KQguard: Binary-Centric Defense against Kernel Queue Injection Attacks

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

[1] Florida International University, Miami, Florida, USA
{weijp,fzhu001}@cs.fiu.edu
[2] Georgia Institute of Technology, Atlanta, Georgia, USA
calton@cc.gatech.edu

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 run malicious logic. Current defense mechanisms for kernel code and data integrity have difficulties with kernel queue injection (KQI) 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 implement KQguard on the Windows Research Kernel (WRK) and Linux and extensive experimental evaluation shows that KQguard is efficient (up to ~5% overhead) and effective (capable of achieving zero false positives against representative benign workloads after appropriate training and very low false negatives against 125 real-world malware and nine synthetic attacks). KQguard protects 20 KQs in WRK, can accommodate new device drivers, and through dynamic analysis of binary code 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 callback 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 (as input parameters of the event handler function). When an event happens, a kernel thread invokes the specified callback function 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 malware such as rootkits (Section 5.1 and [5, 24]). For example, the Pushdo spam bot has misused the Registry Operation Notification

J. Crampton, S. Jajodia, and K. Mayes (Eds.): ESORICS 2013, LNCS 8134, pp. 755–774, 2013.

Queue of the Windows kernel to monitor, block, or modify legitimate registry operations [10]. This paper includes 125 examples of real-world malware misusing KQs demonstrating these serious current exploits, and nine additional synthetic potential misuses for illustration of future dangers.

The above-mentioned kernel-level malware misuses the KQs to execute malicious logic, by inserting their own requests into the KQs. This kind of manipulation is called *KQ Injection* or simply KQI. Although KQI appears similar to Direct Kernel Object Manipulation (DKOM) [6] or Kernel Object Hooking (KOH) [13], it is more expressive thus powerful than the other two. While DKOM attacks only tamper with non-control data and KOH attacks only tamper with control data, KQI attacks are capable of doing both because the attacker can supply both control data (i.e., the callback function) and/or non-control data (i.e., the parameters). Moreover, KQI is stealthier than DKOM or KOH in terms of invasiveness: DKOM or KOH attacks *modify* legitimate kernel objects so they are invasive, while KQI attacks just *insert* new elements into KQs and do not have to modify any legitimate kernel objects.

Several seminal defenses have been proposed for DKOM and KOH attacks [1, 3, 26, 36]. Unfortunately, they are not directly applicable to KQI attacks either because of their own limitations or the uniqueness of KQIs. For example, CFI [1] is a classic defense against control data attacks, but it cannot address non-control data attacks launched via KQ injection (Section 2.2 provides a concrete example in WRK). Gibraltar [3] infers and enforces *invariant* properties of kernel data structures, so it seems able to cover KQs as one type of kernel data structure. Unfortunately, Gibraltar relies on periodic snapshots of the kernel memory, which makes it possible for a *transient* malicious KQ request to evade detection. Petroni [26] advocates detecting DKOM by checking the integrity of kernel data structures against specifications, however, the specifications are elaborate and need to be manually written by domain experts. Finally, KQI attacks inject *malicious* kernel data, which makes HookSafe [36] an inadequate solution because the latter can only protect the integrity of *legitimate* kernel data. Therefore, new solutions are needed to defend against KQI attacks.

Inspired by the above research, our KQ defense endorses the general idea of using data structure invariants. However, we address the limitation of existing approaches so that our KQ integrity checking covers both persistent and transient attacks. More specifically, our defense intercepts and checks the validity of *every* KQ request to ensure the execution of legitimate event handlers only, by filtering out all untrusted callback requests. In [37], we develop a KQ defense for Linux (called PLCP) that employs static source code analysis to automatically derive specifications of legitimate KQ requests. However, the reliance on source code limits the practical applicability of PLCP in systems such as Windows in which there are a large number of third-party, closed source device drivers that need KQs for their normal operation.

Therefore, in this paper, we build KQguard, an effective defense against KQI attacks that can support closed source device drivers. Specifically, we make the following contributions: (1) we introduce the KQguad mechanism that can distinguish attack

KQ requests from legitimate KQ event handlers, (2) we employ dynamic analysis of the binary code to automatically generate specifications of legitimate KQ requests (called EH-Signatures) in closed source device drivers, (3) we build a static analysis tool that can automatically identify KQs from the source code of a given kernel, (4) we implement the KQguard in WRK [39] and the Linux kernel, (5) our extensive evaluation of KQguard on WRK shows its effectiveness against KQ exploits (125 real-world malware samples and nine synthetic rootkits), detecting all except two of the attacks (very low false negative rate). With appropriate training, we eliminated all false alarms from KQguard for representative workloads. For resource intensive benchmarks, KQguard carries a small performance overhead of up to about 5%.

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, 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 Injection

2.1 Importance of KQ Injection 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 requester (e.g., a device driver) 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 (Fig. 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.

Kernel-level rootkits exploit the KQ callback mechanism to execute malicious logic by inserting their own request into a KQ (e.g., by supplying malicious callback function or data in step 1 of Fig. 1). This kind of manipulation, called a *KQ injection* attack, only uses legitimate kernel interface and it does not change legitimate kernel code or statically allocated data structures such as global variables. Therefore, syntactically a KQ injection 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 Windows registry. Using it in KQ injection attack, Pushdo [10] can monitor, block, or modify legitimate registry operations.

