Florida International University Technical Report TR-2012-SEC-03-01

KQguard: Protecting Kernel Callback Queues

Jinpeng Wei[1], Feng Zhu[1], Calton Pu[2]

[1]School of Computing and Information Sciences
Florida International University
Miami, FL 33199

{weijp, fzhu001}@cs.fiu.edu

[2]College of Computing
Georgia Institute of Technology
Atlanta, GA 30332

calton@cc.gatech.edu

KQguard: Protecting Kernel Callback Queues

Jinpeng Wei, Feng Zhu, Calton Pu

ABSTRACT

Kernel callback queues (KQs) are the mechanism of choice for handling events in modern kernels. KQs have been misused by real-world malware to get a kernel thread to run malicious code. Current defense mechanisms for kernel code and data integrity have difficulties with KQ attacks, since they work without necessarily changing legitimate kernel code or data. In this paper, we describe the design, implementation, and evaluation of KQguard, an efficient and effective protection mechanism of KQs. KQguard uses static and dynamic analysis of kernel and device drivers to learn the legitimate event handlers. At runtime, KQguard rejects all the unknown KQ requests that cannot be validated. We implemented KQguard on Windows Research Kernel (WRK) and extensive experimental evaluation shows KQguard is efficient (up to 5% overhead) and effective (capable of achieving zero false positives and false negatives against 11 real malware and 9 synthetic attacks). KQguard protects all the 20 KQs in WRK, can be extended to accommodate new device drivers, and through dynamic analysis can support closed source device drivers.

1. INTRODUCTION

One of the most time-critical functions of an operating system (OS) kernel is interrupt/event handling, e.g., timer interrupts. In support of asynchronous event handling, multi-threads kernels store the information necessary for handling an event as an element in a kernel queue (called KQ for short), specialized for that event type. To avoid interpretation overhead, each element of a KQ contains a callback function pointer to an event handler specialized for that specific event, plus its associated execution context and input parameters. When an event happens, a kernel thread invokes the specified callback function as a subroutine to handle the event.

KQs are the mechanism of choice for handling events in modern kernels. As concrete examples, we found 20 KQs in the Windows Research Kernel (WRK) and 22 in Linux. In addition to being popular with kernel programmers, KQs also have become a very useful tool for kernel-level rootkits (Section 5.2 and [5][21]), malicious programs that remain hidden in the kernel to execute kernel tasks ordered by an attacker. For example, the Push-do/Cutwail spam bot has misused the Registry Operation Notification Queue of the Windows kernel to monitor, block, or modify legitimate registry operations [8]. This paper includes 11 examples of real world rootkits misusing KQs demonstrating these serious current exploits, and 9 additional synthetic potential misuses for illustration of future dangers.

To counter these threats, one potential solution is to consider the KQs as bugs as typical exploits of kernel bugs today, and try to fix them. Since the rootkits are only exploiting legitimate APIs of KQs, the only way to fix this bug is to remove KQs themselves. This drastic solution would require significant and extensive kernel redesign and it is currently impractical. Instead, we consider KQs as important kernel features that we would like to keep. Our defense, called KQguard, performs validation checks that will ensure the execution of legitimate event handlers only, by filtering out all the callbacks of unknown origin.

KQguard is carefully designed and implemented to satisfy four stringent requirements of kernel facilities. The first two are functional software requirements, while the last two are practical adoption requirements. The first requirement is efficiency: KQguard should minimize the overhead of callback function validation. The second is effectiveness: KQguard should distinguish attack KQ requests from legitimate event handlers with zero errors. The third is coverage: KQguard should work for all the KQs in the kernel. The fourth is extensibility: KQguard should support future legitimate event handlers such as device drivers. The fifth is inclusiveness: KQguard should work without source code in order to support third-party, closed source device drivers. A survey of potential solutions for addressing the KQ exploits is included in Section 6, where we explain our findings that current solutions have difficulties with one or more of the four requirements.

The main contribution of this paper is the design, implementation, and evaluation of KQguard. By combining several techniques (e.g., static analysis of kernel source code and dynamic analysis of device driver binaries), KQguard is able to satisfy the design requirements of efficiency, effectiveness, coverage, extensibility, and inclusiveness. We have implemented the KQguard in WRK [33] and started work on the Linux kernel. Extensive evaluation of KQguard on WRK shows its effectiveness against KQ exploits (both real and synthetic), detecting all the attacks (zero false negatives). With appropriate training, we eliminated all false alarms from KQguard (zero false positives) for representative workloads. For resource intensive benchmarks, KQguard carries a small performance overhead of up to 5%. The ongoing implementation effort on Linux confirms our experience on WRK. KQguard protects all 20 KQs in WRK; it can be extended to accommodate new device drivers, and it supports closed source device drivers through dynamic analysis.

The rest of the paper is organized as follows. Section 2 summarizes the problem caused by rootkits misusing KQs. Section 3 describes the high level design of KQguard defense by abstracting the KQ facility. Section 4 outlines some implementation details of KQguard for WRK and Linux, validating the design. Section 5 presents the results of an experimental evaluation, demonstrating the effectiveness and efficiency of KQguard. Section 6 outlines related work and Section 7 concludes the paper.

2. PROBLEM ANALYSIS: KQ Hijack

2.1 Importance of KQ Hijack Attacks

Functionally, KQs are kernel queues that support the callback of a programmer-defined event handler, specialized for efficient handling of that particular event. For example, the soft timer queue of the Linux kernel supports scheduling of timed event-handling functions. The original kernel thread requesting the timed event specifies an event time and a callback function to be executed at the specified time. When the system timer reaches the specified time, the kernel timer interrupt handler invokes the callback function stored in the soft timer request queue (Figure 1). More generally and regardless of the specific event semantics among the KQs, their control flow conforms to the same abstract type: For

each request in the queue, a kernel thread invokes the callback function specified in the KQ request to handle the event.

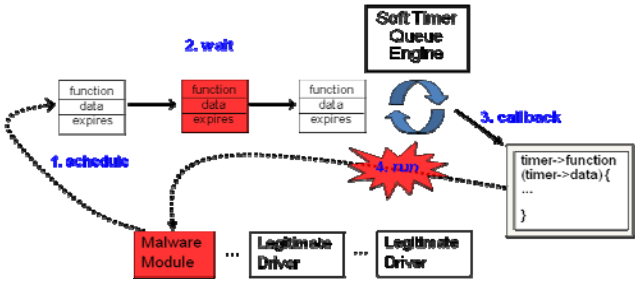

Figure 1: Life cycle of a soft timer request in Linux

Kernel-level rootkits exploit the KQ callback control flow to execute malicious code by inserting their own request into a KQ (e.g., by supplying malicious callback function or data in step 1 of Figure 1). This kind of manipulation, called a KQ hijack attack, only uses legitimate kernel API and it does not change legitimate kernel code or statically allocated data structures such as global variables. Therefore, syntactically a KQ hijack request is indistinguishable from normal event handlers. Consider the Registry Operation Notification Queue as illustration. Using it in defense, anti-virus software event handlers can detect potential intruder malicious activity on the registry. Using it in KQ hijack attack, Pushdo/Cutwail [8] can monitor, block, or modify legitimate registry operations.

Several KQ Hijack Attacks by real world malware have been documented. First, rootkits have misused KQs to hide better against discovery. For example, the Rustock.C spam bot relies on two Windows kernel timers [17] to check whether it is being debugged/traced [2][15] (e.g., whether KdDebuggerEnabled is true). Second, rootkits have misused normal KQ functionality for covert rootkit operations. For example, Pushdo/Cutwail botnet sets up three callback routines by invoking `IoRegisterFsRegistrationChange`, `CmRegisterCallback`, and `PsSetCreateProcessNotifyRoutine`, respectively [8]; these callback routines enable it to monitor file system registrations, monitor, block, or modify a registry operation, and inject a malicious module into a services.exe process. Rustock.C also invokes `PsSetCreateProcessNotifyRoutine` [7][23] to inject code into seemingly benign processes (e.g., sevices.exe) so that spamming can be done in the context of an innocent process. Third, rootkits have misused KQ functionality to attack security products directly. For example, the Storm/Peacomm spam bot invokes `PsSetLoadImageNotifyRoutine` to register a malicious callback function that disables security products when they are loaded [4]. Table 3 shows the KQ hijack attacks against WRK that we have studied. It is a representative sample, covering some of the most notorious malware today: the TDSS botnet consists of 4.5 million infected machines and is considered the most sophisticated threat [11], and Duqu [12] is believed to be closely related to the widespread Stuxnet worm [29]. Undoubtedly, KQ Hijack Attacks represent a clear and present danger to current OS kernels.

2.2 KQ Hijack Attack Model

The KQ hijack malware listed in Table 3 misuse KQs in a straightforward way. They prepare a malicious function in kernel space and use its address as the callback function pointer in a KQ request. We call these *callback-into-malware* attacks. Since their malicious functions must be injected somewhere in kernel space, callback-into-malware attacks can be detected by runtime kernel code integrity checkers such as SecVisor [25] or NICKLE [24]. Callback-into-malware is considered the basic level of attack, which may be countered by current defense techniques.

Unfortunately, sophisticated attack technology already developed is able to bypass the detection of standalone malware, e.g., "return-into-libc" attacks [27][30] that use snippets of legitimate kernel code to achieve their malicious intent. The second level of KQ hijack attacks, called *callback-into-libc* (in analogy to return-into-libc), create a malicious callback request containing a legitimate event handler, but invoked with malicious input parameters. When activated, the event handler may carry out unintended actions on behalf of the attacker.

