Knowledge Base Model for the Linux Kernel

State of the Art and Feasibility Study

Mathieu Desnoyers, Ph.D.

President & Founder
EfficiOS Inc.
2541, Édouard-Montpetit blvd., #10
Montréal, Québec, Canada H3T 1J5
Phone: +1 514 394-0737
Toll-free USA/Canada: +1 888 666-5494
Fax: +1 514 394-0741
E-mail: mathieu.desnoyers@efficios.com
Website: www.efficios.com

For Defence R&D Canada

March 15th, 2011

Principal author

Mathieu Desnoyers, Ph.D.

Approved by

Approved for release by

Abstract

This study surveys the existing technologies currently available to assess the health of computer systems relied on by a soldier in the field of operation. It investigates ways to constantly detect system faults and discusses the feasibility of a Linux Knowledge Base (LKB) to organise the information analyzed by misuse- and anomaly-based detection systems, considering the aspects of operating concepts organisation, and by exposing deployment concerns. It proposes possible solutions in the area of instrumentation sources, LKB structure and LKB deployment.

Within this study, we propose a novel way to structure knowledge and analysis execution within a computer system and throughout a network, which we call "Distributed Observers", as a way to deal with the complexity of operating system modeling and its possibly large resource requirements. This study points at the major role played by communications between the instrumented system and the Observers, which leads us to recommend using an efficient buffered data transfer system to transport information between instrumentation sources and Observers.

Résumé

Cette étude survole les technologies présentement disponibles pour mesurer l'état de santé d'un système informatique utilisé par un soldat dans le théâtre d'opération. Elle analyse les moyens permettant de détecter, de manière continue, les fautes système et discute la faisabilité d'une Base de Connaissance Linux (BCL) pour organiser l'information utilisée par des systèmes de détection d'anomalies et de mauvaise utilisation, considérant l'aspect de l'organisation des concepts d'un système d'exploitation et en exposant les problématiques de déploiement. Cette étude propose d'éventuelles solutions dans le domaine des sources d'instrumentation, de structure du BCL et de son déploiement.

Dans cette étude, nous proposons une nouvelle manière de structurer la connaissance et l'exécution des analyses à l'intérieur d'un système et à travers un réseau, que nous appelons théorie des "Observateurs Distribués", comme moyen de gérer la complexité du modèle du système d'exploitation et sa possiblement grande demande en ressources systèmes. Cette étude montre le rôle majeur des communications effectuées entre le système instrumenté et les Observateurs, ce qui nous amène à recommender l'utilisation d'un système de transfert de données efficace basé sur un tampon circulaire pour transporter l'information entre les sources d'instrumentation et les Observateurs.

CONTENTS

I. Introduction

In a context where soldiers in field operation become increasingly dependent on computer systems, thus requiring to trust a wide range of computer devices, from small embedded systems (smartphones) to large-scale servers, in hostile environments where connectivity is not necessarily fast nor reliable, a major question arises: can a given piece of equipment be trusted, and to what extent ?

This study looks at the existing technologies currently available to constantly answer this main question during system operation and discusses the feasibility of a Linux Knowledge Base (`LKB`), from the aspects organisation of operating system concepts and through a critical analysis of deployment concerns.

Section II of this document presents the methodology used throughout this research, along with its expected limitations. Section III then discusses the targeted systems and the type of faults aimed at. Then, an hypothesis is formulated as the "Distributed Observer" theory in Section IV. Each of its aspects is then scrutinized through a state of the art review in Section V. Then, Section VI presents the useful information sources to consider when instrumenting the operating system and its applications. Section VII proposes a Linux Knowledge Base (`LKB`), and, finally, Section VIII discusses the major design aspects related to the deployment of the `LKB`.

II. Methodology

The methodology used for this research is in large part determined by realisation that the performance impact of the solutions proposed will be of major importance, as supported by the problem analysis and an initial literature study. The *LKB* (Linux Knowledge Base) feasibility cannot be studied independently from its deployment on systems, because this will affect the amount of data that can be gathered and processing power required to perform analysis on the `LKB`.

In this research, we are therefore opting for an initial abduction. We formulate a theory: the *Distributed Observers* based on prior knowledge of operating systems and of the security field. A top-down approach is used to specify a possible `LKB` deployment structure. A top-down approach is also used to specify the `LKB` per se, based on the main Linux kernel abstractions.

Then, a state of the art study is conducted to compare each aspect of the theory of Distributed Observers with prior work in the area. An argumentation is therefore brought forth by critically analyzing the Distributed Observers theory against the information gathered in the state of the art. This phase could be thought of as "bottom-up", for which every major aspect identified in the Distributed Observer theory is analyzed against the state of the art.

In a spirit of applied research, this document looks at the practical aspects of the Knowledge Base, including its deployment. Therefore, the focus is on lower-level concerns, namely the information sources to consider for populating the KB, the organisation of the Knowledge Base, the analysis that can be performed on the KB as well as the performance considerations related to data extraction from the information sources and execution of the analysis on the KB per se. The focus of this report being on the specification of the `LKB` as well as its deployment consideration, analysis techniques that can be applied on the `LKB` are treated more succinctly.

III. Problem Analysis

This section describes the scope of the problem treated, in terms of characteristics of the systems that are deemed interesting (faults and threats), the systems targeted, as well as the consequences of errors in fault identification.

A. Targeted Systems

The systems targeted include online servers as well as small embedded systems, both used in defense operations. The server system class has lots of `CPU` time available, more than enough memory space, some unused CPU-to-memory bandwidth in the interconnects between the CPU cache and the main memory. This class of system is also connected to a network with reliable and fast network connectivity. The embedded system class, e.g. smartphone, has limited memory resources, limited CPU resources, limited memory bandwidth available, as well as unreliable and slow network connection.

B. Targeted Threats and Faults

In the area of threats faced by the system, our targets includes known and yet-unknown (0-day) security holes, execution of known and unknown exploits on the system, and single-stage as well as multi-stage exploits. The attacks can come either from the network or from a direct physical attack (e.g. `DMA` (Direct Memory Access) performed by the USB or Firewire ports).

This study also targets system faults in a more general sense. The goal is to be able to monitor a system to determine if it is in a condition that could affect its ability to perform the tasks it was designed to perform. This includes system faults, application faults, out-of-memory conditions, hardware failure, etc.

C. Missed Faults vs False Positives

When considering the impact of failure to accurately identify a fault, two cases can emerge: either we miss an actual fault or threat, which means that the end-user will not be informed that he has an unhealthy system, or we falsely consider a behavior part of the normal system operation to be a fault or threat.

Therefore, not only is it important to be able to identify the faults; the fault detection system must also make sure to limit the number of false positives, otherwise the fault detection system becomes useless due to the amount of noise it generates. If humans analyzing the results given by the fault detector are unable to cope with the flow of false-positives, they will therefore discard the real actual faults as well.

False-positives are a real concern for a project targeting the behavior of the Linux Kernel due to the amount of information the can be produced by an operating system kernel and the large number of corner-cases that can arise from the diverse patterns that can emerge from the various the execution order caused by concurrent thread execution, both on multiprocessor systems and uniprocessor systems, the latter being caused by scheduler preemption.

IV. Distributed Observers Theory

We propose a theory, which we will call *Distributed Observers*, that attempts to deal with the problems faced, namely: the limited system resources (CPU time, memory bandwidth and memory space) available to embedded systems, the large amount of behavior data that can be generated by a farm of servers under `DDoS` (Distributed Denial of Service) attack, as well as deal with the amount of complexity that arises from the corner-cases caused by concurrent thread execution within the Linux kernel.

The core notion of this theory is the *Observer*. Each *Observer* executes at the privilege level it is assigned to (e.g. application-level, kernel-level, virtual machine manager-level, and, by extension, external CPU, or external node), which provides more or less isolation from the monitored system. Each *Observer* receives information gathered from a subset of the system, and stores information about the system's state in its own database in memory. Each *Observer* scrutinize its view of the system state to determine if it should produce a report – which we can also call an "abstraction" – of the system health. This is a type of data aggregation that is performed directly by the observer, thus lowering the amount of data that needs to be extracted, and consequently the amount of I/O bandwidth required. The level at which aggregation should be performed depends on the level at which interaction between sub-systems occurs: although monitoring process-specific information might be enough to generate some types of process-centric reports, information coming from multiple observers might be required to determine if a given portion of the system is healthy.