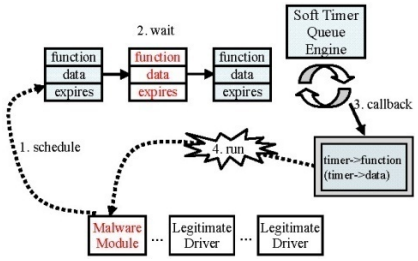

Fig. 1. Life cycle of a timer request in Linux

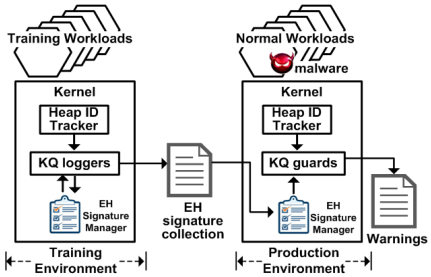

Fig. 2. Overall Architecture of KQguard

Several KQ injection attacks by real world malware have been documented (Table 1 in Section 5.1). Specifically, malware has misused KQs to hide better against discovery [2,18], to carry out covert operations [9,10,27], and to attack security products directly [4]. Further details can be found in our technical report [38]. Undoubtedly, KQ injection attacks represent a clear and present danger to current OS kernels.

2.2 KQ Injection Attack Model

The KQ injection malware listed in Table 1 (Section 5.1) 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 the kernel space, callback-into-malware attacks can be detected by runtime kernel code integrity checkers such as SecVisor [29]. Therefore, they are considered the basic level of attack.

Unfortunately, a more sophisticated level of KQ injection attacks, called *callback-into-libc* (in analogy to return-into-libc [32, 35]), create a malicious callback request containing a legitimate callback function but malicious input parameters. When activated, the legitimate callback function may carry out *unintended* actions that are beneficial to the attacker. For example, one legitimate callback function in the asynchronous procedure call (APC) queue of the WRK is `PsExitSpecialApc`, which can cause the currently executing thread to terminate with an exit status code that is specified in the "NormalContext" parameter field of the APC request structure. Therefore, hypothetically an attacker can inject an APC request with `PsExitSpecialApc` as the callback function to force a thread to terminate with a chosen exit status code (set in the "NormalContext" field). This kind of Callback-into-libc attack can be used to shutdown an anti-virus program but make the termination appear normal to an Intrusion Detection System, by setting a misleading exit status code.

Callback-into-libc KQ injection attacks represent an interesting challenge, since they allow an attacker to execute malicious logic without injecting his own code, and the above example shows that such attacks can target non-control data (e.g., the exit status code of a thread). Therefore, they cannot be defeated by approaches that focus on control data (e.g., CFI [1]).

The design of KQguard in Section 3 shows how we can detect both callback-into-malware and callback-into-libc KQ injection attacks.

2.3 Design Requirements of KQ Defense

An effective KQ defense should satisfy four requirements: efficiency, effectiveness, extensibility, and inclusiveness. In this section, we outline the reasons KQguard satisfies these requirements. Some previous techniques may solve specific problems but have difficulties with satisfying all four requirements. We defer a discussion of related work to Section 6.

Efficiency: It is important for KQ defenses to minimize their overhead; KQguard is designed to protect KQs with low overhead, including the time-sensitive ones. *Effectiveness*: KQ defenses should detect all the KQ injection 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. *Extensibility*: Due to the rapid proliferation of new devices, it is important for KQ defenses to extend their coverage to new device drivers; the KQguard design isolates the knowledge on legitimate event handlers into a table (EH-Signature collection) that 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 implementation is described in Section 4.

3.1 Architecture Overview and Assumptions

The main idea of KQguard is to differentiate legitimate KQ event handlers from malicious KQ injection attacks based on characteristics of *known-good* event handlers. For simplicity of discussion, we call such characteristics *Callback-Signatures*. A Callback-Signature is an effective representation of a KQ event handler (or a KQ request) for checking. One special type of Callback-Signatures is those of the legitimate KQ event handlers, and we call them *EH-Signatures*.

How to specify or discover the EH-Signatures is a practical challenge in the design of KQguard. Since legitimate KQ requests are originated from legitimate kernel or device drivers, in order to specify EH-Signatures we need to study the behavior of the core kernel and legitimate drivers. In an ideal kernel development environment, one could imagine annotating the entire kernel and all device driver code to make KQ requests 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.

Unfortunately, this is not practical because many third-party closed source device drivers are unlikely to share their source code.

Therefore, our design decision is to apply dynamic binary code analysis to automate the process of obtaining a specialized EH-Signature collection that fits the configuration and usage of each system. Specifically, our design uses the architecture shown in Fig. 2. We extend the kernel in a dedicated training environment to log (collect) EH-Signatures of KQ requests that the kernel encounters during the execution of legitimate device drivers. Then we extend the kernel in a production environment to use such learned EH-Signatures to guard against KQ injection attacks, which can be launched by malware installed in the production environment.

By employing dynamic analysis, our design does not require source code of the device drivers, thus it satisfies the inclusiveness requirement. Moreover, by having two kinds of environments, we decouple the collection and the use of EH-Signatures, which allows future legitimate drivers to be supported by KQguard: we can run the new driver in the training environment to collect its EH-Signatures and then add the new EH-Signatures into the signature collection used by the production environment. By using this method, our design satisfies the extensibility requirement.

In order to guarantee that EH-Signatures learned from the training environment is applicable to the production environment, we assume that the training environment and the production environment run the same OS and set of legitimate device drivers.

In order to guarantee that all the Callback-Signatures learned from the training environment represent legitimate KQ requests, we assume that any device driver that is run in the training environment is benign. This assumption may not hold on a consumer system because a normal user may not have the knowledge and capability to tell whether a new driver is benign or not. Therefore, we expect that KQguard is used in a strictly controlled environment (such as military and government) where a knowledgeable system administrator ensures that only benign device drivers are installed in the training environment, by applying standard security practices.