Callback-into-libc KQ hijack attacks represent a new and interesting challenge, since they allow an attacker to execute malicious logic without injecting his own code. To demonstrate the feasibility of callback-into-libc attacks, we developed a prototype KQ hijack program that only uses legitimate kernel code to bypass SELinux. In recent Linux kernels [26], SELinux has been implemented as a Linux Security Module (LSM). SELinux registers its security auditing functions through the `register_security` facility, which fills a global variable called `security_ops` with SELinux functions. Since there is a legitimate function `reset_security_ops` that overwrites `security_ops` to the initial value that points to another global data structure called `default_security_ops` that does no security auditing, our prototype KQ hijack program inserts a soft timer request with `reset_security_ops` as the callback function. When this soft timer expires, `reset_security_ops` is invoked and SELinux is turned off.

The design of KQguard in the next section shows how we can detect both the basic level (callback-into-malware) and the second level (callback-into-libc) KQ hijack attacks.

2.3 Design Requirements in KQ Defense

An effective KQ defense should satisfy five requirements, i.e., efficiency, effectiveness, coverage, extensibility, and inclusiveness. In this section, we outline the reasons some previous techniques may solve specific problems, but have difficulties with satisfying all five requirements. At the same time, we outline the reasons KQguard satisfies these requirements. A more detailed discussion of related work is in Section 6.

Efficiency. Although some KQs have higher efficiency requirements than others (e.g., timer interrupts vs. registry operations), it is important for KQ defenses to minimize their overhead. KQguard is designed to protect all KQs (the coverage requirement) with low overhead, including the time-sensitive ones.

Effectiveness. KQ defenses should detect all the KQ Hijack Attacks (zero false negatives) and make no mistakes regarding the legitimate event handlers (zero false positives). KQguard is designed to achieve this level of precision and recall by focusing on the recognition of all legitimate event handlers.

Coverage. It is important for a KQ defense to protect all KQs, not just the ones under past attacks. Although we do not rule out specialized solutions to protect individual KQs *a priori*, KQguard is designed to protect all KQs that activate their requests by executing a callback function.

Extensibility. Due to the rapid proliferation of new devices, it is important for kernel defenses to extend their coverage to protect the new drivers. The KQguard design isolates the knowledge on legitimate event handlers into a table (the EH-Signature Collection), which is easily extensible.

Inclusiveness. A practical concern of commercial kernels is the protection of third-party, closed source device drivers. KQguard uses static analysis when source code is available, and dynamic analysis to protect the closed source legitimate drivers.

3. DESIGN OF KQGUARD

In this section, we describe the design of KQguard as a general protection mechanism for the KQ abstract type. The concrete implementations (WRK and Linux) are described in Section 4. In the rest of the paper, we will refine our terminology slightly. We will use the term "event handler" to denote legitimate KQ callback functions when the context is clear.

3.1 Assumptions and Architecture

3.1.1 Security Assumptions

We make the following assumptions about the underlying system:

(1) The core kernel and all legitimate device drivers maintain integrity in both their code and data against tampering by malware. This is a realistic assumption due to existing solutions that can protect/check the data integrity of legitimate programs [3][6][14][16].

(2) KQguard code and data (EH-Signature Collection) are similarly protected from tampering by any rootkits installed in the kernel.

(3) KQguard design assumes that detailed information about loaded modules (e.g., name and address range) is maintained by the kernel and accessible by KQguard.

Kernel source code availability is a more complex issue. KQguard relies on static analysis of source code to find the KQs and the event handlers associated with them. However, to satisfy the inclusiveness requirement KQguard should provide an alternative solution for protecting device drivers for which source code is unavailable. Thus we have an additional assumption:

(4) A representative and comprehensive workload is available for a training process to find all the legitimate KQ event handlers (Section 3.3.4). During the training process, we further assume the system is free of rootkits.

A limitation of current KQguard implementation (Section 4) is that the training process is run only once. Thus the runtime dynamic loading of legitimate device drivers is supported only partially. Namely, new drivers that use known KQ event handlers can be dynamically loaded with full protection. Acceptance of new drivers that introduce new KQ callback functions (not seen during training) is postponed to future work.

3.1.2 KQguard Architecture Overview

The main function of KQ defense is to decide at runtime whether a callback function in a KQ request is a legitimate event handler or a malicious KQ hijack attack. KQguard achieves this goal by collecting all of the known event handlers into a table (called EH-Signature Collection) and checking the callback function in question against the EH-Signature Collection.

Although the checking decision is superficially similar to that of anti-virus malware scanners, our approach is fundamentally different from, and complementary to, malware scanners. On the one hand, the construction of signature-based malware scanners is typically reactive: samples of malware are needed in order to design their signatures. On the other hand, KQguard is proactive by identifying the set of legitimate event handlers as a starting point. Instead of assuming the unknown to be benign (malware scanners), KQguard assumes the unknown to be a threat. Consequently, KQguard focuses on the legitimate code and it is largely independent of any concrete knowledge about the implementation specifics of KQ hijack attacks.

KQguard uses a compact and "relocatable" representation of callback functions (the Callback-Signature, explained in Section 3.2) for efficient and effective checking. In the following, we will refer to the Callback-Signatures of known legitimate event handlers as EH-Signatures. Under the assumption of full kernel source code availability and without concern for inclusion of closed source device drivers, we could use static analysis of the entire source code (kernel + drivers) to find all the EH-Signatures. However, this assumption is not always satisfied in practice.

To relax the full kernel source code availability assumption, we adopt dynamic analysis. Figure 2 shows the two-phase KQguard approach, based on dynamic analysis to satisfy the inclusiveness requirement (which avoids the source code assumption). In the first (training) phase, we collect all the legitimate EH-Signatures used by the system into the EH-Signature Collection (Section 3.3). Under the assumption of training in a controlled environment free of rootkits, no malware callback-signature will be generated, so we can guarantee zero false negatives in the second (guarding) phase. In addition, under the assumption of having a representative and comprehensive workload, dynamic analysis would also produce a complete EH-Signature Collection. Our experimental evaluation shows that it is feasible to reduce the false positives to zero and thus confirms the reasonableness of workload assumption.

The actual validation of callback functions requires simple but extensive kernel instrumentation. Conceptually, every KQ dispatcher needs to check every callback function invocation before execution. Section 3.4 describes the non-trivial search for all the locations of these simple changes. Concretely, we use static analysis of kernel code to find all the KQ dispatchers, which is another necessary condition for zero false negatives.

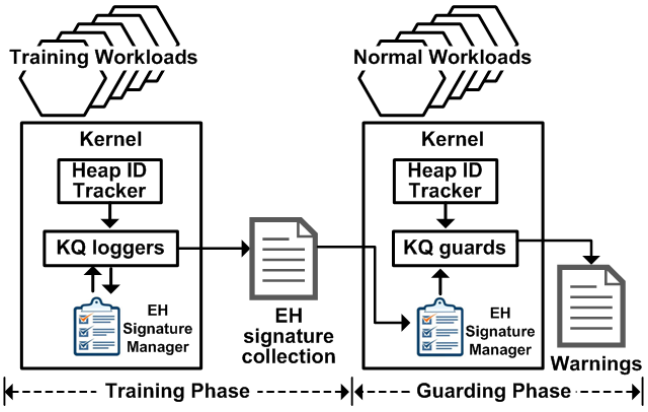

Figure 2: Overall Architecture of KQguard

3.2 Callback-Signature Abstract Data Type

The central data structure of KQguard is the *Callback-Signature*, an efficient and effective representation of callback functions for checking. At an abstract level, the Callback-Signature of a KQ is a symbolic pair (callback_function, callback_parameters). For clarity, we use the term *EH-Signature* to denote known Callback-Signatures of legitimate event handlers.

For different concrete types of KQ (e.g., timer_list in Figure 3), the syntax and semantics of their Callback-Signatures may vary, but their function is the same: to contain sufficient information to identify a legitimate KQ request with a known callback function. The two parts of Callback-Signature reflect the two attack modes described in Section 2.2. The callback_function part of Callback-Signature is used to protect the kernel against callback-into-malware. Both callback-function and callback_parameters parts are used to protect the kernel against callback-into-libc attacks.

```
struct timer_list {
    struct list_head list;
    unsigned long expires;
    unsigned long data;
    void (*function)(unsigned long);
};
```

Figure 3: The Definition of timer_list in Linux Kernel 2.4

To achieve efficient and effective checking, the Callback-Signature from a KQ request is generated at runtime and matched against the set of EH-Signatures generated in the training phase. We describe first the callback-function part of the Callback-Signature. The starting value is a physical address of an executable. Since a module's physical addresses from different reboots are unlikely to match, we perform a translation process that can be called "de-linking": it translates a linked callback function pointer back into the location-independent "object code" format prior to becoming linked into kernel memory. The callback-function pointer is translated back into module name and then a unique module ID, plus the displacement relative to the starting point of its containing module (usually a device driver). Under the assumption that the kernel maintains a uniform mapping of module location to module ID (e.g., driver's name), the pair (module ID, displacement) becomes an invariant representation of the callback-function pointer, suitable for KQguard checking.

The translation of the second part (callback_parameters) is more involved, since it may contain up to three sub-types: actual values, global variables, and heap objects. Actual values are non-pointer types such as integers, which require no translation. Global variables are pointers to kernel shared variables (Figure 4.a), and they are translated to a pair (module ID, displacement) the same way as the callback-function. Heap objects (Figure 4.b) could have been translated similarly, if the mapping between a heap object and its allocator is maintained by the kernel.