The *Distributed* nature of the *Observers* comes from the tree-like hierarchy of the reports being sent and analyzed: the lower-level observers (leaf nodes of the tree) observe directly information gathered either from applications, the operating system, the virtual machine manager or from an external CPU, and sends reports to the uppermost observer (its parent node). For instance, an application observer would send reports to the operating system kernel observer to enhance the information collected by the latter. In turn, the operating system kernel observer would generate reports either sent over the network to an external observer node, or sent to an observer running outside of the operating system kernel security context: either to the virtual machine manager or to an autonomous CPU dedicated to system monitoring.

Communication between the *Observers* is buffered to amortize the runtime cost incurred by switching between privilege levels. The observer reaction to threats and faults is therefore delayed, in opposition to a preemptive reaction measure that would interrupt the monitored system execution while waiting for a decision. This fault reaction latency depends on three factors: the amount of observer communication buffering needed (which impacts system throughput), the privilege level at which data analysis must be performed to take a well-informed decision, as well as the amount of CPU time needed to perform the analysis at each level of abstraction.

V. State of the Art

The state of the art presented in this section is organised as follows. We first survey papers on known attack techniques and prevention mechanisms. Then, we review articles on executable integrity validation techniques. We present prior studies on knowledge bases and intrusion response systems. Anomaly detection is discussed, to finish with a presentation of earlier work on tracing.

One important attack class is buffer overflows, caused by lack of validation of array bounds in programs, for which techniques like *Canary* can be applied to mitigate the chances of attack success and report attempts [1]. The *Canary* is a piece of random data automatically inserted by the compiler around every stack area subject to buffer overflow used to detect overflows by checking the canary validity before performing key operations that rely on the stack data, such as returning from a function.

Exploitation of kernel-level security holes [2] is becoming an increasing concern in the practitioner community. Typical exploit scenarios are multi-stage: the first stage is exploitation of a security hole in an unprivileged application to gain local access to the system, followed by exploit of a kernel-level security hole to gain complete control.

Multi-stage attacks are also present across machines within a network, where attack on series of nodes gradually lead to increased network accesses. Work has been done on representation of these multi-stage attacks [3].

Return-to-libc is an example of attack schemes that require knowledge of the address-space layout of the process attacked. Address space randomization [4], as initially available under Linux with the PaX [1] module – some features of which are now included in the mainline Linux kernel – provides some degree of protection against these attacks by randomizing the base addresses used by the stack and library memory mappings, requiring the attacker to brute-force the address space to try finding the right addresses. On 32-bit address-spaces, the effectiveness of this technique is low due to the limited entropy of the number of bits available, but it provides a better degree of attack mitigation on 64-bit. It is interesting to notice that transforming an attack that would otherwise succeed in a single attempt into a brute-force attack statistically allows an intrusion detection system to become aware of the attack before it succeeds.

Further work has been done on addresss-space obfuscation [5], where it is proposed to randomize the layout of the stack per se by e.g. varying the size of the function stack frames. This study also points out that *Canary* values can be defeated by attacks targeting code pointers in the heap area, and by attacking arguments to `chmod` (change file permission) or `execve` (execute binary program) system calls, integer overflow, heap overflows and double-free. Randomization of the program address-space layout significantly decreases the chances of attack success by adding diversity in the application.

In the area of system integrity validation, an interesting survey of checksum techniques to validate integrity of executables in memory, along with their pitfalls in a context where the operating system is untrusted, is presented in [6]. This specific article describes how presenting different views of the instruction memory and data memory (Harvard architecture) can bypass checksum techniques. This requires, however, complete control over the operating system, so the attack technique discussed here would be entirely irrelevant if we can validate the OS integrity (its executable pages and the consistency of some key memory mappings) from an external observer (isolated processor or virtual machine manager). It interestingly shows, however, how checksum techniques can be applied to executable code to validate its integrity.

The area of certified (or trusted) execution is also interesting. Certified execution try to ensure that

1. PaX ASLR: http://pax.grsecurity.net/

the code executed is indeed the code signed by the authors. This is enforced both at the processor level, and, as proposed in this article [7], at the memory or cache level. This article proposes a scheme to certify execution of code located in untrusted memory (memory that can be modified without processor awareness). It proposes the use of a Merkle tree (hash tree) to put the root hash in a trusted location, and leave leafs in insecure memory. It also proposes doing the memory page content validation at the cache-level rather than between cache and memory, to miminise the impact on the memory bus.

Going further in the area of trusted execution, this article [8] presents how the Trusted Platform Module (TPM), available as a security-enhancing co-processor, e.g. the Trusted Execution Technology (TXT) on recent Intel platforms, can be used to certify that the executable pages across the Linux software stack are indeed those expected to run. They explain that the dynamic nature of the kernel, with executable modules being loaded and unloaded in various order, complicates the task of validating executable pages content. They present solutions to certify the content of the dynamic loader, dynamically loaded libraries and kernel modules, in addition to validation of the loaded executables. However, the TPM technology used must be trustable and solid, as these TXT attack techniques presented at the BlackHat conference demonstrate [9].

In the area of Knowledge Bases and Intrusion Detection Systems (IDS), Stakhanova's Ph.D. thesis [10] proposes an integrated intrusion detection and response system based on anomaly and specification-based approaches. It details the major IDS approaches as either misuse-based (based on attack signature), anomaly-based or specification-based. It distinguishes between preemptive counter-measures (active response) and delayed counter-measures (passive response). Preemptive counter-measures will stop the attack before it succeeds, but greatly suffers from false positives. Delayed response can allow multi-stage analysis of the situation, which is more precise. The author states that signature-based detection results in low rates of false-positive, but cannot identify novel intrusions. It is worth noting that this statement apply to behavior signatures, e.g. input-outputs of a system, not to signatures in the sense of system integrity validated by certificates or checksums. None of the IDS approaches presented by the author (misuse, anomaly and specification-based) focus on checking system's integrity, probably because this work evolved from network-based detection systems which information input is limited to the network packets exchanged on a network, and must thus consider the system as a black box. This thesis also discusses attack graphs, presenting them as a way to represent intrusion signatures through nodes, but their complexity is stated as a scalability problem. It also discusses the importance of performing intrusion response at a level where knowledge about the overall system is available, thus arguing for having a detection system that is aware of the system as a whole. An important aspect of this work is the classification of system resources in memory (text, data, stack), input/outputs (file system and network interface).

An example of knowledge bases used for misuse detection in networks is presented in [11]. Their approach is based only on known vulnerability, hence limiting its applicability to our current research.

Focusing slightly more on security faults per se, Aslam's Master thesis [12] presents a taxonomy of security faults in the UNIX operating system. Its figure 2.1 (p. 40) presenting the taxonomy of faults is especially interesting, as it splits the type of security faults into the various sub-categories that must each

be considered in the detection techniques proposed. The author proposes dynamic and static analysis for fault identification, and discusses the cost of testing.

In the area of anomaly detection, this article [13] compares anomaly outlier detection schemes. It discusses misuse detection (by detection of signatures) and anomaly detection. It points out that misuse detection fails to discover yet unknown attacks, but is very good to identify known attacks. Anomaly detection yield to a high rate of false alarms. This occurs primarily because it recognises yet unseen, but legitimate, behaviors as intrusions.

Still in the area of anomaly detection, article [14] presents a lightweight IDS based on observation of short sequences of system calls to perform anomaly intrusion detection. This approach seems to target an ideal scenario where the attacker would not purposefully inject system calls as noise to defeat the short attention-span of the detection engine. Their model is a per-process trace of system calls. They perform an interesting investigation of the false positives they found, along with a detailed analysis of each case.

In the area of delayed intrusion detection, transporting information from the instrumented system to the detection engine is likely to be resource-consuming, especially in servers servicing heavy workloads, including DDoS scenarios. Desnoyers's Ph.D. thesis [15] presents the LTTng [2] tracer infrastructure, which can be used to transfer information efficiently between privilege levels in a Linux system through a lock-free ring-buffer scheme, thus amortizing the performance impact of privilege level changes. This ring buffer infrastructure is also available on various embedded architectures, such as ARM, MIPS, SuperH and PowerPC, in addition to x86. This work also presents techniques to perform source-level kernel and application instrumentation, and surveys dynamic instrumentation techniques with code replacement.