As is typical of any dynamic analysis approach, we assume that a representative and comprehensive workload is available during training to trigger all the legitimate KQ event handlers. Because some legitimate KQ requests may be made only under certain conditions, the workload must be comprehensive so that such KQ requests can be triggered and thus logged. Otherwise, KQguard may raise false alarms.

3.2 Building the EH-Signature Collection

In order to collect EH-Signatures in a training environment, we first instrument the kernel with KQ request logging capability and then run comprehensive workloads to trigger legitimate KQ requests.

3.2.1 Instrumentation of the Kernel to Log EH-Signatures

To collect EH-Signatures, we instrument all places in the kernel where KQ request information is available. Specifically, we extend kernel functions that initialize, insert, or dispatch KQ requests. We extend these functions with a KQ request logging utility,

which generates and logs Callback-Signatures from every "raw" KQ request (i.e., with absolute addresses) submitted by the legitimate kernel and device drivers. The details of Callback-Signature generation are non-trivial and deferred to Section 3.5.

In general, the information contained in EH-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. Section 3.6 describes our non-trivial search for all the locations of these simple changes, in which we employ static source code analysis on the entire kernel. The extensions are applied to the kernel at source code level. The instrumented kernel is then rebuilt for the EH-Signature collection process.

3.2.2 Dynamic Profiling to Collect EH-Signatures

In this step, we run a representative set of benchmark applications using a comprehensive workload on top of the instrumented kernel. During this phase, the kernel extensions described in Section 3.2.1 are triggered by every KQ request.

To avoid false negatives in KQ defense, the training is performed in a clean environment to ensure no malware Callback-Signatures are included. To avoid false positives, the training workload needs to be comprehensive enough to trigger all of the legitimate KQ requests. Our evaluation (Section 5.3) 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.

3.3 Validation Using EH-Signature Collection

As shown in the "Production Environment" part of Fig. 2, we modify the dispatcher of every identified KQ to introduce a KQ guard that checks the legitimacy of a pending KQ request before the dispatcher invokes the callback function. To perform the check, the KQ guard first builds the Callback-Signature from a pending request (detailed in Section 3.5), and then matches the Callback-Signature against the EH-Signature Collection. 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 details of signature matching are discussed in Section 3.4.

To reduce performance overhead, we cache the results of KQ validation so as to avoid repeatedly checking a KQ request if its Callback-Signature has not changed since the last time it is checked. Specifically, we maintain cryptographic hashes of the "raw" KQ requests (identified by memory location) that pass the validation, so that when the same KQ request (at the recorded memory location) is to be checked again, we recalculate the cryptographic hash and compare it with the stored one. Our profiling study confirms that a significant fraction (~90%) of KQ validation is redundant because the same KQ requests are repeatedly enqueued, dispatched, dequeued, and enqueued again. Therefore, caching the validation results for such repeated KQ requests can reduce performance overhead of KQ defense.

3.4 Specification of the Callback-Signatures

A critical design issue of KQguard is the determination of the set of characteristics in the Callback-Signatures: it must precisely identify the same KQ requests in the training and production environments. On one hand, the set must not include characteristics that can vary between the two environments (e.g., the expiration time in a soft timer request) because otherwise even the same legitimate KQ requests would appear different (false positives); on the other hand, the set must include all the *invariant* characteristics between the two environments because otherwise a malicious KQ request that differs from a legitimate request only in the missing characteristics would also pass the check, resulting in false negatives. For example, the malicious KQ request in Fig. 3.b is allowed by a KQ guard that only checks the shaded fields, although it causes a malicious function `bar_two` to be invoked; and the malicious KQ request achieves this by tampering with the "action" field of structure `se` that is not covered by the Callback-Signature. Here when the KQ request is dispatched, `foo` is invoked with `qe.data` as its parameter.

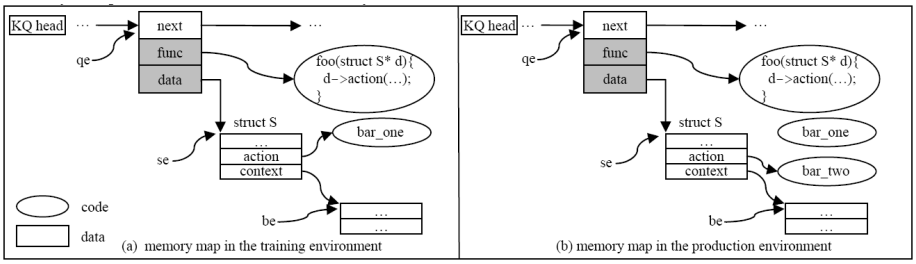

Fig. 3. Illustration of a False Negative Caused by a Callback-Signature that Only Includes the Shaded Fields. The two KQ requests have different executions (i.e., `bar_one` vs `bar_two`), but their Callback-Signatures are the same. Here `bar_two` is a malicious function.

In order to minimize false negatives such as the one demonstrated in Fig. 3, one could include more characteristics (e.g., `se.action`) into the Callback-Signatures. However, there are some challenges in doing that with closed source device drivers. Specifically, in order to realize that `se.action` is important, one can get hints from how `foo` works, but without source code, it is non-trivial to figure out that `foo` invokes `se.action`. Another possibility is to use the type information of `se` (e.g., `struct S`) to know that its "action" field is a function pointer and such information can be derived from the type of KQ request data fields (e.g., `qe.data`); unfortunately, this is often not possible because the data fields of KQ requests are often generic pointers (i.e., `void *`); in that case, one cannot figure out the type of `se` easily if it resides in a closed source device driver. Therefore, in order to support closed source device drivers, our KQ defense assumes that:

Kernel data reachable from KQ requests (e.g., `se.action`) can be identified and it has integrity in both the training and the production environments (i.e., changing of this field from `bar_one` to `bar_two` is prohibited by some other security measures).