Since most kernels do not maintain the mapping between the requester and its allocated heap objects, we add this mapping to their heap allocator function (e.g., `ExAllocatePoolWithTag` in WRK). Specifically, the extended heap allocator function searches the call stack backwards until it reaches a return address that falls within the code segment of a device driver; then such a return address (after subtracting from it the starting address of the device driver) becomes the displacement part of the heap object representation. The extended heap allocator searches backwards on the call stack instead of stopping at the immediate return address because the immediate return address may be in some wrapper function for the heap allocation function (e.g., in WRK, `ExAllocatePoolWithTag` is called by `ObpAllocateObject`), and this kind of chained function call can continue (see an example in Figure 5); a kernel device driver can call a function at the top of the call chain to allocate a heap object (for example, `atapi.sys` calls `IoCreateDevice` to create a heap object).

In summary, the design goal of Callback-Signature is to support an efficient and effective runtime determination of the legitimacy of callback functions. The callback-function part detects callback-intro-malware attacks while the callback-parameters part detects callback-into-libc attacks.

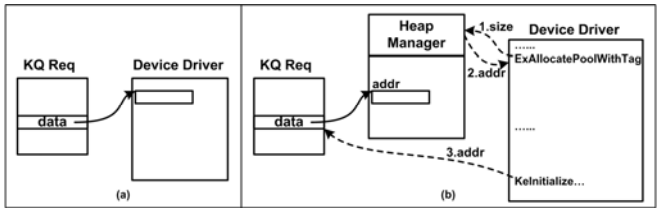

Figure 4: Illustration of Different Origins of callback_parameters: (a) Global Variable; (b) Heap Variable

3.3 Building the EH-Signature Collection

3.3.1 Design Requirements

The next step in the construction of KQguard defense is to collect the legitimate EH-Signatures. The set of all EH-Signatures forms the EH-Signature Collection. The challenge of building the EH-Signature Collection is to find the exact set of legitimate EH-Signatures. In the process of distinguishing malware Callback-Signatures from legitimate EH-Signatures, missing an EH-Signature causes false positives, denying the execution of a legitimate KQ request. Worse yet, including a malware Callback-Signature into the EH-Signature Collection causes false negatives, allowing malware to execute.

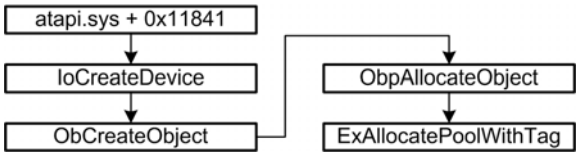

Figure 5: Indirect Heap Object Allocation in a Call Stack

In an ideal kernel development environment, one could imagine annotating the entire kernel and all device driver code to make KQs explicit, e.g., by defining a KQ abstract type. Processing the KQ annotations in the complete source code will give us the exact EH-Signature Collection. There are several reasons this ideal scenario is unlikely to happen. First, there are many third-party closed source device drivers that are unlikely to share their source code. Second, it is likely that human errors will be introduced in the large number of lines of kernel code, currently on the order of millions. Third, it is likely that most OS kernel builds will only contain a small subset of all device drivers, making it unwieldy to include all possible EH-Signatures in all systems.

Our design decision is to build software tools to automate the process of obtaining a specialized EH-Signature Collection that fits the configuration and usage of each system. In a training

phase, we run representative and comprehensive workloads. Under the assumption that the training is done in a secure environment, we consider all the KQ requests made as legitimate and include them into the EH-Signature Collection.

3.3.2 Constructing Callback-Signatures

The software tools used in the building of general Callback-Signatures and EH-Signatures in particular are small extensions of kernel facilities that manipulate KQs: KQ request creation, KQ request initialization, and KQ request dispatch. These are APIs provided by the core kernel for the device drivers. We extend these kernel calls with a KQ request logging utility, which records every KQ request mentioned in the legitimate kernel and device drivers. This happens because during the training phase our goal is to approach the complete coverage as much as possible.

Concrete examples of other useful kernel functions include device driver loader functions, which are extended to log address range information about loaded device drivers. They help us "de-link" (Section 3.2) the physical address in the Callback-Signature and map it back to its originating device driver. Similarly, the heap allocation functions are extended to provide allocation ranges of device driver requests of heap objects.

In general, the information contained in Callback-Signatures is readily available in the kernel, although the precise location of such information may differ from kernel to kernel. It is a matter of identifying the appropriate location to instrument the kernel to extract the necessary information. The extensions are applied to the kernel at source code level. The instrumented kernel is then rebuilt for the Callback-Signature collection process.

3.3.3 Automated Detection of KQs

Since every KQ can be exploited by malware, we need to build the EH-Signatures for all of KQs. This is straightforward for known KQs that have published APIs and source code as outlined in Section 3.3.2. Under the hypothesis that a system software provider can apply our method and tools to protect all KQs in a kernel, we can claim the completeness of our proposed solution.

On the other hand, for the purposes of our research, it is still important to find out the KQs in a large kernel such as WRK and Linux for several reasons. First, we need to evaluate the importance and potential impact of the KQ exploitation, which is related to the quantity and variety of KQs present in a kernel (the attack surface). Second, we want to demonstrate the effectiveness of KQguard for a realistic set of KQs. Third, by calling attention of the research community to the named KQs, we may be able to find additional KQ exploits that have not been identified so far.

Therefore, we designed and implemented a KQ discovery tool that automates the process of finding KQs in a kernel by analyzing its source code. Since kernel programmers are not intentionally hiding KQs, they usually follow similar programming patterns that our tool uses effectively:

- A KQ is typically implemented as a linked list or an array. In addition to insert/delete, KQ has a dispatcher that operates on the corresponding type.
- A KQ dispatcher usually contains a loop to act upon on all or a subset of queue elements. For example, `pm_send_all` in Figure 7 contains the dispatcher loop for the Power Management Notification queue of Linux kernel 2.4.32.
- A KQ dispatcher usually changes the kernel control flow, e.g., invoking a callback function contained in a queue element.

Based on the above analysis, the KQ discovery tool recognizes a KQ in several steps. It starts by detecting a loop that iterates through a candidate data structure. Then it checks whether a queue element is derived and acted upon inside the loop. Next, our tool marks the derived queue element as a taint source and performs a *flow-sensitive* taint propagation through the rest of the loop body; this part is flow-sensitive because it propagates taint into downstream functions through parameters (e.g., `dev` passed from `pm_send_all` to `pm_send` in Figure 7). During the propagation, our tool checks whether any tainted function pointer is invoked (e.g., `dev->callback` in `pm_send` in Figure 7), and if that is the case, it reports a candidate KQ. Pseudo code of our KQ discovery algorithm is shown in Figure 6. Due to space constraints we omit further details, but the results (e.g., KQs found in WRK) are interesting and shown in Sections 4.1.2 and 4.2.2.

For each function,
  For each *while* or *for* loop in that function,
    For each assignment statement L = R_exp inside the loop,
      if R_exp satisfies either of the following conditions:
        - it is an array element Ar[i] and Ar is a global array with a non-primitive element type (e.g., a structure or a function pointer),
        - it is a type casting (T) rr_exp, T is a structure type that has a *link* field (e.g., a pointer to its own type, or LIST_ENTRY) and rr_exp contains a pointer arithmetic operation,
      then, (1) set L as a tainted variable and perform a transitive taint analysis through the rest of the loop body; (2) if during the taint analysis, some tainted function pointer is invoked, report the top-level data structure (e.g., Ar[] or rr_exp) as a potential KQ.

Figure 6: KQ Detection Algorithm

3.3.4 Dynamic Profiling to Collect EH-Signatures

Under the assumption of full kernel source code access, we can generate the code for building the EH-Signature for all known KQs. By finding all the KQ invocations, theoretically we will be able to build corresponding EH-Signatures. Unfortunately, this method requires the source code of third-party device drivers, which often are unavailable.

Instead of relying on the full source code access assumption, we chose to relax this assumption by adopting a practical method based on dynamic profiling. We run a set of representative set of benchmark applications using a comprehensive workload. We call this process of collecting the EH-Signatures "training phase", since it is somewhat reminiscent of training a filter in machine learning. During the training phase, the kernel extensions described in Section 3.3.2 is triggered by every KQ request and logged.

To avoid false negatives in KQguard, the training phase is performed in a controlled environment (a clean and instrumented set of kernel and device drivers) to ensure no malware Callback-Signatures are included. To avoid false positives, the training phase needs to be comprehensive enough to trigger all of the

```
/* linux-2.4.32/kernel/pm.c */
int pm_send_all (pm_request_t rqst, void *data)
{
      struct list_head *entry;
      entry = pm_devs.next;
      while (entry != &pm_devs) {
            struct pm_dev *dev = list_entry(entry,
struct pm_dev, entry);
            if (dev->callback) {
               int status = pm_send(dev, rqst, data);
               ……
            }
            entry = entry->next;
      }
      ……
      return 0;
}
```

```
struct pm_dev
{      pm_dev_t        type;
       unsigned long    id;
       pm_callback      callback;
       ……
       struct list_head entry;
};
```

```
/* linux-2.4.32/kernel/pm.c */
int pm_send(struct pm_dev *dev,
pm_request_t rqst, void *data)
{     ……
      if (prev_state != next_state){
……
          status = (*dev->callback) (dev,
rqst, data);
          ……
      } ……
}
```

```
/* linux-2.4.32/kernel/pm.c */

static LIST_HEAD(pm_devs);
struct pm_dev *pm_register(pm_dev_t
type,  unsigned long id, pm_callback call-
back)
{
      struct pm_dev *dev =  kmalloc (sizeof
(struct pm_dev), GFP_KERNEL);
      if (dev) {
            memset(dev, 0, sizeof(*dev));
            dev->type = type;
            dev->id = id;
            dev->callback = callback;
            list_add(&dev->entry,
&pm_devs);
      }
      return dev;
}
```

Figure 7: Details of the Power Management Notification Queue on Linux Kernel 2.4.32

legitimate KQ requests. In practice, we expect an OS software vendor (e.g., Microsoft), a capable system software reseller (e.g., Red Hat), or a reputable software validator (e.g., a software security company) to perform the training phase, which is similar to (and uses the same tools of) the comprehensive testing before a release. As in all software testing, missing some execution paths may introduce false positives when a previously unseen Callback-Signature is triggered during runtime. We note that our coverage requirements are significantly less than software correctness testing, since we only need to trigger each execution path that invokes a KQ request once, not the entire input space.

