/Sony_BMG_copy_protection_rootkit_scandal`.
2. B. Alberts. Dr linux 2.6 rootkit released. `http://lwn.net/Articles/296952/`.
3. D. Aucsmith. Tamper resistant software: An implementation. In *Proceedings of the first International Workshop on Information Hiding*, pages 317–333. Springer, 1996.
4. T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *Proceedings of the first European Conference on Computer Systems (EuroSys)*, volume 40, pages 73–85. ACM, 2006.
5. S. Boyd-Wickizer and N. Zeldovich. Tolerating malicious device drivers in linux. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 9–9. USENIX Association, 2010.
6. H. Chang and M. J. Atallah. Protecting software code by guards. In *In Proceedings of the first ACM Workshop on Digital Rights Management (DRM)*, pages 160–175. Springer, 2002.
7. A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *Proceedings of the eighteenth ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 2001.
8. Cuadro cpu benchmark. `http://sourceforge.net/projects/cuadrocpubenchm`.
9. T. Garfinkel, M. Rosenblum, et al. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the tenth Annual Symposium on Network and Distributed Systems Security (NDSS)*, 2003.
10. K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (very) large: ten years of implementation and experience. In *Proceedings of the twenty-second ACM Symposium on Operating Systems Principles (SOSP)*, pages 103–116. ACM, 2009.
11. Z. Gu, W. N. Sumner, Z. Deng, X. Zhang, and D. Xu. Drip: A framework for purifying trojaned kernel drivers. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2013.
12. B. Horne, L. Matheson, C. Sheehan, and R. E. Tarjan. Dynamic self-checking techniques for improved tamper resistance. In *In Proceedings of the first ACM Workshop on Digital Rights Management (DRM)*, pages 141–159. Springer, 2002.
13. Iperf benchmark. `http://sourceforge.net/projects/iperf/`.
14. IPSECS. The kbeast rootkit. `http://core.ipsecs.com/rootkit/kernel-rootkit/kbeast-v1/`.
15. X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction. In *Proceedings of the fourteenth ACM Conference on Computer and Communications Security (CCS)*, pages 128–138. ACM, 2007.

16. A. Kadav and M. M. Swift. Understanding modern device drivers. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, volume 40, pages 87–98. ACM, 2012.
17. S. Kagstrom. Provide ways of crashing the kernel through debugfs. `http://lwn.net/Articles/371208/`.
18. G. Keizer. Researchers spot rootkits on more sony usb drives. `http://www.computerworld.com/s/article/9033798/Researchers_spot_rootkits_on_more_Sony_USB_drives`.
19. X. Kovah, C. Kallenberg, C. Weathers, A. Herzog, M. Albin, and J. Butterworth. New results for timing-based attestation. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 239–253. IEEE, 2012.
20. V. Kuznetsov, V. Chipounov, and G. Candea. Testing closed-source binary device drivers with ddt. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 12–12. USENIX Association, 2010.
21. Y. Mao, H. Chen, D. Zhou, X. Wang, N. Zeldovich, and M. F. Kaashoek. Software fault isolation with api integrity and multi-principal modules. In *Proceedings of the twenty-third ACM Symposium on Operating Systems Principles (SOSP)*, pages 115–128. ACM, 2011.
22. D. Mitchell. The rootkit of all evil. `http://www.nytimes.com/2005/11/19/business/media/19online.html?_r=0`.
23. N. L. Petroni Jr, T. Fraser, J. Molina, and W. A. Arbaugh. Copilot-a coprocessor-based kernel runtime integrity monitor. In *USENIX Security Symposium*, pages 179–194, 2004.
24. A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In *Proceedings of the twentieth ACM Symposium on Operating Systems Principles (SOSP)*, volume 39, pages 1–16. ACM, 2005.
25. A. Srivastava and J. T. Giffin. Efficient monitoring of untrusted kernel-mode execution. In *Proceedings of the eighteenth Annual Symposium on Network and Distributed System Security (NDSS)*. Citeseer, 2011.
26. stealth. Announcing full functional adore-ng rootkit for 2.6 kernel. `http://lwn.net/Articles/75991/`.
27. styx^. Infecting loadable kernel modules: kernel versions 2.6.x/3.0.x. `http://www.phrack.org/issues.html?issue=68&id=11#article`.
28. W.-K. Sze and R. Sekar. A portable user-level approach for system-wide integrity protection. In *Proceedings of the 29th Annual Computer Security Applications Conference*, pages 219–228. ACM, 2013.
29. J. Viega and M. Messier. *Secure Programming Cookbook for C and C++: Recipes for Cryptography, Authentication, Input Validation & More*. O'Reilly Media, Inc., 2009.
30. D. Williams, P. Reynolds, K. Walsh, E. G. Sirer, and F. B. Schneider. Device driver safety through a reference validation mechanism. In *Proceedings of the 8th USENIX conference on Operating Systems Design and Implementation (OSDI)*, pages 241–254, 2008.
31. X. Xiong, D. Tian, and P. Liu. Practical protection of kernel integrity for commodity os from untrusted extensions. In *Proceedings of the eighteenth Annual Symposium on Network and Distributed System Security (NDSS)*, 2011.
32. F. Zhang, K. Leach, K. Sun, and A. Stavrou. Spectre: A dependable introspection framework via system management mode. In *Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12. IEEE, 2013.