To avoid "reinventing the wheel", we note that techniques such as KOP [7] can correctly locate kernel data such as `se.action` despite the existence of generic pointers, and techniques such as HookSafe [36] can prevent malware from tampering with invariant function pointers in legitimate kernel data structures, such as `se.action`. Moreover, both KOP and HookSafe can be used to cover even "deeper" kernel data such as `be` in Fig. 3. Note that the inclusion of `qe.data` in the Callback-Signature is very critical because it ensures that if `qe` can pass the check performed by KQguard, `se` is a legitimate kernel data structure, and thus its "action" field can be protected by HookSafe (HookSafe is designed to protect only legitimate kernel data structures).

Note that HookSafe cannot be an alternative defense against KQ injection attacks from the top level (e.g., by ensuring that "func" and "data" fields in Fig. 3 are not tampered with) for two reasons. First, not all top-level KQ request data structures are legitimate because malware can allocate and insert its own KQ request data structure. Second, not all top-level legitimate KQ request data structures are invariant (i.e., their values do not change) but HookSafe can only protect invariant kernel data. We have observed multiple cases in the APC queue of the WRK in which top-level legitimate KQ requests change their values during normal execution. For example, `IopfCompleteRequest` (in WRK\base\ntos\io\iomgr\iosubs.c) inserts an APC request with callback function `IopCompleteRequest` (in WRK\base\ntos\io\iomgr \internal.c); when this APC request is dispatched (i.e., `IopCompleteRequest` is invoked), its callback function field is changed to `IopUserCompletion` before it is inserted back to the APC queue.

To summarize the above discussion, (1) we need to support closed source device drivers, (2) we need a way to defend against KQ injection attacks from the top level, and (3) techniques are available to guard deeper kernel data reachable from KQ requests. Based on these three observations, in this paper we choose a Callback-Signature format that focuses on KQ request level (the top level) characteristics: (callback_function, callback_parameters, insertion_path, allocation). Here callback_function is the callback function pointer stored in a KQ request, callback_parameters represents the relevant parameters stored in it, insertion_path represents how the KQ request is inserted (by which driver? along which code path?), and allocation represents how its memory is allocated (global, heap, or stack? by which driver?).

Each characteristic in our Callback-Signature is important for effective KQ guarding. callback_function is used to protect the kernel against callback-into-malware attacks, and both callback_function and callback_parameters are used to protect the kernel against callback-into-libc attacks (Section 2.2). Furthermore, insertion_path and allocation provide the context of the KQ request and thus can also be very useful. For example, if KQguard only checks callback_function and callback_parameters, malware can insert an *existing* and *legitimate* KQ request object LKQ if it can somehow benefit from the dispatching of LKQ (e.g., resetting a watchdog timer).

To ensure that the signature matching of a KQ request observed during the production use and one observed during the training can guarantee the same *execution*, we need to make sure that the code and static data of the core kernel and legitimate device drivers have integrity in the production environment. We also need to ensure

that malware cannot directly attack KQ guards, including their code and the EH-Signature collection. We can leverage a hypervisor (e.g., Xen) to satisfy the above requirements. The idea is to run the modified kernel (with KQ guards) on top of a hypervisor and extend the shadow-based memory management of the hypervisor to write-protect code and static data of the modified kernel [37]. Note that this protection covers KQ guards and the EH-signature collection because they are part of the modified kernel.

3.5 Generation of Callback-Signatures from KQ Requests

In both EH-Signature collection (Section 3.2) and KQ request validation (Section 3.3), Callback-Signatures need to be derived from raw KQ requests. This is called Callback-Signature generation and we discuss the details in this subsection.

3.5.1 Motivation for Delinking

As we discuss in Section 3.4, a Callback-Signature is a tuple (callback_function, callback_parameters, insertion_path, allocation). Since callback_function and callback_parameters correspond to fields in KQ requests (e.g., the "func" field of `qe` in Fig. 3), it seems that we can simply copy the value of those fields into a Callback-Signature. However, when a Callback-Signature contains a memory reference (e.g., a parameter that points to a heap object), we have to overcome one challenge: namely, what the KQ loggers and the KQ guards can directly observe is an absolute memory address; however, the absolute addresses of the same variable or function can be different in the training and production environments, for example, when they are inside a device driver that is loaded at different starting addresses in the two environments. Therefore, if we use absolute addresses in the Callback-Signatures, there will not be a match for the same callback function, which results in false positives.

In order to resolve this issue, we raise the level of abstraction for memory references in the Callback Signatures so that variations at the absolute address level can be tolerated. For example, we translate a callback function pointer (absolute address) into a unique module ID, plus the offset relative to the starting address of its containing module (usually a device driver, and we treat the core kernel as a special module). Under the assumption that the kernel maintains a uniform mapping of module location to module ID, the pair (module ID, offset) becomes an invariant representation of the callback function pointer independent of where the module is loaded. This kind of translation is called *delinking*.

3.5.2 Details of Delinking

KQguard delinks memory references (i.e., pointers) in different ways depending on the allocation type of the target memory. As Fig. 4 shows, there can be three types of allocations: global variable, heap variable, and local variable.

The pointer to a global variable is translated into (module ID, offset), in the same way as the callback function pointer (Section 3.5.1). There can be two kinds of global variables depending on whether they reside in a device driver inside the kernel or in a

user-level library (e.g., a DLL on Windows). We care about user-level global variables because some KQ parameters reference user-level memory (e.g., the APC queue on Windows). We regard device drivers and user-level libraries uniformly as modules and we modify the appropriate kernel functions to keep track of their address ranges when they are loaded (e.g., `PspCreateThread` for DLLs).