VI. Information Sources

This section presents the various information sources that could be used as input to populate the knowledge base. These inputs come from various code locations and privilege levels, including operating system, application and library instrumentation, sampling, as well as checksum-based integrity verification techniques such as canaries and checksums.

A. Operating System Source-Code Instrumentation

Source-level instrumentation is typically added directly into the source-code by application developers or by software vendors. It is transformed into executable statements by the compiler and added to the resulting executable objects. It is therefore shipped with the operating system, allowing a tracer to export trace data whenever an instrumented code site is executed.

The Linux kernel is statically instrumented with the Tracepoints infrastructure [15], which, amongst a lot of information that can be selected, provides information about system call entry and exit, as well as scheduler activity, and executable names. The information targeted by the static instrumentation should be carefully chosen to enable population of the Linux Knowledge Base.

On the topic of dynamic (runtime) versus static (added at source-code level) instrumentation, from an open source project point of view, source-level instrumentation follow code changes more reliably than

2. LTTng: http://lttng.org

binary-level instrumentation, because it is updated by the developers. Source-level instrumentation will also allow a faster execution in all cases, while dynamic instrumentation only provides fast execution for a limited set of object code locations.

B. Application and Library Source-Code Instrumentation

Instrumentation of libraries and application source-code can be performed similarly to kernel instrumentation using the `LTTng-UST` [3] tracer, as described in [15]. This type of instrumentation is useful to extract application-specific behavior information such as incorrect password attempts. This requires knowledge of what the applications inputs are (where the passwords are entered) to accurately report the unsuccessful attempts.

C. Execution Trace vs Sampling

Gathering an application or kernel execution trace allows in-depth analysis of the system state changing through time. This gives, for instance, knowledge of the exact execution sequence within a specific process which led to a fault, and which instruction pointer address (and thus which of the main executable file or which library) caused the fault. The downside of gathering execution traces is generally their rather heavy performance impact, which can be mitigated by activating a smaller subset of instrumentation, and by using an efficient data-gathering mechanism.

Sampling allows gathering general system tendencies (e.g. approximate CPU time used by each application) without impacting performance as much as tracing. It usually hooks into the interrupt handler, and at key sites in the operating system, such as the scheduler, to gather samples periodically. It can therefore produce reports on the most statistically significant part of the operating system activity. Its downside is that rare outliers are usually not accounted for. It can however be used to find if overall resource usage dramatically change over time.

D. Canary

In computer security, a Canary is a buffer overflow protection technique that can be explained with the analogy of the canary used in coal mines [4], which were used to detect the presence of toxic gases before the miners were affected by them.

1) Stack Canary: In the context of stack buffer overflow detection, the Canary technique can be used to report stack overflows before returning from functions. These canaries are added to the application by providing compiler add-ons. Canaries are provided by Stackguard, GCC stack-smashing protector (ProPolice) -fstack-protector flag, Microsoft Visual Studio /GS flag, the IBM Compiler with -qstackprotect flag, and StackGhost, a hardware-based technique unique to the SPARC architecture.

These canaries can provide very valuable input about break-in attempts by reporting most attempts to exploit a stack overflow. This allows attack detection by identifying integrity alteration attempts.

3. `LTTng-UST`: http://lttng.org/ust
4. For more information see http://en.wikipedia.org/wiki/Buffer_overflow_protection.

2) Heap Canary: Techniques similar to Canary could be integrated into memory allocators to discover heap overflows, double-free and use-after-free bugs. This would allow gathering a wider range of break-in attempts. This mechanism also enables attack detection by identifying integrity alteration attempts.

On Linux, an example of such systems that validate the integrity of the memory allocator is KERNHEAP. On NetBSD/OpenBSD, a memory allocator diagnostic and debug feature can be activated [5]. These tools, however, are all targeted at validating the use of the kernel heap, not the application-level heap. An example of a project targeting the user-level is the *Malloc Debug Library* [6].

3) Segmentation Faults: Along with address randomization techniques presented in the state of the art, we can rely on reports of segmentation faults occurring in the application as indicator of break-in attempts. By adding diversity into the application, the chances of success of an exploit are lowered, and thus brute-force techniques have to be used to guess the right addresses to exploit. In a scheme where most unsuccessful attempts generate a segmentation fault, it becomes easy to use these segmentation faults as a system integrity alteration report.

E. Executable Page Checksums

Checksumming executable pages can be used to check the integrity of the executable code. From the perspective of the operating system kernel, all application executable pages are known, which allows detection of executable pages added to a process by successful attack of an intrusion vector, and existing executable pages modified. Page protection flags allow protection of executable pages against modification, and changes to these protection flags must be performed by the operating system through the system call mprotect() in Linux. It is therefore possible to catch these changes to the memory pages access rights in the traces by monitoring a few system calls, mainly: exec(), mprotect(), brk(), mmap() and munmap(). Verifying the checksums of executable pages at these key kernel sites could allow to certify the validity of all the user-level executable code.

Validation of the kernel-level executable code should be performed by a privilege level higher than the kernel per se, because we cannot trust the kernel code, since it may have been tampered with. Using a `TPM` co-processor, an external isolated processor, or a virtual machine manager, could allow external verification of the kernel executable pages checksum.

VII. Linux Knowledge Base

This section proposes a Linux Knowledge Base (`LKB`) which describes the operating system resources that need to be represented into an entity-relationship structure to link these resources together. The entity-relationship structure is presented first, followed by a description of each resource and their links to other resources. Then, some of the operating system's critical points are presented, followed by example usage scenarios using the Knowledge Base to detect abnormal behavior, including programming error faults triggered by the end user to active exploitation of security breaches.

5. See http://www.phrack.org/issues.html?issue=66&id=15.
6. See http://www.hexco.de/rmdebug/

A. Knowledge Base Structure

This subsection presents the `LKB` structure, which consists of entities, attributes (describing each entity), and relationships between entities.

1) Entities and Relationships: A natural way to represent the operating system resources is to insert them into a hierarchical structure (a tree) of resources. We then superimpose references between its constituent entities (objects) to form a directed acyclic graph describing the parent-child relationships between the entities, as shown in Figure 1.

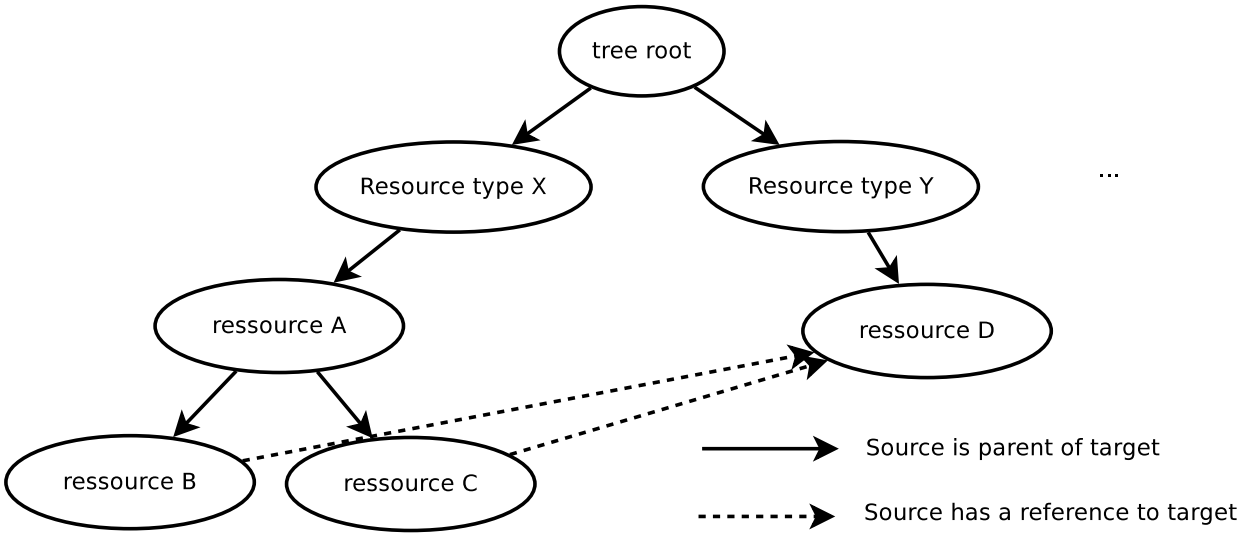

Figure 1. Abstract view of a resource graph

For example, as presented in Figure 2 one or more processes can share a file descriptor, but a thread only belongs to a single process. Therefore, we can create a link from each process having a reference to a given file descriptor to the resource entity that represents this file descriptor. In the case of the threads, we can create a link from each process to each thread it contains, and optionally a link from each thread to its container process. The reason why we use a reference link from processes to threads rather than a parent link is that kernel threads do not belong to any process.