In our experiments, we approximate the software validation process in two ways. First, we choose a representative set of benchmark programs (e.g., PerformanceTest, Super PI, and Iozone). Second, in addition to the test input, we also run the instrumented kernel during normal usage, including rebooting the OS. Our evaluation (Section 5) shows a very low false positive rate, indicating the feasibility of the dynamic profiling method. In general, the issue of test coverage for large scale software without source code is a significant challenge and beyond the scope of this paper.

3.4 Validation Using EH-Signature Collection

We modify the dispatcher of every identified KQ so that the dispatcher checks the legitimacy of a pending KQ request before invoking the call back function. To perform the check, the dispatcher first build the Callback-Signature from a pending request, and then matches the Callback-Signature against the EH-Signature Collection (the complete list of legitimate event handlers, see Section 3.3). If a match is found, the dispatcher invokes the confirmed event handler. Otherwise, the dispatcher takes necessary actions against the potential threat (e.g., generating a warning message).

The construction of the KQ request's Callback-Signature follows the same algorithm as the construction of EH-Signatures (Section 3.3.2), including the de-linking of callback-function and the analysis of callback-parameters by sub-type (integer values, global variables, and heap objects).

4. IMPLEMENTATIONS OF KQGUARD

4.1 WRK Implementation of KQguard

The KQguard design (Section 3) was implemented on the WRK. Our implementation consists of about 2,300 lines of C code and 2,003 lines of Objective Caml code.

4.1.1 Construction of Callback-Signatures in WRK

In order to collect the Callback-Signatures for the 20 KQs in the WRK, we instrumented the kernel in two sets of functions. First, we studied and modified the functions that manipulate the KQs directly. Second, we studied and modified auxiliary functions that contribute with useful information in the construction of Callback-Signatures.

The main functions that manipulate KQs include the functions that initialize KQ requests, insert KQ requests, and dispatch KQ requests. These functions have direct access to KQ request data structures and can provide both the callback-function and callback-parameters parts of a Callback-Signature. For both EH-Signature collection in the training phase and for Callback-Signature validation in the testing phase, we instrument every KQ manipulation function. This is because sometimes a KQ request can be removed before the dispatching (e.g., a timer request can be cancelled). In total, our instrumentation of the KQ functions consists of 500 lines of C code in WRK. The specific KQ dispatch functions instrumented are list in Table 2, and the specific KQ request insertion functions instrumented can be found in Table 9 in the Appendix.

As explained in Section 3.3.2, the construction of a Callback-Signature involves the de-linking of callback-function address and global variables, and the identification of heap objects. The de-linking follows the algorithm outlined in Section3.2. The analysis of heap objects is somewhat specific to WRK. It starts from the instrumentation of heap allocation and deallocation functions, which in WRK include `ExAllocatePoolWithTag` and `ExFreePool`. Our instrumentation maintains a table of tuples <address, length, driver ID, call site>, where address and length are the starting address and length of an allocated heap memory block at runtime, and the driver ID and call site identify the requester or "owner" of the heap memory block. When a heap memory block

is allocated, a new entry is inserted into this table. Symmetrically, the instrumented deallocation function removes the entry with key <address>. The driver ID is derived from a search of the call stack backwards until it reaches a return address that falls within the code segment of a device driver. The return address is de-linked (subtracting from it the starting address of the device driver), becoming the call site part of the tuple for the heap object. The call stack walk resolves chained function calls (Figure 5). Our instrumentation of the heap allocator / deallocator consists of 600 lines of C code in total.

4.1.2 Automated Detection of KQs for the WRK

As explained in Section 3.1, we alleviate the assumption of full kernel source code availability through a combination of static analysis to find the KQs and dynamic analysis to build the EH-Signature Collection. We implemented the KQ discovery tool for WRK using the C Intermediate Language (CIL) [19]. Our implementation consists of 2,003 lines of Objective Caml code.

We applied the KQ discovery tool to the WRK source code (665,950 lines of C) and 20 KQs were detected with 16 distinct KQ data structure types (some KQs share a common type, e.g., Process Creation/Deletion Notification queue and Thread Creation/Deletion Notification queue). Due to space constraints, we list a few representative KQs in Table 1, and the full list in Table 9 of the Appendix. Most of these KQs are implemented as linked lists (such as the I/O timer queue), but a few of them are implemented as arrays (e.g., the Process Creation/Deletion Notification queue). A majority of these KQs (15 out of 20) have a global variable as their queue head. For each KQ, we report the number of callback-parameters that are included in the Callback-Signature (in the "# param. tainted" column). As illustration of KQ manipulation functions, we include the KQ insertion function for each KQ.

4.1.3 EH-Signature Collection

After the KQs have been found, we instrument these KQ-related functions (insertion, initialization, and dispatch) as described in Sections 3.2 and 3.3). Then we run the training workloads (Section 5.1) in a training process (Section 4.1.3). The Callback-Signatures collected during training are assumed to be legitimate and incorporated into the EH-Signature Collection.

We developed a set of utility functions to manage the Callback-Signatures, including the EH-Signature Collection. These functions support the generation, comparison, insertion, and search of Callback-Signatures. For example, during the training phase, the same EH-signature can be detected repeatedly from time to time, so one helper function checks for duplicate EH-signatures. These helper functions are implemented in 900 lines of C code, and they are invoked both in the training phase (for EH-Signature collection) and in the testing phase (to query EH-Signatures during Callback-Signature validation).

Table 1: Representative Automatically Detected KQs

KQ Name	Queue header variable	# param. tainted	Name of Request Insertion Function(s)
Windows Research Kernel			
I/O timer queue	IopTimer-QueueHead	2	IoInitializeTimer
RegistryCall-back queue	CmpCallBack-Vector	1	CmRegisterCallback
Load image notification queue	PspLoadImage-NotifyRoutine	0	PsSetLoadImageNo-tifyRoutine
Linux Kernel 2.4.32			
Tasklet queue	tasklet_vec[], tasklet_hi_vec[]	1	tasklet_schedule, tasklet_hi_schedule
Packet type queue	ptype_all, ptype_base[]	1	dev_add_pack

4.1.4 Validation of Callback-Signature in WRK

We instrument the dispatcher of every identified KQ in the WRK so that the dispatcher checks the legitimacy of a pending KQ request before invoking the call back function. To perform the check, the dispatcher first retrieves the Callback-Signature from a pending request, and then matches the Callback-Signature against the EH-Signature Collection. If a match is found, the dispatcher invokes the call back function; otherwise, the dispatcher takes necessary actions (e.g., generating a warning message).

More specifically, we instrument the dispatch functions listed in Table 2, and our instrumentation consists of about 300 lines of C code.

The current implementation of Callback-Signature validation in the dispatcher follows directly from the conceptual design. It guarantees that only a validated KQ request will be executed. However, we are aware of the lengthened critical path for event handling in the kernel. We are considering several implementation alternatives that will take the validation code out of the critical path in the KQ dispatcher. For example, validation can be done at the KQ request insertion time. We will add a validation stamp (an encrypted checksum) to protect each validated KQ request. At the KQ request dispatch time, the validation stamp can be quickly checked with small constant overhead. All unknown and invalid KQ requests (e.g., the ones inserted without using the official API and thus without the stamp) will be rejected.

4.2 Linux Implementation of KQguard

4.2.1 Construction of Callback-Signatures in Linux

We have started the work on the Linux implementation of KQguard, which follows the same conceptual design described in Section 3. We instrumented the Linux kernel functions that manipulate the Linux KQs. Of course, the names and syntax of specific functions that we instrument are different: they are listed in Table 10 of the Appendix, in the last two columns.

As a concrete example, the de-linking of Callback-function addresses and global variables as well as the identification of heap objects follow the same design (Section 3.3.2) as in our WRK implementation, but the heap object identification uses `kmalloc` and `kfree`, which are specific to Linux.

4.2.2 Automated Detection of KQs in Linux

We apply the KQ detector to Linux kernel 2.4.32 (with 482,369 lines of C code) and it found 22 KQs. Two representative KQs are listed in Table 1, and the full list is in Table 10 of the Appendix.

Perhaps not surprisingly, we found differences as well as similarities between WRK and Linux. As an example of differences, WRK KQs are implemented both as linked lists and arrays, but all Linux KQs are implemented as linked lists. More concretely,

WRK and Linux have different name conventions: the WRK linked list implementation of KQs utilizes the structure type LIST_ENTRY, while the Linux linked list uses the structure type list_head. With minor modifications to reconcile these differences, our KQ detector is able to analyze both the WRK and the Linux kernel.

4.2.3 Validation of Callback-Signature in Linux

We instrument the functions listed in the "name of dispatch functions" column of Table 10 to validate the pending KQ requests in Linux. The retrieval of Callback-Signature follows the same process as the WRK implementation, and the EH-Signature management functions were ported from our WRK implementation.

5. Evaluation of KQguard in WRK

Due to the ongoing efforts to implement KQguard in Linux (Section 4.2), we only report the evaluation results of the more mature WRK implementation of KQguard in this section.