The pointer to a heap object is translated into a call stack that corresponds to the code path that originates from a requester (e.g., a device driver) and ends in the allocation of the heap object. We use a call stack rather than the immediate return address because the immediate return address may not be in the requester's address space (i.e., it may be in some wrapper function for the heap allocation function and the requester can call a function at the top of the call chain to allocate a heap object). Since most kernels do not maintain the request call stack for allocated heap objects, we instrument their heap allocator functions to collect such information, and the instrumentation is called Heap ID Tracker in Fig. 2. Specifically, the Heap ID Tracker traverses the call stack frames backwards until it reaches a return address that falls within the code section of a device driver or it reaches the top of the stack; if no device driver is found during the traversal, the core kernel is used as the requester; all return addresses encountered during this traversal are part of the call stack, and each of them is translated into a (module ID, offset) pair, in the same way as the callback function pointer discussed in Section 3.5.1. Similar to global variables, our delinking supports two types of heap objects: kernel-level and user-level.

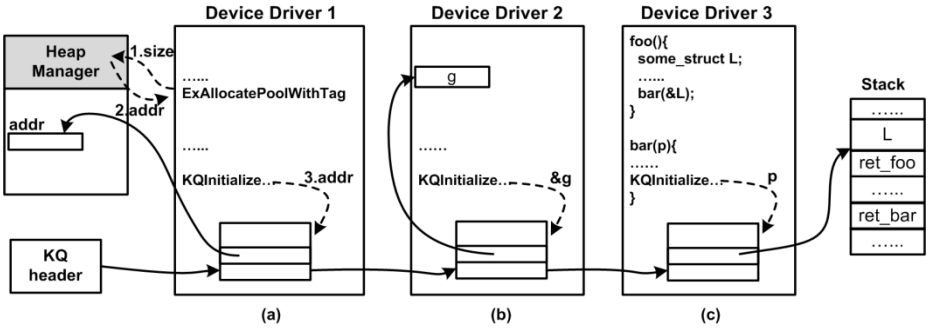

Fig. 4. Illustration of Different Allocation Types of Pointers: (a) Heap Variable, (b) Global Variable, (c) Local Variable

The pointer to a local variable is translated into a pair (call_stack, l_offset). The call stack starts in a function where a KQ request is inspected (e.g., in a KQ insertion function), and it stops in the function that contains the local variable (e.g., `L` in function `foo` in Fig. 4.c).Each return address encountered during the traversal is translated into a (module ID, offset) pair. Finally, l_offset is the relative position of the local variable in its containing stack frame. For example, if [ebp-8] is used to represent the local variable, l_offset is 8. We have not observed any pointers to user-level local variables, so we do not cover the translation for pointers to user-level local variables.

Because the static type of a KQ request data (e.g., the "data" field of a soft timer request structure) is often a generic pointer (i.e., `void *`), we have to detect its actual type at runtime. Given the raw value of a piece of KQ request data, we run a series of tests to decide the suitable delinking for it if it is considered a pointer. First, we test whether the raw value falls within the address range of a loaded driver or a user-level library to decide whether it should be delinked as a pointer to a global variable. If the test fails, we test whether it falls within the address range of an allocated heap object to decide whether it should be delinked as a pointer to a heap variable. If this test also fails, we test whether it falls within the address ranges of the stack frames to see whether it should be delinked as a pointer to a local variable. If this test still fails, we determine the KQ data to be a non-pointer, and no delinking is performed.

3.6 Automated Detection of KQs

Since every KQ can be exploited by malware (part of the attack surface), we need to build the EH-Signatures for all of KQs. But before we can guard a KQ, we must first know its existence. Therefore, we design and implement 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, a KQ has a dispatcher that operates on the corresponding type.
- A KQ dispatcher usually contains a loop to act upon all or a subset of queue elements. For example, `pm_send_all` in Fig. 5 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.

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

Fig. 5. Details of the Power Management Notification Queue on Linux Kernel 2.4.32

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 Fig. 5). During the propagation, our tool

checks whether any tainted function pointer is invoked (e.g., `dev->callback` in `pm_send` in Fig. 5), and if that is the case, it reports a candidate KQ. Due to space constraints we omit further details, but the results (e.g., KQs found in WRK) are interesting and discussed in Section 4.

4 Implementations of KQguard

The KQguard design (Section 3) is implemented on the WRK and Linux (kernel version 3.5). Due to space constraint, we only present our implementation on the WRK, which consists of about 3,900 lines of C code and 2,003 lines of Objective Caml code.

Construction of Callback-Signatures in WRK. In order to collect the Callback-Signatures for the 20 KQs in the WRK, we instrument the kernel in two sets of functions. The first set of functions initialize, insert, or dispatch KQs and our instrumentation consists of 600 lines of C code. To support delinking of Callback-Signatures, we instrument the device driver loader function (`IopLoadDriver`) and the thread creation function (`PspCreateThread`), and we also instrument heap allocation or deallocation functions (`ExAllocatePoolWithTag`, `ExFreePool`, `NtAllocateVirtualMemory`, and `NtFreeVirtualMemory`) to keep track of the address ranges of allocated heap memory blocks and the call stack to the heap allocation function. Our instrumentation of the heap allocator / deallocator consists of 800 lines of C code.