2) Entity Attributes: Each entity has attributes. These attributes describe properties specific to each entity or class of entities. These attributes can represent the state of an entity over a period of time (temporal attributes), or sampled average and standard deviation of a peculiar entity attribute. It should be noted that statistic attributes can be populated by any instrumentation sources, including event-based static and dynamic instrumentation sources, sampling, and automated integrity validation methods. Statistical attributes can be inherited from bottom nodes (children) to upper nodes (their parents) in the hierarchy, as well as by following some of the references backwards (e.g. from file descriptors to all the processes containing them).

Let's consider processes attributes as an example. Some attributes for processes would be their process ID (which is invariant along the life of the process), reference to their set of threads and their process name (which can both vary during the process lifetime), their user ID and group ID, the file descriptor

references they keep, and so on. If we consider thread attributes, we could consider their CPU runtime as an example of statistical attribute. File descriptor attributes would include the opened file, memory mapping or interprocess communication resource they refer to, and the access rights. More information on the specific operating system resources attributes follows in the next subsection.

3) Entity Constraints: A set of constraints can be applied on these per-entity and per-entity class attributes. These constraints can be either temporal, punctual or statistical. Temporal constraints apply over a certain duration. For instance, we can specify a limit on the number of new connection attempts (SYN packets) per source IP address on a network card per second. Punctual constraints are simple assertions that must hold at a single moment (a way to express more complex assertions applying to more than a single entity is presented in Section VIII-F). An example of simple assertion would be that a process should never receive a segmentation fault. Finally, the statistical property would include a reference value and an expected maximum standard deviation over a period of time. If the system goes beyond the specified maximum standard deviation, a fault is reported. Constraints can be applied from upper nodes (parents) to bottom nodes (children) of the hierarchy, as well as by following references forward (e.g. from a given process to all file descriptors they contain).

It is important to point out that the way faults can be detected in the model is by comparing attribute values to the constraints (or ranges) that define their normal operation.

Basic constraints (applied to attributes) can be grouped together with logical operators to create more elaborate constraints. For instance, a constraint could target specifically process name “apache” *and* a specific maximum CPU usage, thus only applying the constraint onto the “apache” processes.

Independently of the language and user interface used to describe patterns and anomalous behaviors, these descriptions should be translated into an intermediate representation and stored into the proposed `LKB` so constraint verification can be performed efficiently. Therefore, the choice of the user interface and/or constraint description language is outside of the scope of the `LKB` per se. These user interfaces should however allow expression of `LKB` constraints: targeting a class of entities, some of its attributes,

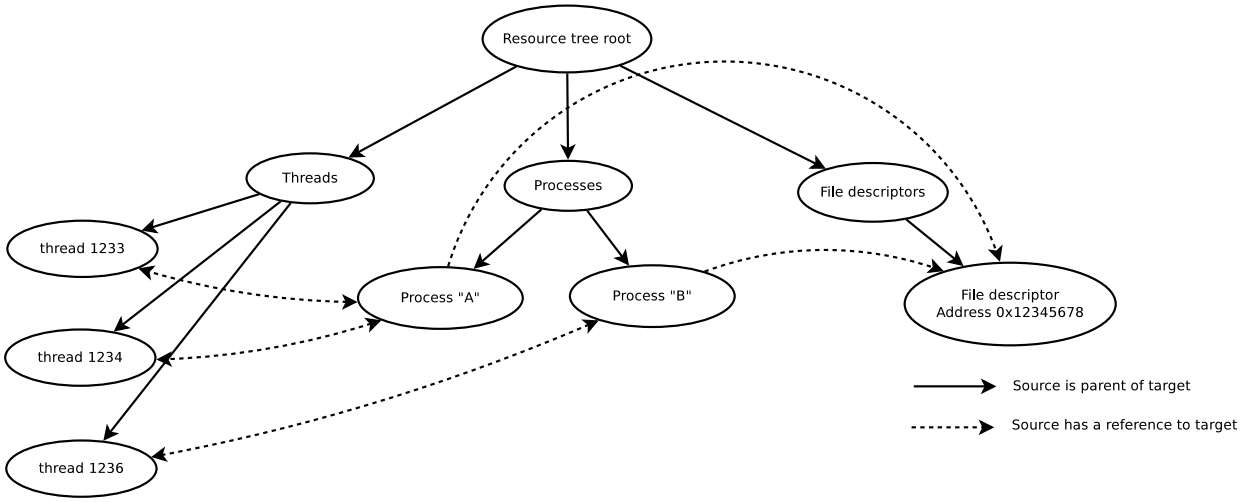

Figure 2. Graph of process, threads and file descriptors

along with ranges of values permitted for the attributes.

B. Operating System Resources

In this section, we use the abstractions conveyed within the Linux kernel as a basis for listing the key system resources that need to be represented in the `LKB`. These resources are detailed in [16], and some more information about the Linux-specific implementations of these abstractions are described in [17]. A good starting point to find the detailed description of the system calls discussed in this section is the Open Group Base Specifications, IEEE Std 1003.1-2008 (also known as POSIX-1.2008) [18]. Approaching the operating system instrumentation from the point of view of the POSIX standard will contribute to facilitate porting the `LKB` from Linux to other operating systems implementing the POSIX API (such as some flavors of BSD).

Conversely to an approach that would require instrumentation of each driver present in the Linux kernel, this approach focuses on the main abstractions used by these drivers (internal Linux kernel APIs) to organise information collected about the system behavior.

The entities populating the resource tree presented in Section VII-A1 are operating system resources, as presented in this subsection. We describe here each entity along with their parent-children relationships with other entities and references to other entities. A top-level view of the resulting resource graph is presented in Figure 3, which present the type of relationships allowed between each resource class.

1) Processes and Sessions: The main operating system resource is the process, also known as a task. It contains all the resources required to properly execute a user-space program.

It has a set of signal handlers, and references to thread, memory mappings (some of which can be shared with other processes) as well as references to file descriptors handlers. A process also has a name attributed by the exec() system call (which can change during the process life-time), a unique process ID in the system (unique for the process life-time), a parent process ID and a session ID. It also belongs to a specific user (real user ID) and group (real group ID), and has an effective user ID and group ID, which are set at program execution when the setuid bit is set.

Each process belong to a session. There does not seem to be any hard rule to specify when a process must create its own session, but usage shows that each command-line terminal has its own session assigned, and each graphical application usually receives its own session ID (depending on the window manager behavior). This helps grouping applications made of many processes.

2) Threads and Processors: Threads are the scheduling entity used to represent one execution flow of a program. Threads belong to a single process.

Each thread contains an execution stack attribute, which specify which execution contexts are being executed on the thread. These include: threads, traps, interrupts and soft-interrupts.

The scheduler activity determines on which processor each thread is running, or if a thread is waiting for another resource to become available before it resumes its execution. Therefore, we can have a reference relationship between processors and threads that changes over time based on scheduler activity. A thread is running on a processor over a specific time-frame (this scheduler-related link would be called the processor run-queue).

