HolIDS: A Practical Approach to Holistic Host-based Intrusion Detection

Author Names and Affiliations protected for double blind review.

Abstract

Host-based intrusion detection systems that profile individual subsystems are limited by local or myopic views of the system's state. This limitation can lead to incorrect inferences of a system's intrusion status. HolIDS is a holistic approach to host-based intrusion detection and is governed by three principles that ensure both practicality and effectiveness. First, HolIDS monitors the global state of the system by auditing events produced by various subsytems (e.g., *process*, *memory*, *filesystem*, and *networking*). The architecture of HolIDS enables efficient correlation and integration of events within subsystems for describing the global state of the system. Second, HolIDS restricts the detection domain to root compromises *only* in order to streamline the task and decrease the runtime overhead; user-domain compromises are ignored. Third, HolIDS adopts a parallel approach to intrusion detection by combining an "at-entry" detector using vulnerability-exploiting, malicious-activity signatures, with "post-entry" detection using anomalous trigger events and validifying normal-activity signatures. An Linux implementation of HolIDS can detect intrusions that go undetected with existing subsystem-centric techniques as well as reduce the number of false-positives, while operating within acceptable, administrator-determined, overhead bounds.

1 Introduction

An intrusion is a sequence of operations that may lead to a deviation from the normal behavior of a computer system. Each operation occurs as a system call within a subsystem (e.g. networking, memory, filesystem, process, etc...) and can either be classified as malicious or a result of normal behavior. For example, at an abstract level, a buffer overflow exploit consists of two operations within different subsystems: (i) a vulnerable program receives malformed input from either the network or filesystem and (ii) writes the input data to memory. The write and read operations are involved in a sequence of events called an intrusion that can result in a system compromise.

Intrusion detection is the process of analyzing information generated from various computer subsystems to identify possible system compromises. There are two types of intrusion detection: "at-entry" signature-based detection and "post-entry" analomy-based detection. A signature-based intrusion detection system uses a defined set of vulnerability-specific signatures to detect known exploits while data is inserted at the entry of a system. An analomy-based intrusion detection system monitors deviations from an established baseline of system behavior "post-entry" which occurs after the system is compromised. An intrusion detection system (IDS) can either be network-based, host-based, or a combination (hybrid) [Prelude IDS]. IDSes are also classified as either passive or reactive. A passive IDS will log an intrusion and alert the administrator; whereas, a reactive IDS will log the intrusion and take an action such as terminating a socket connection or logging off a user. Examples of IDSes include: Tripwire, a passive host-based, signature-based, filesystem integrity IDS; Snort and Bro, signature-based, network IDSes; and StackGuard, an active analomy-based IDS.

Intrusion dectection solutions can be limited in scope and inferences. Host-based IDSes that monitor specific subsystems operate on local views of the system's state and may consequensly make incorrect inferences about the system's intrusion status. This behavior can cause IDSes to create unessary false positives. Signature-based "at-entry" IDSes are at a disadvantage when new and unknown vulnerabilities and superuser error (e.g., a misplaced root password used by an intruder to gain legal access) allow entry into the system; such limitations can lead to generation of a large number of false positives. False negatives and false positives are inaccurately generated due to IDSes inabilities to comprehend the variances within system, user, and superuser behavior models.

In this paper, we argue for monitoring the global state of the system to improve the accuracy of IDSes. Further, we argue for specific governing principles of our prototype IDS, HolIDS, that effectively address the challenges in building such a system. **[[Differentiate between the challenges and practical issues]]** These challenges include minimizing the system overhead associated with the information collection process, the scope and detail of the information, accurate correlation of the collected information within and across subsystems, and the efficient use of the such information in performing both *at-entry* exploit detection and *post-entry* anomaly detection. **[[the choice of target exploitation domain (i.e., user v/s root)]]**

1.1 Motivating Examples

Several real-world exlpoits and *post-intrusion* behavior of compromised systems prompted our work. We present two examples here.

Example 1.1 *A* SUID root *executable runs with root privileges regardless of the invoking user. If such an executable invokes the C-library routine* `system(cmd)` *without specifying full path to* `cmd` *(or alternatively invokes* `execlp` *which accepts non-fully-qualified executable pathname), a user can simply modify the PATH environment variable to run arbitrary code with root privileges. Earlier versions of the Mac OS X DirectoryService program had such a privilege escalation vulnerability [**?**]. Detection of intrusions that exploit this class of input validation errors is possible by monitoring the* process *subsystem for invocation of suid root executables by non-superusers followed by monitoring the same process for non-fully-qualified paths supplied to* `exec` *upon invocation.*

Example 1.2 *Rootkits typically modify system binaries (e.g.,* `ls, ps`*, etc.) to conceal their presence [**?**]. For example, modification of the* `/bin/ls` *root-domain file can be interpreted as a sign of a rootkit by file-integrity checkers such as Tripwire [**?**]. However, the RedHat package update,* `up2date` *executable also modifies system programs including* `/bin/ls`*. Differentiating between these two activities requires information and correlation of both* process *subsystem and* filesystem *activity. A process subsystem event (*`exec`*) determines the program which makes the modification and the filesystem event (*`write` *or* `unlink,create`*) determines the file being written to or replaced.*

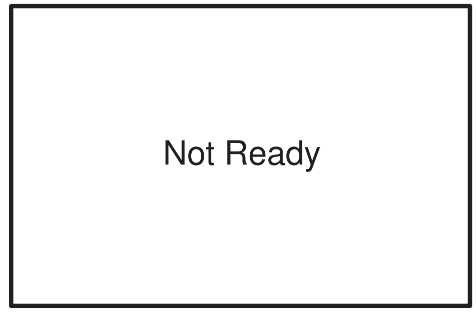

Figure 1: Global system state in HolIDS is obtained by correlating events occurring within the individual subsystems: process, memory, filesystem, and network.