Automated Detection of KQs for the WRK. We implement the KQ discovery algorithm (Section 3.6) based on static source code analysis, using the C Intermediate Language (CIL) [22]. 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), 20 KQs were detected (seven of them are mentioned in Table 1 and the rest can be found in [38]), and they include all the KQs that we are aware of, which suggests the usefulness of our KQ discovery algorithm. However, whether these 20 KQs cover *all* KQs in the WRK is an interesting and open question.

Callback-Signature Collection Management. We developed a set of utility functions to manage the Callback-Signatures, including the EH-Signatures. These functions support the generation, comparison, insertion, and search of Callback-Signatures. They are implemented in 2,200 lines of C code.

Validation of Callback-Signature in WRK. We instrument the dispatcher of every identified KQ in the WRK in the production environment so that the dispatcher checks the legitimacy of a pending KQ request before invoking the callback function (Section 3.3). Our instrumentation consists of about 300 lines of C code.

5 Evaluation of KQguard in WRK

Due to space constraints, we only report the evaluation results of the WRK implementation of KQguard in this section. 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 injection attacks. By efficiency we mean the overhead introduced by KQguard. In both the training and the production systems used in our evaluation, the hardware is a 2.4 GHz Intel Xeon 8-Core server with 16 GB of RAM, and the operating system is Windows Server 2003 Service Pack 1 running the WRK.

5.1 Real-World KQ Injection Attacks

We start our evaluation of KQguard effectiveness by testing our WRK implementation (Section 4) against real-world KQ injection attacks in Windows OS. Since malware technology keeps advancing, we focus on the most recent and the most influential malware samples that represent the state of the art. Specifically, we chose 125 malware samples from the top 20 malware families [40] and the top 10 botnet families [41]. These samples are known to have KQ injection behaviors.

Overall, our test confirmed that 98 samples inject the APC queue, 34 samples inject the DPC queue, 32 samples inject the load image notification queue, 20 samples inject the process creation/deletion notification queue, four samples inject the file system registration change queue, four samples inject the registry operation notification queue, and two samples inject the system worker thread queue.

Table 1 reports the results of 10 representative spam bot samples. We started with malware with reported KQ injection attacks, which are marked with a "√" with citation. We were able to confirm some of these attacks, shaded in gray. The rows with shaded "√" without citations are confirmed new KQ injection attacks that have not been reported by other sources. For example, Rustock.J injects an APC request with a 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 APC request raises an alarm because it does not match any of the EH-Signatures we have collected.

For all the malware that we were able to activate (the Rustock.C sample failed to run in our test environment), we confirmed the reported KQ injection attacks, except for the Duqu attack on load image notification queue and Storm on the APC queue.

Table 1. Known KQ Injection Attacks in Representative Malware

KQ / Malware	Timer/ DPC	Worker Thread	Load Image	Create Process	APC	FsRegistration Change	RegistryOp Callback
Rustock.C	√ [2, 18]			√ [27]	√ [27]		
Rustock.J			√	√	√		
Pushdo	√			√ [10]	√	√ [10]	√ [10]
Storm	√		√ [4]		√ [23]		
Srizbi	√				√		
TDSS			√		√	√	
Duqu	√		√ [16]		√		
ZeroAccess	√	√ [11]			√ [11]		√
Koutodoor	√			√			
Pandex					√		
Mebroot	√						

The Rustock samples show that malware designers have significant ability and flexibility in injecting 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. Overall, our evaluation indicates that KQguard can have a low false negative rate because it detects all except two of the KQ injection attacks by 125 real-world malware samples.

5.2 Protection of All KQs

In addition to real world malware, we create synthetic KQ injection attacks for two reasons. First, nine KQs have maximum queue length of zero during the testing in Section 5.1, suggesting that malware is not actively targeting them for the moment; however, the Rustock evolution shows that malware writers may consider such KQs in the near future, so we should ensure that guards for such KQs work properly. Second, the malware analyzed in Section 5.1 belongs 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 KQ-guard for both kinds of attacks. Therefore, for completeness, we developed test Windows device drivers for each of the KQs that have not been called and we have confirmed that our KQ defense can detect all the test drivers, which suggests that our defense is effective against potential and future KQ injection attacks.

5.3 False Alarms

We have experimentally confirmed that it is possible to reduce the false positives of KQ guarding to zero. This is achievable when the training workload is comprehensive enough to produce the full EH-Signature collection.

We first collect EH-Signatures on a training machine with Internet access. We repeatedly log in, run a set of normal workload programs, and log off. In order to trigger all possible code paths that insert KQ requests, we actively do the above for fifteen hours. During this process, we gradually collect more and more EH-Signatures until the set does not grow. At the end of training, we collect 813 EH-Signatures. The set of workload programs include Notepad, Windows Explorer, WinSCP, Internet Explorer, 7-Zip, WordPad, IDA, OllyDbg, CFF Explorer, Sandboxie, and Python.

Next we feed the collected EH-Signatures into a production machine with KQ guarding and use that machine for normal workloads as well as the KQ injection malware evaluation and the performance overhead tests. During such uses, we observe zero false alarms. The normal workload programs include the ones mentioned above as well as others such as Firefox not used in training.

While the experimental result appears encouraging, we avoid making a claim that dynamic analysis can always achieve zero false positives. For example, the APC queue has 733 EH-Signatures, such EH-Signatures have 14 unique callback functions, and the most popular callback function is `IopCompleteRequest`, occurring in 603

EH-Signatures. While these 603 EH-Signatures share the same callback function, their insertion paths originate from 51 device drivers, two DLLs, and the core kernel, so the average number of EH-Signatures per requester (e.g., a device driver) is 11, and the largest number is 45 (from the driver ntfs.sys). This result implies that there can be potentially many code paths within a driver that can prepare and insert an APC request with the same callback function, which may or may not be triggered in our training. Moreover, there are in total 199 device drivers in our evaluation system, but our training only observes a subset of them (e.g., 51 in terms of `IopCompleteRequest`); so some legitimate KQ requests from the remaining drivers may be triggered by events such as inserting a USB device, which we have not tested yet. Fortunately, our experience suggests that it is possible to collect the set of EH-Signatures that fits the configuration and usage of a given system with enough training workloads.