A thread can be running in user-mode (the application or library code is being executed) or in kernel mode on behalf of an application (when the application performs a system call or if a trap is encountered). System calls are the main interface between user-space applications/libraries and the kernel. Traps deal with situations that require kernel intervention, such as page faults, division by zero, and others. Some traps, but not all of them, are the results of abnormal activity. For instance, page faults happen at the creation of new processes due to the copy-on-write scheme used within the Linux kernel across the fork() system call. These are resolved by populating the appropriate pages in the page table. However, page faults can also be triggered by an attacker trying to access unmapped page address range. The page fault handler will typically fail in that case, sending a SIGSEGV signal (segmentation fault) to terminate the application, which indicates that a problem occurred. If the application does not catch (and deal with) SIGSEGV signals, this should be accounted as a fault resulting from a program bug or attack attempt.

Other execution context, unrelated to threads, are interruption and soft-interrupts (also called bottom-halves). Interrupts are triggered asynchronously by the processor, typically when one of its interrupt lines is activated by a device. Soft-interrupts are raised by interrupt handlers for later execution after the currently-running interrupt handlers have completed their own execution. These can therefore be considered as linked to a processor, and asynchronous, just like interrupts. Both of these execution contexts nest over the thread context, which may have some impact on statistics collected, because they operate on behalf of a thread. Therefore, a resource list of interruptions and soft-interrupts should be kept, each containing attributes specifying which device is being serviced and which driver is used. Processors, when running an interrupt handler, can temporarily link to the interrupt or soft-interrupt object in its execution stack.

Kernel threads are yet another type of execution context. They are scheduled on processors just like normal threads, but they do not belong to any process, and therefore they have no user-level memory mapping. They only execute in kernel context to perform some kernel-specific tasks.

3) Memory Maps: Memory mappings are a link between a virtual memory address space range and some physical memory pages. These mappings have access flag (read, write, execute) attributes that are enforced by the hardware. New memory mappings are created when the exec() system call loads a new executable file in memory, with section name attributes: bss and data sections for the data (e.g. global and static variables of a program), text segment for the executable code, and a section reserved for the main thread stack. As the program executes, the dynamic linker uses the brk() and mmap() system calls to dynamically allocate memory (section name: heap). Libraries are located in memory allocated with mmap(). Their protection flags are changed with the mprotect() system call. Threading libraries such as pthread split the stack memory mapping into smaller areas (8MB typically), and gives a different stack address range for each of thread created. Therefore, all threads share a common memory mapping (they all belong to the same process), but they each use a different stack.

The kernel has a set of memory mappings only accessible from the kernel privilege level (some of it can be explicitly shared with processes). The two main mappings are the kernel linear memory mapping and its virtual memory mapping.

Each process has a set of memory mappings, some of which can be shared amongst many processes,

and/or shared with the kernel.

4) Signal Handlers: Each process has signal handlers set to handle specific signals (actions to perform whenever a signal is received). These associations between signals and actions are the attributes of the "signal handlers" entities. The default behavior when no signal handler is set, depending on the signal, is to either ignore the signal or cause the process to exit. Processes can manipulate the behavior to exhibit when a signal is received for all signals except SIGKILL and SIGSTOP. Knowing which signal handlers are installed by applications and which behavior is triggered when the signal is received can help finding out if a signal received is part of the normal program operation or if it is generated by an attack. For instance, a program could override the segmentation fault signal (SIGSEGV) to implement a rather elaborate on-demand memory allocation mechanism, which would be part of its normal operation, but a program not expecting to trigger any segmentation fault would be considered to be faulty as soon as it receives a SIGSEGV signal.

5) File Descriptors: A file descriptor entity represents an open file, network socket or inter-process communication (`IPC`) artefact such as System V shared memory segments and pipes. A file descriptor can belong to one or more processes and can be duplicated with the dup(), dup2() and dup3() system calls. It is also possible to pass file descriptors and credentials between processes, in a way that is controlled by the kernel, through UNIX sockets. Monitoring these system calls is therefore required to updated the `LKB` entities and relationships accordingly.

File descriptors have access modes specified upon creation (open() system call), which can be altered by the fcntl() system call. It is deleted when the last process holding a reference to it releases it with either a close() system call or by exiting.

Typical read, write, message receive and send operations are performed on file descriptors. We can therefore keep counts of megabytes read and written on a per file descriptor basis as file descriptor attributes. File descriptors can also be associated with a memory mapping by the mmap() system call, which triggers accesses to the file backing the memory map automatically when memory accesses are performed on the memory map region. The memory map and the file are synchronized lazily, using page faults to fetch missing data from disk and flushing modified (dirty) memory pages to disk only when the memory cache pressure is high enough or when explicitly required, e.g. by munmap() or msync(). Therefore, accounting disk bandwidth utilization is not a trivial matter with mmap().

6) Virtual File System and Mounted Filesystems: The Virtual File System (VFS) is a tree consisting of access paths and associated operations. The root of this tree is the root path ("/"). Its children are the files and sub-directories it contains. Each of its sub-directory can contain files and directories, and directories can recursively contain sub-directories.

A mounted file system is an association between a physical block device (e.g. a disk partition) and a directory in the Virtual File System (VFS) tree of the operating system. For instance, Linux can map the first partition of the second SCSI disk (/dev/sdb1) to the VFS location `/boot`. A file system maps all operations performed on the VFS tree located under the mounted location (here `/boot`) to their actual implementation aware of the block device layout. These operations include readdir() (to list a directory content), open() (to open a file), unlink() (erase a file) and many others. These basic file system operations

are performed on inodes, which are found by performing a path lookup in the VFS path name tree. File systems also implement file operations, such as read() and write(), which apply on file descriptors returned by the open() system call.

The structure recommended to represent the VFS is to first keep a resource list of all mounted file systems, along with their attributes (target block device, target VFS mount point and mount options). Each of these mounted filesystems should have a tree representing their directory structure and their files. A path lookup is the act of associating a location in the VFS (e.g. /home/username/file) to the first inode of the file it points to. Path lookups and VFS traveral can therefore be followed by going through the mounted file systems resource list (e.g. to lookup on which file system the "/" directory is located), and then use the inner representation of the file system directory tree to perform the path lookup. Whenever a mount point is reached in a directory traversal, the mounted file system list should be used to move to the appropriate mounted file system to continue the path lookup.

A mounted file system object could keep attributes specifying the number of reads and writes performed on the file system overall, and on a per sub-directory basis. The number of filesystem data reads and writes versus the number of filesystem metadata accesses (e.g. use of the stat() system call) can be computed separately as an indicator of the amount of operations performed on the file system per se compared to the amount of operations performed within the files it contains.

7) Files: The files are the leafs of the VFS tree. They are identified with an access path within a mounted file system, and each have specific access rights (read, write, execute for each of user, group and others), as well as the setuid, setgid bits and the restricted delection flag (also called sticky bit). Each file has a owner user and group, a size, and a time of last access, last modification and last status change. All this information, usually available through the stat() system call, should appear in the leaf of the VFS.

Each file can have attributes describing the number of reads and writes and the amount of data read and written.

8) Block Devices (Disks): Disks are presented as block devices to the operating system. A resource list of block devices should be kept, each identified with their name in /dev (e.g. /dev/sdb1 for the first partition of the second SCSI block device). In addition, the Universally Unique Identifier (UUID) tag which may be associated with a block device file system could be used to uniquely identify a block device. This is useful, for instance, in a context where devices are dynamically added and removed (e.g. USB memory sticks), which leads to change in their block device name depending on the context.

For each block device, the number of interrupts could be kept, the measurement of data transferred (read and written), as well as the number of read and write operations performed.

Block devices can have a hierarchical mapping, thanks to the Device Mapper. This is used to create pseudo block-devices that perform encryption tasks or replication, such as RAID. A "md" block device could therefore have some properties representing the type of operation it performs, along with children block devices on which the operations are done (e.g. target disk block devices).

9) Network Interfaces and Routing Tables: Network interfaces are another type of resource. They are usually linked to an interrupt line (not shown in Figure 3 to preserve readability), which signals arrival of network data to a processor, a base address for I/O accesses and shared memory address range. They each

have one or more associated internet address (IPv4 or IPv6), along with a network mask, which specifies the addresses directly reachable. A system-wide routing table defines how non-network-local addresses should be contacted.

Each network interfaces has an expected throughput, defined by the type of its network interconnection. Each network interface has a counter of bytes sent and received, packets sent and received, as well as counters of erroneous packets, packets dropped and overruns, which give information on the link quality of the network. A network interface has a somewhat unique MAC address, assigned by the interface vendor. It is not truly unique because it can be overridden by software.