Table 2: WRK functions modified to guard the KQs

Kernel Queue	Dispatcher Functions Modified
Per-stream context queue	FsRtlTeardownPerStreamContexts
I/O timer queue	IopTimerDispatch
File system registration change notification queue	IoRegisterFileSystem, IoUnregisterFileSystem, IoRegisterFsRegistrationChange
Process creation/deletion notification queue	PspCreateThread, PspExitProcess
Driver Reinitialize routine queue	IopCallDriverReinitializationRoutines
Boot driver reinitialize routine queue	IopCallBootDriverReinitializationRoutines
Thread creation/deletion notification queue	PspCreateThread, PspExitThread
RegistryCallback Queue	CmpCallCallBacks
Load-image notification queue	PsCallImageNotifyRoutines
Bug check reason callback queue	KiInvokeBugCheckEntryCallbacks
Callback object queue	ExNotifyCallback
Waiting IRP queue	FsRtlRemoveAndCompleteWaitIrp, FsRtlUninitializeOplock
IRP waiting lock queue	FsRtlPrivateCancelFileLockIrp
Firmware table providers queue	ExpGetSystemFirmwareTableInformation
BugCheckCallback routine queue	KiScanBugCheckCallbackList
Deferred write queue	CcPostDeferredWrites
Change directory notification queue	FsRtlNotifyFilterReportChange
System worker thread queue	ExpWorkerThread
The DPC queue	KiExcuteDpc, KiRetireDpcList
The APC queue	KiDeliverApc

5.1 Experimental Method and Setup

We evaluate both the effectiveness and efficiency of KQguard through measurements on production kernels. By effectiveness we mean precision (whether it misidentifies the attacks found, measured in false positives) and recall (whether it misses a real attack, measured in false negatives) of KQguard when identifying KQ hijack attacks. By efficiency we mean the overhead introduced by KQguard.

We divide the effectiveness evaluation into two groups: defense against real malware attacks (Section 5.2) and defense against synthetic attacks (Section 5.3). Real malware represents an easier class of attacks (only callback-into-malware). Synthetic attacks include both callback-into-malware and callback-into-libc attacks. We show that KQguard can detect all the real malware and all the synthetic attacks (no false negatives). In Section 5.4, we study the potential false positives from KQguard, which may arise due to an incomplete EH-Signature Collection or imperfect Callback-Signature design. We found no false positives in our experiments.

The efficiency evaluation of KQguard introduces some new challenges due to the asynchronous nature of KQ executions. First, the application benchmarks do not invoke directly any KQ requests. Second, the kernel services that create KQ request are not necessarily invoked directly by applications. Consequently, besides microbenchmarks that measure direct invocations of KQ operations, it is non-trivial to measure directly the impact on performance caused by the various KQs.

The microbenchmark 1 in Section 5.5 measures the actual time taken to run the Callback-Signature validation algorithm of KQguard. The microbenchmark 2 measures the difference between a kernel with and without KQguard when handling specific events. Both show a very small overhead, particularly when compared to typical callback functions that it guards.

The macrobenchmarks in Section 5.5 measure the total elapsed time of an application macrobenchmark run. The design rationale of the experiments is that various kernel services are triggered during a resource-intensive application execution. Consequently, the total elapsed time is a direct measure of application resource requirements plus kernel requirements. By comparing the elapsed time of the original kernel (without KQguard) and the instrumented kernel with KQguard, we have a measure of the additional overhead introduced by the KQguard. This is a composite measure of the accumulated overhead for all the KQs.

All the experiments are run on a 2.4 GHz Intel Xeon 8-Core server with 16 GB of RAM. The host operating system is Microsoft Windows XP Service Park 3 running Microsoft Virtual PC 2007 (version 6.0.156.0). The guest operating systems is Windows Server 2003 Service Pack 1 running the WRK, and it is allocated 256 MB of RAM and 20 GB of hard disk.

Workloads. For performance evaluation, we have chosen several resource-intensive application benchmarks. They have been chosen because each one saturates some resource and therefore kernel overhead should have a noticeable effect on the elapsed time. The first benchmark is Super PI, a CPU-intensive workload calculating 32 million digits of **π.** The second benchmark copies a directory with a total size of 1.5 GB, which stress the file system. The third benchmark is also CPU-intensive, performing the compression and decompression of the 1.5 GB directory with 7-Zip. The fourth benchmark downloads a 160 MB file with WinSCP, which

stresses the network connection. We also ran standard benchmarks such as PostMark and PassMark.

For effectiveness evaluation, we run a set of normal workload programs in the modified WRK to measure the false positive rate of our KQ guarding. These programs include Acrobat Reader, Windows Driver Kit, Firefox, Windows Media Player, Easy Media Player, and several games (Minesweeper, Microsoft Hearts Network, and 3D pinball). We chose these programs to represent the normal workload on a Windows platform, for example document processing, programming (Windows Driver Kit has tools to compile device drivers), web browsing, and entertainment (multimedia applications and gaming). This set of normal use programs showed zero false positives (Section 5.4).

For the false negatives part of effectiveness evaluation, we chose 11 real-world malware samples (section 5.2) and nine synthetic test device drivers (section 5.3) as test workloads. The real-world malware samples come in the form of Windows executable files, and they can be activated by running the respective executables (e.g., `malware.exe` in Figure 8), which includes the automatic installation of malicious device drivers that hijack KQs. Once active, the malicious drivers initiate the KQ hijack attacks that KQguard detects. The synthetic device drivers that we develop are manually installed into the WRK using the "Add Hardware Wizard" from the Control Panel.

5.2 Real World KQ Hijack Attacks

We start our evaluation of KQguard effectiveness by testing our WRK implementation (Section 4.1) against real work KQ Hijack attacks in Windows OS. Such malware samples are available from open sources such as Offensive Computing (http://offensivecomputing.net). So far, we studied 10 spam bots downloaded from this website.

To test the malware samples in a safe environment, we created a malware analysis laboratory consisting of several dedicated virtual machines that have no access to the public network. We start the virtual machine monitor (e.g., VMware Workstation) in a clean state and boot a clean copy of WRK instrumented with KQguard. We run each malware sample in the virtual machine and collect the monitoring log generated by KQguard during the malware's execution.

We have studied the KQ hijacking behavior of several well-known malware (e.g., spam bots), and report the results in Table 3. We started with malware with reported KQ Hijack Attacks, which are marked with a "√" with citation. We were able to confirm some of these attacks, shaded in green. The rows with green "√" without citations are confirmed new KQ Hijack Attacks that have not been reported by other sources.

We discuss the KQ Hijack Attack in Rustock as an illustrative example. It is known [2][15][23] that Rustock.C uses the Timer queue, the Create Process Notification queue, and the APC (Asynchronous Procedure Call) queue. However, all the Rustock.C samples that we obtained failed to run in our lab, perhaps due to the virtual environment. Instead, we were able to run a closely related variant, Rustock.J, and we confirmed that Rustock.J uses the KQ Hijack Attack on Load-Image Notification queue, the Create Process Notification queue, and the APC queue. Figure 8 shows a screenshot in which KQguard for the APC queue generated a warning message about a suspect APC callback function at address `0xF83FE316`, which falls within the address range of a device driver called msliksurserv.sys that is loaded by Rustock.J. This confirmed detection of Rustock.J hijacking the APC queue marks the corresponding entry in Table 3 as green.

To confirm the origin of callback-into-malware, we use WRK's `PsLoadedModuleList`. However, advanced malware such as Rustock.J can remove their entries from this global list in order to hide. To find the true information (e.g., address range) about a hidden malicious device driver, we instrument the device driver loading function of the WRK to log information of every loaded driver, including its name and address range. We find this approach effective in detecting hidden drivers installed by Rustock.J, Pushdo/Cutwail, Storm/Peacomm, Srizbi, TDSS, and ZeroAccess.

Another technique that we use to attribute KQ requests to a malware sample is call stack information. For example, when a new DPC request structure is initialized via the API `KeInitializeDpc`, we check the device driver calling this API. In order to confirm the origin of the caller, we modify functions such as `KeInitializeDpc` to traverse the call stack,

Table 3: Known KQ Hijack Attacks in Current Malware

KQ => ---------- Malware	Timer /DPC	Worker Thread	Load Image Notify	Create Process Notify	APC	FsRegistration Change	RegistryOpCallback
Rustock.C	√ [2] [15]			√ [23]	√ [23]		
Rustock.J			√	√	√		
Pushdo / Cutwail	√			√ [8]	√	√ [8]	√ [8]
Storm / Peacomm	√		√ [4]		√ [20]		
Srizbi	√				√		
BlackEnergy			√ [11]				
Brazilian banker			√ [11]				
TDSS			√		√	√	
Duqu	√		√ [13]		√		
ZeroAccess	√	√ [9]			√ [9]		√
Koutodoor	√			√			
Pandex					√		
Mebroot	√						

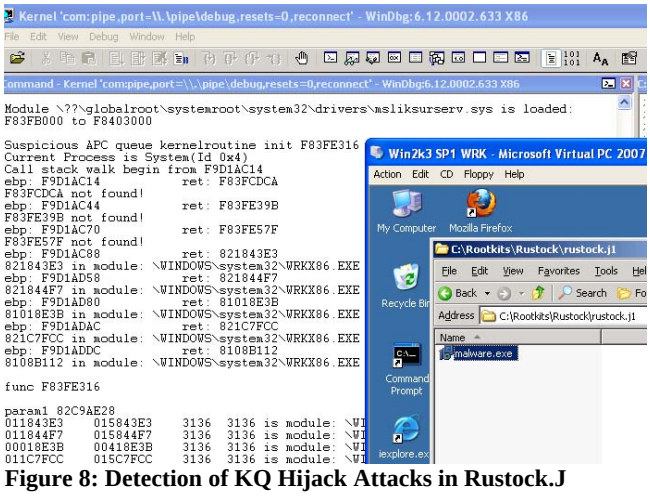

Figure 8: Detection of KQ Hijack Attacks in Rustock.J

which contains the return addresses in the caller, the caller's caller, and so on. By mapping the caller function(s) to the loaded device drivers, we can find any malicious driver(s) anywhere in the call chain, thus confirming the KQ Hijack Attack on DPC. Call stack walk has helped us confirm the KQ Hijack Attacks in Rustock.J, Storm/Peacomm, and ZeroAccess.