5.4 Performance Overhead

We evaluate the performance overhead of KQguard in two steps: microbenchmarks and macrobenchmarks.

For the first step, we measure 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 four parts of a Callback-Signature. Heap object tracking affects every heap allocation and deallocation operation (e.g., `ExAllocatePoolWithTag` and `ExFreePool`). These heap operations are invoked at a global level, with overhead proportional to the overall system and application use of the heap. Specifically, we measure the total time spent in performing 1,000 KQguard validation checks for the DPC queue and the I/O timer queue, two of the most active KQs. The main result is that global heap object tracking during the experiment dominated the KQguard overhead. Specifically, 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). Therefore, the total CPU consumed by our KQguard validation for DPC queue and the supporting heap object tracking is 8,620.7 milliseconds (or about 3.4% of the total elapsed time). The measurements of the I/O timer queue (180 ms for validation, 11,807 ms for heap object tracking, and 345,825 ms total elapsed time) confirm the DPC queue results.

For the second step, Table 2 shows the results of five application level benchmarks that stress one or more system resources, including CPU, memory, disk, and network. Each workload is run multiple 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 copy. These elapsed time increases are consistent with the microbenchmark measurements, with higher or lower heap activities as the most probable cause of the variations. We also run the PostMark file system benchmark and the PassMark PerformanceTest bench-mark and see similar overhead (3.9% and 4.9%, respectively).

Table 2. Performance Overhead of KQ Guarding in WRK

Workload	Original (sec)	KQ Guarding (sec)	Slowdown
Super PI [33]	2,108±41	2,213±37	5.0%
Copy directory (1.5 GB)	231±9.0	244±15.9	5.6%
Compress directory (1.5 GB)	1,113±24	1,145±16	2.9%
Decompress directory (1.5 GB)	181±4.1	186±5.1	2.8%
Download file (160 MB)	145±11	151±11	4.1%

6 Related Work

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

SecVisor [29] and NICKLE [28] 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 [36] 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 injection attacks because they do not modify existing and legitimate kernel function pointers but instead supply malicious data in their own memory (i.e., the KQ request data structures). CFI [1] can ensure that control transfers of a program during execution always conform to a predefined control flow graph. Therefore, it can be instantiated into an alternative defense against KQIs that supply malicious control data. However, CFI cannot defeat the type of KQI attacks that supply malicious non-control data because they do not change the control flow. SBCFI [25] 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 injection attacks are transient, so SBCFI may miss many of them. Moreover, SBCFI requires source code so it does not satisfy the inclusiveness requirement. IndexedHooks [19] 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 [37] is a comprehensive defense against KQ injection attacks, capable of defeating both callback-into-malware and callback-into-libc attacks. However, PLCP does not satisfy the inclusiveness requirement due to its reliance on source code.

7 Conclusion

Kernel Queue (KQ) injection attacks are a significant problem. We test 125 real world malware attacks [2,4,10,11,14,16,18,23,27] and nine synthetic attacks to cover 20

KQs in the WRK. It is important for a solution to satisfy four requirements: efficiency (low overhead), effectiveness (precision and recall of attack detection), extensibility (accommodation of new device drivers) and inclusiveness (protection of device drivers 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 injection 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 very low false negatives (detecting all but two KQ injection attacks in 125 real world malware and nine synthetic attacks) and zero false positives (no false alarms after a proper training process). In performance, KQguard introduces a small overhead of about 100 microseconds per validation and up to about 5% slowdown for resource-intensive application benchmarks due to heap object tracking.

Acknowledgements. We thank Chenghuai Lu for sharing his knowledge and experience on real-world malware and Open Malware for sharing their malware samples. We also thank the anonymous reviewers for their useful comments. This research is supported by Centre for Strategic Infocomm Technologies (CSIT), Singapore. The opinions in this paper do not necessarily represent CSIT, Singapore.

References

1. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control flow integrity. In: Proceedings of the 12th ACM CCS (2005)
2. Anselmi, D., et al.: Battling the Rustock Threat. Microsoft Security Intelligence Report, Special edn. (January 2010 through May 2011)
3. Baliga, A., Ganapathy, V., Iftode, L.: Automatic inference and enforcement of kernel data structure invariants. In: Proceedings of ACSAC 2008 (2008)
4. Boldewin, F.: 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.: Invisible intruders: rootkits in practice. Login: 24 (September 1999)
6. Butler, J.: DKOM (Direct Kernel Object Manipulation), http://www.blackhat.com/presentations/win-usa-04/bh-win-04-butler.pdf
7. Carbone, M., Cui, W., Lu, L., Lee, W., Peinado, M., Jiang, X.: Mapping kernel objects to enable systematic integrity checking. In: Proceedings of ACM CCS 2009 (2009)
8. Castro, M., Costa, M., Harris, T.: Securing Software by Enforcing Dataflow Integrity. In: Proceedings of OSDI 2006 (2006)
9. Chiang, K., Lloyd, L.: A Case Study of the Rustock Rootkit and Spam Bot. In: Proceedings of the First Workshop on Hot Topics in Understanding Botnets, HotBots 2007 (2007)
10. Decker, A., Sancho, D., Kharouni, L., Goncharov, M., McArdle, R.: Pushdo/Cutwail: A Study of the Pushdo/Cutwail Botnet. Trend Micro Technical Report (May 2009)