Wireless interfaces additionally have information about their access point name (ESSID) and MAC address, the protocol they use, their frequency, their bit rate, as well as general information about signal level and link quality.

10) Sockets: Sockets are yet another type of resource. They represent network connections. Like opened files, a socket entity is is associated to a file descriptor entity in the `LKB`. A socket specify one of various protocol families (AF_INET for internet protocols, AF_IPX for IPX, AF_PACKET for packet interface at the device level, AF_UNIX and AF_LOCAL for local UNIX sockets) and socket types (SOCK_STREAM for TCP (connection-based), SOCK_DGRAM for UDP (connection-less)). The protocol family and socket type is a socket attribute.

Basic system calls that manipulate the connection state of a TCP socket are bind(), listen(), accept() and connect(). Incoming connections are received with the following sequence: bind() opens a port, listen() waits for new incoming connections, and accept() accepts the connections. The connect() system call is used to perform an outgoing connection. The state of a TCP socket should be an attribute part of the socket attributes.

UDP sockets can be used with the system calls sendto(), sendmsg(), when provided with a valid destination address as argument. The connect() system call can also be used to specify a destination address, that will thereafter be used by the send() and write() system calls. The bind() system call is used to wait for incoming packets, which can be received one at a time with recvmsg() (and recvmmsg() to receive multiple messages).

Sockets can also be manipulated by other Linux-specific system calls, such as sendfile() and splice(), to perform the data transfer operations more efficiently.

A connected internet protocol socket has an associated address (either IPv4 or IPv6 address) and target port. An accepted incoming connection has remote address and port attributes.

Each socket's attributes can contain the number of bytes read and written, as well as the number of packets sent/received.

11) Interprocess Communication (IPC): Another type of resources are the interprocess communication objects. Interprocess Communication (IPC) are used to exchange data between processes within a Linux system. The categories of IPC are: System V shared memory map, pipes and semaphores. Each is represented as a file descriptor to a process, thus adding a link between file descriptor entities and IPC in the `LKB`. Each IPC object can be shared between multiple processes.

12) Timers: Timers are also a resource to consider in the resource hierarchy. The Linux kernel has per-process POSIX interval timers and kernel-level timers as well. POSIX.1-2008 marks the getitimer() and setitimer() system calls obsolete, and recommends using the POSIX timer API (timer_create(), timer_gettime(), timer_settime(), and others) instead. Therefore, both APIs should be instrumented, as well as the kernel-internal high-resolution timer API, to track the state of timer resources.

The attributes associated with a POSIX timer are: its timer ID, its kind (whether it is automatically rearming or one-time), its time domain, its initial expiration time (either relative to the current time or absolute), its interval and the action that must be executed when the timer fires (either none, delivering a signal, or spawning a thread). In the `LKB`, timers can be associated with a process, and a signal to trigger.

The kernel-level timer API allows to associate a callback that must be executed in interrupt context when a after a specified delay expires.

Tracking POSIX interval timers is useful to follow sequences of system calls that encompass the execution of many timer handlers, which would otherwise allow attackers to obfuscate the sequences of system calls within the normal program's execution by hiding them within chains of timer handlers.

C. Operating System Critical Points

We could identify some critical points of the operating system as follows. We should note that this section does not pretend to list all the possible operating system critical points, as there are as many of them as the operating system, libraries, programs and architectures interactions are complex.

Some critical points:

- Return address of a function (in user-space, kernel-space, virtual machine manager).
- Any program variable that can be used to override the execution flow (e.g. any function pointer, strings passed as parameter to system calls like exec() or function calls like system()).
- The kernel module loader.
- Unhandled/unexpected signals sent to applications.
- Calls to mprotect() system call (changing memory protection) outside of the normal library loading of user-space application.
- Modifications to core system files (modification of application binary files, or modification of configuration files) when the system is not in "maintenance" mode.
- Direct access to block devices from user-space applications, thus bypassing the Virtual File System layer.
- Detection of new USB devices, which can access memory directly through Direct Memory Access (DMA).
- Network sockets opened by applications can be used for communication with a command center (for worms).
- Any executable page placed at a non-randomized address can be used to construct an attack shell-code from existing code.

Besides these obvious critical points, more general information about the system behavior (e.g. CPU time consumption, currently executing threads, amount of memory allocated) should be gathered to enable

identification of abnormal behaviors which are not necessarily triggering these critical points.

D. Example Knowledge Base Usage Scenarios

This subsection presents examples of how the Knowledge Base can be leveraged to facilitate detection of abnormal behaviors in the system.

1) Buffer Overflow Detection with Canary: The scenario is as follows: a canary is added on the application execution stack before each function return address, and a verification of the canary integrity is performed before returning from each function, trapping into the operating system when integrity verification fails.

In this case, we keep an extra attribute into each process class (one class per executable name) which counts the number of invalid canary detected. For certain class of processes, we specify two thresholds on the number of failures expected per hour before we report an attack attempt. For instance, 3 failure could indicate that a programming error is known, and 20 could indicate that an attacker is actively trying to exploit this program. This could therefore report programming errors triggered by the user as different threat level than programming errors actively exploited by attackers, thanks to the information about the total number of invalid canary encountered and the thresholds specified.

2) Denial of Service Detection with Operating System Instrumentation: Detection of denial of service can leverage the ability to keep statistical information on CPU usage, interrupt line usage, network interface usage, as well as source IP address tracking. Constraints on the expected CPU usage and network interface usage can be set to report a problem as soon as the system's CPU usage is reaching a level too high (for instance 90%) for a period longer than one minute.

The therefore uses information about the CPU scheduling (showing how frequently the CPU has been running on behalf of threads and IRQs, excluding the idle thread) to calculate the amount of CPU time the system was active on the overall time (which is the sum of active and idle time).

Keeping a system-wide table of recent connexions on a network interface, along with the source IP addresses, could allow pinpointing the source IP causing the denial of service, or, in case of a distributed denial of service, the overall system load helps indicating that the system is under attack, without being able to identify the source of the attack.

3) Rootkit Installation Detection with Operating System Instrumentation: Some usual operations are performed by rootkit installation after a system has been compromised, such as modifying files in the "/bin" directory, adding users in "/etc/passwd", loading modules, and so on. Monitoring each of these resources in the `LKB` permits reporting these abnormal behaviors happening after a successful intrusion.

False positives caused by normal administrative tasks could be taken care of by, for instance, adding a hardware switch that puts the system in "maintenance" mode rather than "normal usage" mode. This special operation mode would consider modification to core system files as normal, and would therefore not report these as attacks during that time. Disconnecting the system from the network would be recommended in "maintenance" mode.

VIII. Linux Knowledge Base Deployment Considerations

In this section, we present some recommendations regarding the deployment of the LKB proposed in the preceding section.

For detection of inappropriate behavior using the LKB, we propose the combination of 3 detection approaches: the (usual) misuse-based detection, which focuses on attack signatures, the (also usual) anomaly-based detection, which performs behavior analysis based on specifications or automatically trained. In addition, we recommend considering integrity verification approaches to monitor the integrity of key internal structures. For example, using Canary, executable page checksumming and instrumentation of key system calls such as mprotect() can help identifying attempts to attack the system. Finally, adding diversity into the applications will statistically help reporting exploit attempts before it succeeds.

The former of these approaches (misuse-based), is known to allow detection only of known attacks, but has a low rate of false-positives, hence it can be used with a high confidence level. It should ideally be integrated into a standard framework that permits information exchange on current security threat signatures, such as Mitre OVAL [7].

The anomaly-based approach can be used to find out about unexpected system faults and attacks, including yet unknown attack methods. However, this approach is likely to lead to many false-positives, and therefore should be used more as a low-confidence indicator.

The integrity verification approach is somewhat novel compared to the other methods in the field on intrusion detection, probably because typical IDS scenarios are based on network traffic monitoring, which only provides behavioral information gathered from the host's input-outputs. In this case, having direct access to the host operating system allows us to validate that key aspects of the system have not been tampered with. This can lead to detection of entire classes of exploits (e.g. stack overflows, heap overflows, double-free, etc.), including 0-days, without relying on knowledge about each specific exploit (unlike the misuse-based approach). Adding diversity to the applications can also allow to discover attack attempts before they are successful. Such diversity can be added by randomizing the address-space and obfuscating the application memory layout.