For all the malware that we were able to activate, we confirmed the reported KQ Hijack Attacks, except for the Duqu attack on Load-Image Notification queue. The study of Rustock shows that malware designers have significant ability and flexibility in hijacking different KQs. Concretely, Rustock.J has stopped using the Timer queue, which Rustock.C uses, but Rustock.J started to use the Load-Image Notification queue, which Rustock.C does not. This may have happened to Duqu's attack on the same queue, or Duqu does not activate the attack on Load-Image Notification queue during our experiment. This kind of evolution underscores the importance of guarding all KQs preventively, not just reacting to the ones that have been reported.

5.3 Protection of All KQs

In addition to real world malware, we created synthetic KQ Hijack Attacks for two reasons. First, the malware testing was incomplete, as shown by several KQs (9 in Table 8 have maximum queue length of zero) that have not been called during our testing. Examples of such KQs include the Thread Creation/Deletion Notification queue and the Waiting IRP queue. Second, the malware analyzed in Section 5.2 belong to the callback-into-malware category. Although there have been no reports of callback-into-libc attacks in the wild, it is important to evaluate the effectiveness of KQguard for both kinds of attacks.

For completeness, we developed a test Windows device driver for each of the nine KQs that have not been called (zero length in Table 8). These test drivers have a common code pattern: initializing and inserting a KQ request data structure into their respective KQs, and using the same callback function (which calculates a factorial of a small fixed number). The main difference among the test drivers is the specific APIs called to initialize and insert KQ requests. These test drivers are not part of the normal set of legitimate drivers. Therefore, the KQguard should raise alarms when such test drivers become active.

We have confirmed that our KQ defense can detect all the test device drivers, which suggests that our defense is effective against potential and future KQ Hijack attacks. We observe that the KQguard defense does not contain specific knowledge of the malware internals or the internals of the test drivers. Consequently, we have confidence that KQguard defense will work for a wide range of KQ Hijack Attacks regardless of their implementation details.

5.4 False Alarms

The complete defense (zero false negatives) achieved by KQguard against malware (Section 5.2) and synthetic test drivers (Section 5.3) may suggest that KQguard could be easily triggered. In experiments running a variety of benchmarks and normal use application workloads, we have found zero false positives. This is achievable when two conditions hold: (1) the Callback-Signature validation against EH-Signature Collection has no errors, and (2) the training workload is comprehensive to produce the full EH-Signature Collection.

To evaluate the false positive rate of our KQ guarding, we first run the WRK in the training mode to collect EH-Signatures. We let the machine run for half an hour, during which time we run Notepad, My Computer, WinSCP, and Internet Explorer; then we reboot the machine and repeat the process. During the first run we collect 855 EH-Signatures, after the first reboot we collect 10 more EH-Signatures, and after the second, third, and fourth reboots we collect three, one, and one new signature, respectively. After the fifth reboot we do not see any new signatures. Therefore, we stop the training after five reboots with 870 EH-Signatures in the EH-Signatures Collection.

Next we rebuild the WRK so that it will be running in the test mode based on the EH-Signatures Collection. Then we reboot the virtual machine using the instrumented WRK. During the period of more than one day, we run the workload programs discussed in Section 5.1, as well as other normal applications to keep the instrumented WRK active as much as possible. Throughout the testing phase, we observe zero false alarms.

The zero false positives result shows that both conditions hold. First, we have an effective Callback-Signature validation algorithm capable of matching all legitimate KQ requests against the EH-Signatures Collection. Second, the training phase outlined in Section 3.3 has built a full EH-Signatures Collection with respect to the workload described in Section 5.1.

While the experimental result appears encouraging, we avoid making a claim that dynamic analysis will always achieve zero false positives. Instead, we note that the 870 EH-Signatures are not collected in one run. Concretely, 15 of them are collected after rebooting the machine at least once. We analyzed the details of these 15 "late-coming" EH-Signatures to better understand our dynamic analysis based signature collection. These 15 EH-Signatures are different from previous signatures in terms of either the callback-function (4 cases) or the callback-parameters (11 cases). More specifically, the new callback functions can be from the core kernel or a device driver, and the new callback parameters can be a global variable or a heap variable. This variety suggests some potential limitations of a dynamic analysis based approach. Fortunately, our experience suggests that the set of legitimate KQ signatures can be collected within a few reboots.

We note that an alternative to dynamic analysis is the full kernel source code availability assumption. For example, a software vendor in possession of the entire kernel source code and device drivers (with or without source code), should be able to build the full EH-Signatures Collection through a combination of static and dynamic analysis as described in this paper.

5.5 Performance Overhead

As mentioned in Section 5.1, the performance evaluation of KQguard is non-trivial due to the asynchronous nature of KQ operations. We divide the performance measurements into two steps. First, we measure the overhead of KQguard through microbenchmarks that invoke those operations directly. Second, we measure the difference in elapsed time of application macrobenchmarks by comparing the measured response time on the original kernel and the KQguard-instrumented kernel.

For the first step, we measured the overhead of KQguard validation check and heap object tracking. KQguard validation check matches Callback-Signatures against the EH-Signature Collection, and its overhead consists of matching the callback-function part and the callback-parameters part. Heap object tracking affects every heap allocation and deallocation operation (e.g., ExAlloca-

tePoolWithTag and ExFreePool in WRK). These heap operations are invoked at a global level, with overhead proportional to the overall system and application use of the heap.

Microbenchmark 1: we measured the total time spent in performing 1,000 KQguard validation checks for the DPC queue and the I/O timer queue. We chose these two KQs because they are more actively invoked than other KQs (Table 8). The main result is that global heap object tracking during the experiment dominated the KQguard overhead. Table 4 shows that DPC queue validation consumed 93.7 milliseconds of CPU, while heap object tracking consumed 8,527 milliseconds. These 1,000 DPC callback functions are dispatched over a time span of 250,878 milliseconds (4 minutes 11 seconds). The total CPU consumed by our KQguard validation for DPC queue and the supporting heap object tracking is 8,620.7 milliseconds (about 3.4%). The measurements of the I/O timer queue confirm the DPC queue results.

Microbenchmark 2: we measure the overhead based on each individual KQ callback function. This is because the validation checks for each individual callback function may vary in complexity (e.g., they may check different numbers of parameters and the de-linking of each parameter may also be different across callback functions), and each callback function may require different amount of execution time, so the relative slowdown caused by our validation may vary from one callback function to another.

Table 5 shows overhead measurements for three representative KQ callback functions in the DPC queue and one callback function in the I/O timer queue. For each callback function, we measure its execution time, including the validation check and the actual execution in the original WRK (shown in the "Original" column); then we measure the same callback function with KQguard and show the result in the "With KQguard" column. From the difference between these two execution times we calculate the signature validation check time (shown in the "Signature Validation" column). Then the overhead is calculated by dividing signature validation time by the original execution time. The execution time results for each callback function are averaged across multiple invocations of that callback function and we report the number of measurements in the last column of Table 5.

Table 4: KQguard Overhead (1,000 Callback Function calls)

KQ	Heap Object Tracking Time (milliseconds)	KQ Validation Time (milliseconds)	Total Time (milliseconds)	CPU Overhead per minute
DPC queue	8,527	93.7	250,878	3.4%
I/O timer queue	11,807	180	345,825	3.5%

Table 5: KQguard Overhead of 4 Callback Functions (C_1: acpi.sys+0x2c50; C_2: atapi.sys +0x8a0a; C_3:acpi.sys+0x6cce; C_4: classpnp.sys+0x1069)

KQ: Callback function	Original (us)	With KQguard (us)	Signature Validation (us)	Overhead	Number of measurements
DPC: C_1	461 ±102	471 ±102	10.2 ±0.6	2.2%	26
DPC: C_2	94 ±2.8	184 ±4.2	90 ±2.5	96%	505
DPC: C_3	286 ±31	296 ±31	10 ±0.27	3.5%	18
I/O Timer: C_4	1,470 ±174	1,577 ±174	107 ±13	7.3%	100

Table 5 shows the execution times of different callback functions varying significantly, from 94 to 1,470 microseconds. In comparison, the signature validation time is relatively small (from 10 microseconds to 107 microseconds). As a result, the relative overhead added by KQguard varies from 2.2% for callback function `acpi.sys+0x2c50` to 96% for callback function `atapi.sys+0x8a0a`. The main source of variation comes from heap object tracking. These overhead measurements were obtained with a relatively busy heap: an average of 164 heap operations per second, with 37,000 to 38,000 objects in the heap, and a peak of 39,681 objects.

For the second step, Table 6 shows the results of 5 application level benchmarks that stress one or more system resources, including CPU, memory, disk, and network. Each workload is run at least 5 times and the average is reported. We can see that in terms of execution time of the selected applications, KQguard incurs modest elapsed time increases, from 2.8% for decompression to 5.6% for directory copying. These elapsed time increases are consistent with the microbenchmark measurements (Table 4 and Table 5), with higher or lower heap activities as the most probable cause of the variations.

Table 7 shows the results of running the PostMark file system benchmark. The KQguard-instrumented WRK increases the read/write throughput of the file system to drop by about 3.9%. Similar results were found when we ran the PassMark PerformanceTest benchmark, with an execution time increase of 4.9%, and memory consumption increase of 2.9%.