11. Giuliani, M.: ZeroAccess – an advanced kernel mode rootkit, rev 1.2., http://www.prevxresearch.com/zeroaccess_analysis.pdf
12. Hayes, B.: Who Goes There? An Introduction to On-Access Virus Scanning, Part One. Symantec Connect Community (2010)
13. Hoglund, G.: Kernel Object Hooking Rootkits (KOH Rootkits) (2006), http://my.opera.com/330205811004483jash520/blog/show.dml/314125
14. Kapoor, A., Mathur, R.: Predicting the future of stealth attacks. In: Virus Bulletin 2011, Barcelona (2011)
15. Kaspersky Lab. The Mystery of Duqu: Part One, http://www.securelist.com/en/blog/208193182/The_Mystery_of_Duqu_Part_One
16. Kaspersky Lab. The Mystery of Duqu: Part Five, http://www.securelist.com/en/blog/606/The_Mystery_of_Duqu_Part_Five
17. Kil, C., Sezer, E., Azab, A., Ning, P., Zhang, X.: Remote attestation to dynamic system properties: Towards providing complete system integrity evidence. In: Proceedings of the International Conference on Dependable Systems and Networks, DSN 2009 (2009)
18. Kwiatek, L., Litawa, S.: Yet another Rustock analysis... Virus Bulletin (August 2008)
19. Li, J., Wang, Z., Bletsch, T., Srinivasan, D., Grace, M., Jiang, X.: Comprehensive and Efficient Protection of Kernel Control Data. IEEE Transactions on Information Forensics and Security 6(2) (June 2011)
20. Microsoft. Using Timer Objects, http://msdn.microsoft.com/en-us/library/ff565561.aspx
21. Microsoft. Checked Build of Windows, http://msdn.microsoft.com/en-us/library/windows/hardware/ff543457%28v=vs.85%29.aspx
22. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate language and tools for analysis and transformation of C programs. In: Nigel Horspool, R. (ed.) CC 2002. LNCS, vol. 2304, pp. 213–228. Springer, Heidelberg (2002)
23. OffensiveComputing. Storm Worm Process Injection from the Windows Kernel, http://offensivecomputing.net/papers/storm-3-9-2008.pdf
24. Petroni, N., Fraser, T., Molina, J., Arbaugh, W.A.: Copilot—a coprocessor-based kernel runtime integrity monitor. In: Proceedings of the 13th USENIX Security Symposium (2004)
25. Petroni, N., Hicks, M.: Automated detection of persistent kernel control flow attacks. In: Proceedings of ACM CCS 2007 (2007)
26. Petroni, N., Fraser, T., Walters, A., Arbaugh, W.A.: An Architecture for Specification-Based Detection of Semantic Integrity Violations in Kernel Dynamic Data. In: Proceedings of the 15th USENIX Security Symposium (2006)
27. Prakash, C.: What makes the Rustocks tick! In: Proceedings of the 11th Association of anti-Virus Asia Researchers International Conference, AVAR 2008 (2008)
28. Riley, R., Jiang, X., Xu, D.: Guest-transparent prevention of kernel rootkits with VMM-Based memory shadowing. In: Lippmann, R., Kirda, E., Trachtenberg, A. (eds.) RAID 2008. LNCS, vol. 5230, pp. 1–20. Springer, Heidelberg (2008)
29. Seshadri, A., Luk, M., Qu, N., Perrig, A.: SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In: Proceedings of ACM SOSP 2007 (2007)
30. Sharif, M., Lee, W., Cui, W., Lanzi, A.: Secure in-VM monitoring using hardware virtualization. In: Proceedings of ACM CCS 2009 (2009)
31. Smalley, S., Vance, C., Salamon, W.: Implementing SELinux as a Linux Security Module. Technical Report. NSA (May 2002)
32. Designer, S.: Bugtraq: Getting around non-executable stack (and fix), http://seclists.org/bugtraq/1997/Aug/63

33. Super PI, http://www.superpi.net/
34. Symantec Connect Community. W32.Duqu: The Precursor to the Next Stuxnet (October 2011), http://www.symantec.com/connect/w32_duqu_precursor_next_stuxnet
35. Tran, M., Etheridge, M., Bletsch, T., Jiang, X., Freeh, V., Ning, P.: On the Expressiveness of Return-into-libc Attacks. In: Sommer, R., Balzarotti, D., Maier, G. (eds.) RAID 2011. LNCS, vol. 6961, pp. 121–141. Springer, Heidelberg (2011)
36. Wang, Z., Jiang, X., Cui, W., Ning, P.: Countering kernel rootkits with lightweight hook protection. In: Proceedings of ACM CCS 2009 (2009)
37. Wei, J., Pu, C.: Towards a General Defense against Kernel Queue Hooking Attacks. Elsevier Journal of Computers & Security 31(2), 176–191 (2012)
38. Wei, J., Zhu, F., Pu, C.: KQguard: Protecting Kernel Callback Queues. Florida International University Technical Report, TR-2012-SEC-03-01 (2012), http://www.cis.fiu.edu/~weijp/Jinpeng_Homepage_files/WRK_Tech_Report_03_12.pdf
39. Windows Research Kernel v1.2., https://www.facultyresourcecenter.com/curriculum/pfv.aspx?ID=7366&c1=en-us&c2=0
40. Top 20 Malware Families in 2010, http://blog.fireeye.com/research/2010/07/worlds_top_modern_malware.html
41. Top 10 Botnet Families in 2009, https://blog.damballa.com/archives/572