The information gathered from the various sources identified in Section VI should be categorized into the LKB proposed in Section VII. It can then be used to drive state validation engines linked one or more nodes of the LKB. Examples of state validation engines range from simple assertions on attributes, limits imposed on statistic attributes, to more complex state machines and Markov chains receiving many attributes from various LKB objects as inputs.

A. Delayed vs Preemptive Fault Identification

Two main approaches to reactive measures have been identified in the state of the art: preemptive response, which interrupts the program execution, and delayed response.

Preemptive response has the advantage to be able to act directly on the program execution, hence disallowing attacks, but its main downside is to have limited knowledge about the overall system context,

7. Open Vulnerability and Assessment Language (OVAL), http://oval.mitre.org/

thus making sub-optimal decisions which increases the risks of incorrectly reacting to a false-positive. Moreover, preemptive response adds overhead to the standard program execution, because it must take a decision on the input-output data paths. It therefore directly impact the system's response-time.

Delayed response faces the disadvantage of not being able to counter attacks, thus letting them through, but can report them more accurately, because the input information provided by the attack will be examined in a larger context, including knowledge about the global system state, and, in the case of multi-stage attacks, knowledge about the initial and following attack stages when taking a decision. Compared to this, preemptive response is limited to a view focused on each step of a multi-stage attack. In addition, delayed response allows buffering of data transfer between the instrumented program and the system performing the fault identification analysis, which can amortize the cost of privilege level changes.

Therefore, in a context where the objective is to be able to report faults, we recommend to use delayed fault identification to lower the risks of false-positives by increasing the knowledge of global system state and provide the ability to follow sequences of events, thus allowing discovery of multi-stage attacks. This should be expected to lower the performance impact incurred by monitoring activity by amortizing the performance impact of privilege level changes by buffering the communication between the instrumented programs and the analyzer.

B. LKB Resource Usage

As noted in the literature review, the inherent complexity of the systems modeled can be expected to make anomaly detection resource-hungry. Moreover, given the amount of information that can be exported form the Linux kernel, issues with memory space, data throughput and processing time can be expected.

The memory space usage considerations are especially applicable to small embedded systems, which will moreover need to be able to support disconnected operation. The database holding the `LKB` should therefore be as compact as possible.

Memory bandwidth is another consideration important for both small embedded systems and server systems. Without the help of special-purpose hardware, exporting data gathered from program and kernel instrumentation outside of a processor requires writing to memory, which consumes precious memory bandwidth otherwise available to applications and to the kernel. One way to minimize the memory bandwidth usage is to populate Knowledge Bases in a distributed fashion, therefore allowing anomaly detection at the level closest to the information retrieval. This would allow sending summarized reports to external memory, thus using the processor cache, which is more efficient, to transport the raw data gathered directly from system's instrumentation rather than the main memory.

Network bandwidth is another concern in terms of resource usage, which is especially important to consider for embedded systems connected with a slow network link and which might have an unreliable connection. This network bandwidth should be used with parsimony by sending higher-level reports to monitoring nodes rather than raw trace data. Therefore, the step of fault detection should in large part be done autonomously by the embedded systems, which in addition allows it to provide health measurements the its user even in disconnected operation mode.

Processor time is also a limited resource on embedded systems, although the current trend is to see these embedded devices becoming increasingly multi-core, which can leave processing power available to perform on-chip anomaly detection. However, the cost of synchronization required when exporting the raw data gathered by instrumentation of the system running on a set of cores to another set of cores that will analyze this information becomes important to consider, along with the extra memory bandwidth that would be consumed by moving this data. The alternative is to perform most of the analysis on the core on which the system runs, which has other downsides: it uses more CPU time, which leaves less to the deployed application.

For server systems under DDoS attack, similar considerations should be taken into account for memory bandwidth, network bandwidth and CPU usage as the embedded case, because these resources, in this scenario, will be limited.

All in all, the need for a compact representation of the LKB arises, along with low-overhead manipulation of the LKB as information is received from the system instrumentation and fast queries to the LKB. The query latency also has to be considered. Many compromises exist in the range between preemptive anomaly detection and delaying the response for a few minutes, which will impact whether the LKB is deployed close to the instrumentation source (the system), on the same CPU and same core, on the same CPU, but different core (thus sharing a common cache), on a different CPU, or on a different system reachable through a network connection.

C. Execution Privilege Levels (Rings)

The location of the LKB, in addition to be determined by the resource usage consumption, should also take into consideration the integrity of the LKB per se, allowing it to, at least, report when an attacker is tampering with the LKB.

The execution rings provided by the processor are one means to reach this target. Execution rings with higher privileges can access the memory of lower-privilege rings, but not the opposite. Therefore, we could use an observer in kernel-ring (high privilege, ring 0 on x86), to check the integrity of the observers located in lower-privilege rings (applications, ring 3 on x86). Hypervisor mode (for the virtual machine manager) could be used to inspect the kernels running in virtualized environments, thus allowing to detect attempts to tamper with the kernel-level LKB.

However, as pointed out in the literature review, using the processor privilege levels as the only mean to protect the LKB and the observers can fail due to processor bugs that would allow bypassing all these security measures. Using an entirely separate processor to verify the integrity of the system's observers would be one way to deal with this risk.

D. Distributed Observers Deployment

Considering the discussion presented in the two previous sub-sections, the distributed observers should be deployed at a privilege level they each require, but no more, because deploying observers at unnecessarily high privilege level is likely to increase the resources required to transport data from the instrumented system to the observer.

The observers should be located as close as possible to the data source to minimize the bandwidth it requires, and each should produce higher-level reports that can be aggregated by observers located in higher privilege levels.

E. Efficiency Recommendations

Coping with workload scalability should be ensured by keeping forward and backward direct references between LKB entities for each lookup operation performed often by the LKB updates and analysis.

Keeping per-cpu and per-thread Distributed Observers (smaller instances of the LKB) to perform local constraint verifications should help keeping a good scalability to multi-core and multi-processor system. In general, the modification operations performed on the LKB by information received from instrumentation sources as well as the lookups done into the LKB by analysis should have a constant complexity order ($O(1)$) and a small runtime constant.

Efficiency of the LKB should be ensured by keeping the LKB memory footprint low, ensure a good cache locality behavior and by keeping the complexity order constant, with a small constant value. This also involves minimizing the number of unneeded execution privilege level changes, for instance by batching data transmission with a ring buffer.

F. Behavior Study Approaches

Some approaches recommended to perform behavior analysis on the LKB are to describe temporal, punctual, and statistical constraints on the LKB attributes. Pattern matching could also be used to detect sequences of events occurring in the system.

Pattern matching should be applied at a hierarchy level needed by the pattern. For instance, a pattern can be matched using system-wide activity, and will thus look at all the branches of the LKB. We can, conversely, have more specific pattern matching automata that will look only at a subset of the LKB, e.g. all activity of a single process. This natural division of patterns into sub-trees of the LKB can be used as a primary criterion to decide where each part of the LKB hierarchy should be deployed (at which privilege level, on which core, and either locally or on a remote system over a network).

G. Open-Source vs Closed-Source LKB

In a context where the target operating system evolves as quickly as Linux, open-sourcing the LKB brings a clear advantage: it helps keeping up with the pace of the target operating system development, and it allows operating system designers and implementors to review the LKB and criticize it, helping making it better.

The downside of open-sourcing the LKB is that bugs and limitations of the LKB would be visible for anyone (including attackers) to see. However, we can argue that attackers could find other means to leak the LKB content, e.g. with direct physical access to devices on which it is deployed, or by executing attacks that disclose the content of the system memory, which therefore render this approach of security through obscurity rather pointless.

Open-sourcing the LKB will therefore thrive continuous improvements to its data structures and code-base. In all cases, whether the LKB is closed or open-source, it should be considered as known by the

attacker, so the focus should be on improving the pace of its development and making sure its integrity is verified by protected execution contexts (ring level, independent processors) rather than trying to hide this mechanism from attackers.