To better understand the reasons for the overhead, we measure the maximum KQ length (i.e., the maximum number of pending requests for each KQ) and the dispatch frequency of the 20 KQs (the APC queue is divided into two sub-queues) during the experiments, as shown in Table 8. The measurements show that 9 KQs have not been called (0 maximum queue length). Of the 11 KQs that have been called, 5 KQs have infrequent dispatches (0 per minute) and 6 KQs are active (non-zero requests per minute). In terms of invocation frequency, the APC queue is the most active: 372 callbacks occur per minute on average. All KQs taken together, callbacks happen 787 times per minute, or 13 times per second.

Table 6: Performance Overhead of KQ Guarding in WRK

Workload	Original (seconds)	KQ Guarding (seconds)	Slowdown
Super PI	2,108 ±41	2,213 ±37	5.0%
Copy a 1.5 GB directory	231 ±9.0	244 ±15.9	5.6%
Compress a 1.5 GB directory	1,113 ±24	1,145 ±16	2.9%
Decompress a 1.5 GB directory	181 ±4.1	186 ±5.1	2.8%
Download a 160 MB file	145 ±11	151 ±11	4.1%

Table 7: Performance Overhead Measured by PostMark

Workload	Original (MB/s)	KQ Guarding (MB/s)	Overhead
Read 855 MB data	3.41±0.11	3.29±0.07	3.5%
Write 1,600 MB data	6.38±0.20	6.13±0.23	3.9%

Table 8: Frequency and intensity of KQ invocation in WRK

K-Queue Name		Max. Length	Frequency of dispatch (per minute)
Per-stream context queue		0	0
I/O timer queue		8	181
File system registration change notification queue		9	2
Process Creation/Deletion notification queue		7	3
Driver Reinitialize routine queue		8	0
Boot driver Reinitialize routine queue		5	0
Thread Creation/Deletion notification queue		0	0
RegistryCallback queue		0	0
Load image notification queue		0	0
Bug check reason callback queue		1	0
Callback object queue		4	0
Waiting IRP queue		0	0
IRP waiting lock queue		0	0
Firmware table providers queue		0	0
BugCheckCallback routine queue		7	0
Deferred write queue		0	0
Change directory notification queue		0	0
APC queue	KernelRoutine	476	312
	NormalRoutine	203	60
DPC queue		127	226
System worker thread queue		15	3
Total		**870**	**787**

6. Related Work

In this section, we survey related work that can potentially solve the KQ hijacking problem and satisfy the five design requirement: efficiency, effectiveness, coverage, extensibility, and inclusiveness (Section 2.3).

SecVisor [25] or NICKLE [24] are designed to preserve kernel code integrity or block the execution of foreign code in the kernel. They can defeat callback-into-malware KQ attacks because such attacks require that malicious functions be injected somewhere in the kernel space. However, they cannot detect callback-into-libc attacks because such attacks do not inject malicious code or modify legitimate kernel code. HookSafe [31] is capable of blocking the execution of malware that modifies legitimate function pointers to force a control transfer to the malicious code. However, HookSafe cannot prevent KQ hooking attacks because they do not modify existing and legitimate kernel function pointers but instead supply malicious function pointers in their own memory (i.e., the KQ request data structures).

CFI (Control Flow Integrity) [1] can ensure that control transfers (including invocations of function pointers) of a given program always conform to a predefined control flow graph. Since KQs involve function pointers (i.e., the callback functions), CFI is a relevant solution. However, CFI requires a *fixed* control flow graph but the introduction of new device drivers definitely requires a change to the control flow graph, so CFI does not satisfy the extensibility requirement. SBCFI [22] performs a garbage-collection style traversal of kernel data structures to verify that all function pointers encountered target trusted addresses in the kernel, so SBCFI can potentially detect a callback-into-malware KQ attack. However, SBCFI is designed for *persistent* kernel control flow attacks (e.g., it only checks periodically) but KQ hijacking attacks are *transient*, so SBCFI may miss many of them. Moreover, SBCFI requires source code so it does not satisfy the inclusiveness requirement. IndexedHooks [16] provides an alternative implementation of CFI for the FreeBSD 8.0 kernel by replacing function addresses with indexes into read-only tables, and it is capable of supporting new device drivers. However, similar to SBCFI, IndexedHooks requires source code so it does not satisfy the inclusiveness requirement.

PLCP [32] is the most comprehensive defense against KQ hijacking attacks so far, capable of defeating both callback-into-malware and callback-into-libc attacks. The basic idea of PLCP is to check the legitimacy of every pending KQ request before servicing it; the check is performed not only on the callback function but also on all possible function pointers reachable from the contextual data by the control flow of the callback function. In order to identify all such "check points" as well as the white list for each "check point", PLCP employs static program analysis (e.g., points-to analysis and transitive closure analysis) of the kernel source code. However, PLCP has some limitations: it does not satisfy the inclusiveness requirement due to its reliance on source code; it has high performance overhead so it does not satisfy the efficiency requirement. For example, in the worst case, it can cause 15 times slowdown to the applications.

7. Conclusion

Kernel Queue (KQ) Hijack Attacks are a significant problem. We outlined 11 real world malware attacks [2][4][8][9][11][13][15][20][23] and 9 synthetic attacks to cover all the KQs in the WRK. It is important for a solution to have 5 requirements: efficiency (low overhead), effectiveness (precision and recall of attack detection), coverage (protecting all KQs), extensibility (accommodation of new KQs) and inclusiveness (protection of kernels with and without source code). Current kernel protection solutions have difficulties with simultaneous satisfaction of all four requirements.

We describe the KQguard approach to defend kernels against KQ Hijack Attacks. The design of KQguard is independent of specific details of the attacks. Consequently, KQguard is able to defend against not only known attacks, but also anticipated future attacks on currently unscathed KQs. We evaluated the WRK implementation of KQguard, demonstrating the effectiveness and efficiency of KQguard by running a number of representative application benchmarks. In effectiveness, KQguard achieves zero false negatives (detecting all the 11 real world malware and 9 synthetic attacks) and zero false positives (no false alarms after a proper training process). In performance, KQguard introduces only a small overhead of about 100 microseconds per validation and up to 5% slowdown for resource-intensive application benchmarks due to heap object tracking.

8. REFERENCES

[1] Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J. 2005. Control-flow integrity. *Proceedings of the 12th ACM Conference on Computer and Communications Security*, Nov. 2005.

[2] Anselmi, D., Boscovich, R., Campana, T. J., Doerr, S., Lauricella, M., Petrovsky, O., Saade, T., Stewart, H. 2011. Battling the Rustock Threat. *Microsoft Security Intelligence Report, Special Edition*, January 2010 through May 2011.

[3] Baliga, A., Ganapathy, V., and Iftode, L. 2008. Automatic inference and enforcement of kernel data structure invariants. In *ACSAC '08: Proceedings of the 2008 Annual Computer Security Applications Conference*, pages 77–86. IEEE Computer Society.

[4] Boldewin, F. 2007. Peacomm.C - Cracking the nutshell. *Anti Rootkit*, September 2007. http://www.antirootkit.com/articles/eye-of-the-storm-worm/Peacomm-C-Cracking-the-nutshell.html.

[5] Brumley, D. 1999. Invisible intruders: rootkits in practice. *;login:*, 24, Sept. 1999.

[6] Castro, M., Costa, M., Harris, T. 2006. Securing Software by Enforcing Data-flow Integrity. In *Proceedings of OSDI'06*.

[7] Chiang, K., Lloyd, L. 2007. A Case Study of the Rustock Rootkit and Spam Bot. *Proceedings of the First Workshop on Hot Topics in Understanding Botnets (HotBots'07)*, April 2007.

[8] Decker, A., Sancho, D., Kharouni, L., Goncharov, M., and McArdle, R. 2009. Pushdo/Cutwail: A Study Of The Pushdo/Cutwail Botnet. *Trend Micro Technical Report*, May 2009.

[9] Giuliani, M. ZeroAccess – an advanced kernel mode rootkit, rev 1.2. www.prevxresearch.com/zeroaccess_analysis.pdf

[10] Hayes, B. 2010. Who Goes There? An Introduction to On-Access Virus Scanning, Part One. *Symantec Connect Community*. http://www.symantec.com/connect/articles/who-goes-there-introduction-access-virus-scanning-part-one

[11] Kapoor, A. and Mathur, R. 2011. Predicting the future of stealth attacks. *Virus Bulletin* 2011, Barcelona.

[12] Kaspersky Lab. The Mystery of Duqu: Part One. http://www.securelist.com/en/blog/208193182/The_Mystery_of_Duqu_Part_One

[13] Kaspersky Lab. The Mystery of Duqu: Part Five. http://www.securelist.com/en/blog/606/The_Mystery_of_Duqu_Part_Five

[14] Kil, C., Sezer, E., Azab, A., Ning, P., and Zhang, X. 2009. Remote attestation to dynamic system properties: Towards providing complete system integrity evidence. *Proceedings of the 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (DSN'09), Lisbon, Portugal.

[15] Kwiatek, L. and Litawa, S. 2008. Yet another Rustock analysis... *Virus Bulletin*, August 2008.

[16] Li, J., Wang, Z., Bletsch, T., Srinivasan, D., Grace, M., and Jiang, X. 2011. Comprehensive and Efficient Protection of Kernel Control Data. *IEEE Transactions on Information Forensics and Security*, 6(2), June 2011.

[17] Microsoft. Using Timer Objects. http://msdn.microsoft.com/en-us/library/ff565561.aspx.

[18] Microsoft. Checked Build of Windows. http://msdn.microsoft.com/en-us/library/windows/hardware/ff543457%28v=vs.85%29.aspx

[19] Necula, G. C., McPeak, S., Rahul, S. P. and Weimer, W. 2002. CIL: Intermediate language and tools for analysis and transformation of C programs. *Proceedings of Conference on Compiler Construction (CC)*, Grenoble, France, Apr. 2002.