IX. Conclusion

In conclusion, our expectations are that, in a near future, evolution in the area of health-assessment will require integration of the various usual detection techniques (misuse-based and anomaly-based, both automated and specification-based) with deeper knowledge of the operating system, provided by instrumentation and integrity verification. Detection schemes will require organised knowledge of execution sequences to deal with pervasiveness of multi-stage host-level attacks enabled by an increasingly complex software stack.

We therefore recommend to integrate the three major detection approaches considered (misuse-based, anomaly-based and based on integrity verification) to combine each of their strengths to lower the impact of their respective weaknesses. We recall that misuse-based detection has a very low rate of false-positive, but fails to detect yet-unknown exploits. Anomaly-based detection triggers more false-positives, but allows detection of new abnormal situations. Finally, the integrity verification based detection we propose in this study can identify attempts to tamper with part of the operating system or application integrity. This is a supplementary analysis of the system that can help lower the dependency on a single approach (anomaly-based detection) that has known problems with false-positives. Therefore, the anomaly-based detection could be used with much more relax thresholds between what is considered good and bad behavior, thus lowering the amount of false positives it produces.

On the deployment side, we outlined the importance of keeping the performance impact as low as possible to ensure that the system being monitored stays usable. Therefore, we believe that transporting information from the system's instrumentation sites to the fault detection analysis modules (and `LKB`) should be done in a way that minimizes the overhead on the system by using a efficient buffered data transport such as `LTTng`. Use of a hierarchical distributed monitoring structure ("Distributed Observers"), as proposed in this study, is advised to bear with the overall complexity of the operating system interactions modeled in the `LKB`.

By following the recommendations brought forth in this document (summarized in the list below), it should be possible to develop a Knowledge Base for the Linux kernel that could be used to detect abnormal system behavior, thus enabling monitoring of system security through detection of attacks and abnormal behaviors, as well as detection of faults caused by user-triggered programming errors.

Recommendations

- Integrate misuse-based, anomaly-based and integrity verification-based fault detection approaches to minimize false-positives.
- Beware of performance-related limitations of `LKB` deployment.
- Use buffered, delayed threat detection.
- Use a distributed approach to aggregate threat-reports as close to the data source as possible.
- Transport data with tools that take performance considerations into account, such as `LTTng`.

– Ensure to limit the overall size of the LKB to fits within limited embedded system memory resources.
– Ensure to limit the computational complexity of the analysis run on the LKB to ensure that the CPU time taken by the analysis does not grow out of control as the system load increases.

REFERENCES

[1] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer overflows: Attacks and defenses for the vulnerability of the decade," *DARPA Information Survivability Conference and Exposition*, vol. 2, p. 1119, 2000.

[2] E. Perla and M. Oldani, *A Guide to Kernel Exploitation: Attacking the Core*. Syngress, 2010.

[3] S. Mathew, R. Giomundo, S. Upadhyaya, M. Sudit, and A. Stotz, "Understanding multistage attacks by attack-track based visualization of heterogeneous event streams," in *Proceedings of the 3rd international workshop on Visualization for computer security*, ser. VizSEC '06. New York, NY, USA: ACM, 2006, pp. 1–6. [Online]. Available: http://doi.acm.org/10.1145/1179576.1179578

[4] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM conference on Computer and communications security*, ser. CCS '04. New York, NY, USA: ACM, 2004, pp. 298–307. [Online]. Available: http://doi.acm.org/10.1145/1030083.1030124

[5] E. Bhatkar, D. C. Duvarney, and R. Sekar, "Address obfuscation: an efficient approach to combat a broad range of memory error exploits," in *Proceedings of the 12th USENIX Security Symposium*, 2003, pp. 105–120.

[6] J. T. Giffin, M. Christodorescu, and L. Kruger, "Strengthening software self-checksumming via self-modifying code," *Computer Security Applications Conference, Annual*, vol. 0, pp. 23–32, 2005.

[7] B. Gassend, G. E. Suh, D. Clarke, van Dijk Marten, and S. Devadas, "Caches and hash trees for efficient memory integrity verification," in *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, ser. HPCA '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 295–. [Online]. Available: http://portal.acm.org/citation.cfm?id=822080.822806

[8] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn, "Design and implementation of a tcg-based integrity measurement architecture," in *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*, ser. SSYM'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 16–16. [Online]. Available: http://portal.acm.org/citation.cfm?id=1251375.1251391

[9] R. Wojtczuk and J. Rutkowska, "Attacking intel trusted execution technology," in *BlackHat*, 2009, [Online]. Available: http://www.invisiblethingslab.com/resources/bh09dc/AttackingIntelTXT-paper.pdf.

[10] N. Stakhanova, "A framework for adaptive, cost-sensitive intrusion detection and response system," Ph.D. dissertation, Iowa State University, 2007.

[11] B. Shepard, C. Matuszek, C. B. Fraser, W. Wechtenhiser, D. Crabbe, Z. Güngördü, J. Jantos, T. Hughes, L. Lefkowitz, M. Witbrock, D. Lenat, and E. Larson, "A knowledge-based approach to network security: Applying cyc in the domain of network risk assessment," in *Proceedings of the 17th innovative applications of artificial intelligence conference*. AAAI Press AAAI Press / The MIT Press, 2005, pp. 1563–1568.

[12] T. Aslam, "A taxonomy of security faults in the unix operating system," Master's thesis, Purdue University, 1995.

[13] A. Lazarevic, A. Ozgur, L. Ertoz, J. Srivastava, and V. Kumar, "A comparative study of anomaly detection schemes in network intrusion detection," in *Proceedings of the Third SIAM International Conference on Data Mining*, 2003.

[14] S. A. Hofmeyr, S. Forrest, and A. Somayaji, "Intrusion detection using sequences of system calls," *Journal of Computer Security*, vol. 6, pp. 151–180, 1998.

[15] M. Desnoyers, "Low-impact operating system tracing," Ph.D. dissertation, Ecole Polytechnique de Montréal, December 2009, [Online]. Available: http://www.lttng.org/pub/thesis/desnoyers-dissertation-2009-12.pdf.

[16] A. S. Tanenbaum, *Modern Operating Systems (3rd Edition)*. Prentice Hall, 2007.

[17] R. Love, *Linux Kernel Development (3rd Edition)*. Addison-Wesley Professional, 2010.

[18] POSIX.1-2008, "The Open Group Base Specifications," Also published as IEEE Std 1003.1-2008, San Francisco, CA, Jul. 2008. [Online]. Available: http://dx.doi.org/10.1109/IEEESTD.2008.4694976

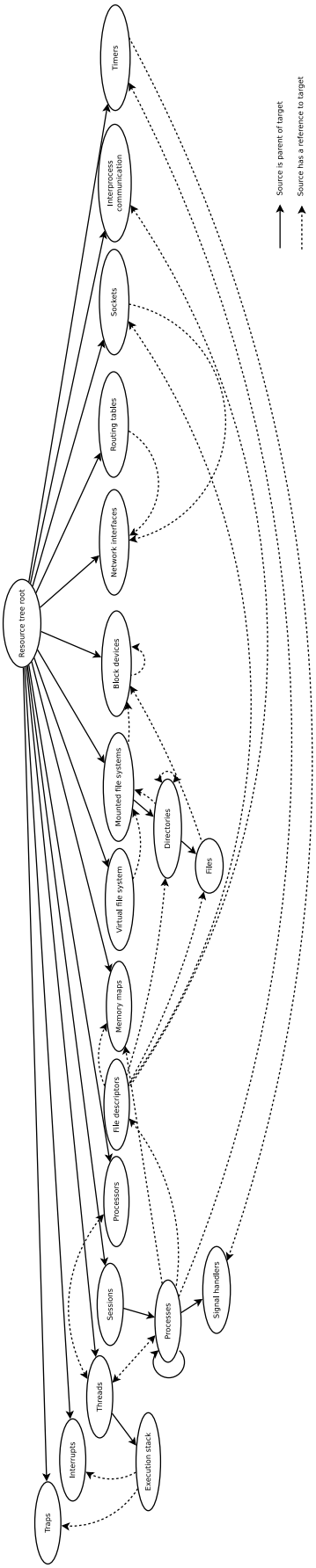

Figure 3. Operating system resources entity-relationship class graph