[20] OffensiveComputing. Storm Worm Process Injection from the Windows Kernel. http://offensivecomputing.net/papers/storm-3-9-2008.pdf

[21] Petroni, N., Fraser, T., Molina, J., Arbaugh, W. A. 2004. Copilot—a coprocessor-based kernel runtime integrity monitor. *Proceedings of the 13th USENIX Security Symposium*, San Diego, CA, Aug. 2004.

[22] Petroni, N. and Hicks, M. 2007. Automated detection of persistent kernel control-flow attacks. *Proceedings of ACM Conference on Computer and Communications Security (CCS'07)*.

[23] Prakash, C. 2008. What makes the Rustocks tick! *Proceedings of the 11th Association of anti-Virus Asia Researchers International Conference (AVAR'08)*, New Delhi, India. http://www.sunbeltsecurity.com/dl/WhatMakesRustocksTick.pdf

[24] Riley, R., Jiang, X., and Xu, D. 2008. Guest-transparent prevention of kernel rootkits with VMM-Based memory shadowing. *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection (RAID'08)*.

[25] Seshadri, A., Luk, M., Qu, N., and Perrig, A. 2007. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. *Proceedings of ACM Symposium on Operating Systems Principles (SOSP'07)*.

[26] Smalley, S., Vance, C., and Salamon, W. 2002. Implementing SELinux as a Linux Security Module. In *Technical Report. NSA*, May 2002.

[27] Solar Designer. Bugtraq: Getting around non-executable stack (and fix). Website. http://seclists.org/bugtraq/1997/Aug/63, accessed March 2011.

[28] Super PI. http://www.superpi.net/

[29] Symantec Connect Community. W32.Duqu: The Precursor to the Next Stuxnet. Oct. 2011. http://www.symantec.com/connect/w32_duqu_precursor_next_stuxnet

[30] Tran, M., Etheridge, M., Bletsch, T., Jiang, X., Freeh, V. W., and Ning, P. 2011. On the Expressiveness of Return-into-libc Attacks. *Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection (RAID 2011)*, Menlo Park, California, September 2011.

[31] Wang, Z., Jiang, X., Cui, W., and Ning, P. 2009. Countering kernel rootkits with lightweight hook protection. *Proceedings of ACM Conference on Computer and Communications Security (CCS '09)*.

[32] Wei, J., and Pu, C. 2012. Towards a General Defense against Kernel Queue Hooking Attacks. *Elsevier Journal of Com-*

puters & Security, Volume 31, Issue 2, pp. 176-191, March 2012. doi:10.1016/j.cose.2011.12.007.

[33] Windows Research Kernel v1.2. https://www.facultyresourcecenter.com/curriculum/pfv.aspx?ID=7366&c1=en-us&c2=0

9. APPENDIX

Table 9: A List of Automatically-Detected KQs for the WRK

K-Queue Name	Data Structure Type	Queue header name (global variable)	# param. tainted (static)	Name of Request Insertion Function(s)
I/O timer queue	IO_TIMER	IopTimerQueueHead	2	IoInitializeTimer
File system registration change notification queue	NOTIFICATION_PACKET	IopFsNotifyChangeQueueHead	0	IoRegisterFsRegistrationChange, IoUnregisterFsRegistrationChange
Process creation/deletion notification queue	Array of EX_CALLBACK	PspCreateProcessNotifyRoutine	0	PsSetCreateProcessNotifyRoutine
Driver reinitialize routine queue	REINIT_PACKET	IopDriverReinitializeQueueHead	3	IoRegisterDriverReinitialization
Boot driver reinitialize routine queue	REINIT_PACKET	IopBootDriverReinitializeQueueHead	3	IoRegisterBootDriverReinitialization
Thread Creation/Deletion notification queue	Array of EX_CALLBACK	PspCreateThreadNotifyRoutine	0	PsSetCreateThreadNotifyRoutine, PsRemoveCreateThreadNotifyRoutine
RegistryCallback queue	Array of EX_CALLBACK	CmpCallBackVector	1	CmRegisterCallback, CmUnRegisterCallback
Load image notification queue	Array of EX_CALLBACK	PspLoadImageNotifyRoutine	0	PsSetLoadImageNotifyRoutine, PsRemoveLoadImageNotifyRoutine
Bug check reason callback queue	KBUGCHECK_REASON_CALLBACK_RECORD	KeBugCheckReasonCallbackListHead	1	KeRegisterBugCheckReasonCallback
Callback object queue	CALLBACK_REGISTRATION	ExpInitializeCallback	1	ExRegisterCallback, ExUnregisterCallback
Waiting IRP queue	WAITING_IRP	Queue head not a global variable, instead one field of any opportunistic lock object	2	FsRtlWaitOnIrp
IRP waiting lock queue	WAITING_LOCK	FsRtlFileLockCancelCollideList, FsRtlWaitingLockLookasideList	2	FsRtlPrivateLock
Firmware table providers queue	SYSTEM_FIRMWARE_TABLE_HANDLER_NODE	ExpFirmwareTableProviderListHead	1	ExpRegisterFirmwareTableInformationHandler
BugCheckCallback routine queue	KBUGCHECK_CALLBACK_RECORD	KeBugCheckCallbackListHead	2	KeRegisterBugCheckCallback
Deferred write queue	DEFERRED_WRITE	CcDeferredWrites	2	CcDeferWrite
Per-stream context queue	FSRTL_PER_STREAM_CONTEXT	FSRTL_ADVANCED_FCB_HEADER structure with a file stream, no global header	1	FsRtlInsertPerStreamContext
Change directory notification queue	NOTIFY_CHANGE	A notify list associated with a file system, no global header	3	FsRtlNotifyFilterChangeDirectory
System worker thread queue	WORK_QUEUE_ITEM	ExWorkerQueue[3]	1	ExQueueWorkItem
APC queue	APC	Not found yet	5,1,3	KeInitializeApc
DPC queue	DPC	Not found yet	3	KiInitializeDpc

Table 10: A List of 22 Automatically-Detected KQs for Linux Kernel 2.4.32

K-Queue Name	Data Structure Type	Queue Header Name (global variable)	# param. tainted (static)	Name of Dispatch Function(s)	Name of Request Insertion Function(s)
ACPI bus driver queue	struct acpi_driver (struct list_head)	acpi_bus_drivers	0	acpi_bus_find_driver	acpi_bus_register_driver, acpi_bus_unregister_driver
Task queue	struct tq_struct (struct list_head)	No single global queue header; device drivers can declare their own queue header.	1	run_task_queue	queue_task
Soft timer queue	struct timer_list (struct list_head)	tvecs[]	1	run_timer_list	add_timer, del_timer, mod_timer
Tasklet queue	struct tasklet_struct (next * to self)	tasklet_vec[], tasklet_hi_vec[]	1	tasklet_action, tasklet_hi_action	tasklet_schedule, tasklet_hi_schedule
Packet type queue	struct packet_type (next * to self)	ptype_all, ptype_base[],	1	dev_queue_xmit_nit, netif_receive_skb	dev_add_pack, dev_remove_pack
IRQ action queue	struct irqaction (next * to self)	irq_desc[].action	1	handle_IRQ_event	request_irq, free_irq
Binary formats handler queue	struct linux_binfmt (next * to self)	formats	0	sys_uselib, search_binary_handler	register_binfmt, unregister_binfmt
PC Card client driver queue	driver_info_t (next * to self)	root_driver	0	bind_request	register_pccard_driver, unregister_pccard_driver
IP socket interfaces queue	struct inet_protosw (struct list_head)	inetsw[]	0	inet_create	inet_register_protosw, inet_unregister_protosw
Power management notification queue	struct pm_dev (struct list_head)	pm_devs	1	pm_send_all	pm_register, pm_unregister, __pm_unregister, pm_unregister_all
PCI driver queue	struct pci_driver (struct list_head)	pci_drivers	1	pci_announce_device_to_drivers	pci_register_driver, pci_unregister_driver
Dead destination cache queue	struct dst_entry (next * to self)	dst_garbage_list	1	dst_run_gc	dst_free
INET protocol handlers queue	struct inet_protocol (next * to self)	inet_protos[]	0	ip_local_deliver_finish, icmp_unreach,	inet_add_protocol, inet_del_protocol
Console drivers queue	struct console (next * to self)	console_drivers	1; 0; 1;	call_console_drivers, console_unblank, tty_open	register_console, unregister_console
Panic notifiers queue	struct notifier_block (next * to self)	panic_notifier_list	1	notifier_call_chain	notifier_chain_register, notifier_chain_unregister
Reboot notifiers queue	struct notifier_block (next * to self)	reboot_notifier_list	1	notifier_call_chain	notifier_chain_register, notifier_chain_unregister
Network notifiers queue	struct notifier_block (next * to self)	netdev_chain	1	notifier_call_chain	notifier_chain_register, notifier_chain_unregister
Network link notifiers queue	struct notifier_block (next * to self)	netlink_chain	1	notifier_call_chain	notifier_chain_register, notifier_chain_unregister
IPv4 Interface address notifiers queue	struct notifier_block (next * to self)	inetaddr_chain	1	notifier_call_chain	notifier_chain_register, notifier_chain_unregister
IPv6 Interface address notifiers queue	struct notifier_block (next * to self)	inet6addr_chain	1	notifier_call_chain	notifier_chain_register, notifier_chain_unregister
ADB device notifiers queue	struct notifier_block (next * to self)	adb_client_list	1	notifier_call_chain	notifier_chain_register, notifier_chain_unregister
HCI notifiers queue	struct notifier_block (next * to self)	hci_notifier	1	notifier_call_chain	notifier_chain_register, notifier_chain_unregister