1.2 Basic Approach of HolIDS

To model the global system state, we first identify four subsystems in a typical computer system: *process, memory, filesystem,* and *network* as shown in Figure 1. Critical system events can be classified as occurring within one of these subsystems. (We describe the concept of an *event* formally in Section 3.) Events are correlated in real-time based on the specific behaviors that HolIDS is configured to monitor. The central concept of a *process* and its associated unique *process id* in operating systems serves as the primary correlation coefficient. Secondary coefficients include the time of the event as well as the data associated with the arguments of the event. HolIDS collects information about all activity in the system, user and superuser, but detects root-domain intrusions alone for minimizing overhead while maximizing effectiveness.

To perform intrusion detection, HolIDS monitors the global system state for two distinct classes of behaviors corresponding to vulnerability-exploiting intrusive behavior and post-intrusion anomalous behavior. Activity signatures form the core elements for detecting both classes of behavior. (Section 3 discusses the activity signature concept in detail.) In addition to monitoring global system activity, post-entry anomaly detection also uses a database of potentially-anomalous events (e.g. modification of `/bin/ls`). Such occurences are validated against the current global system state.

Paper contributions:

1.
2.
3.

[[ROADMAP]]

2 HolIDS Design Principles and System Architecture

HolIDS

2.1 Design Principles

The design of HolIDS follows three key principles. These are:

A. Monitor global system state

B. Restrict detection to *root* exploitation domain

C. Employ both "at-entry" and "post-entry" detection

formalizes and extends the idea of tracking information flow [backtracking-paper, dataflow-anomaly paper]

[[Justify each of the principles]]

[[Principle B justification: Too many application-specific vulnerabilities New applications introduce new entry-points Users make mistakes: cannot be controlled. No uniform user behavior model (also application dependent) User compromise does not affect other users or system OS must implement fairness and it does (mostly) Significantly reduced signature set and detection overhead]]

2.2 System Architecture

[[System architecture. Finalize architecture figure and include. Discussion about subsystems, CIS (find better name), sensors.]]

In the HolIDS system, events may be obtained through three primary techniques: (i) system call interception, (ii) `/proc/` filesystem probing, and (iii) external tools such as Snort, Stackguard, etc.. This architecture allows the easy integration of external subsystem profilers into the HolIDS system by following clearly defined interfaces for *event escalation*.

3 Intrusion and Anomaly Detection

In this section, we first introduce the concepts of *events, activities,* and *signatures* and specific terminology that we shall use in the rest of the paper. Next, we describe the instrusion and anomaly detection techniques used in HolIDS.

3.1 Terminology

Definition 3.1 *An* **event** *is a* significant occurrence *within a single subsystem and due to a single process. An event description comprises of key pieces of information relating to the occurrence including time, subsystem, process-id, user-id, effective user-id, event-type, etc.*

An event description in HolIDS is represented using the following format:

```
[Tstart]:[Tend] [pid] [uid] [euid] [gid] [egid]
[event type] <event sub-type> [arguments]
```

Table 3.1 presents a sample of events that may occur within each subsystem. Figure 3.1 lists a few example events used in the prototype implementation of HolIDS.

Definition 3.2 *An* **activity** *is an partially ordered sequence of events. Useful activity monitored in HolIDS typically consists of a partially ordered sequence of* correlated *events.*

Definition 3.3 *An* **event signature** *is a generic event which does not include timestamp information and allows wildcards in various event description fields. An event signature in not an system occurence in that it does not actually occur in the system but rather, it can be used to represent a class of actual system events, irrespective of their time of occurrence.*

[[Talk about various wildcards, variables, and sets]]

Definition 3.4 *An* **activity signature** *is a generic activity, and can be simply thought of as a partially ordered sequence of event signatures.*

An example of an activity and corresponding activity signature that captures the behavorial aspects of the activity is shown Figure 3. **[[Explain how we got the signature from the specific activity.]]**

Subsystem	Sample events
Process	process created, executable loaded, EUID changed, process terminated
Memory	memory allocated/deallocated, protected memory overwritten
Filesystem	file opened, file permissions/owner changed, symlink created, file read/write performed
Network	TCP connection established, connection closed, connection bandwidth exceeded threshold

Table 1: Sample events for each subsystem.

```
21000:21019 P 13945 0 0 0 0 Exit <add-args>
19453:19454 P 26278 501 0 501 501 Exec /bin/ping 26240 <add-args>
40039:41001 F 25984 504 0 504 0 Access Write /bin/ls <add-args>
95233:92001 F 26295 0 0 0 0 Delete /var/log/messages <add-args>
19883:19889 N 26511 501 501 501 501 ConnectionEstablished TCP 131.94.130.192 <add-args>
65900:66005 N 26278 501 0 501 501 LoadSpike TCP 152.41.20.1 <add-args>
10001:10010 M 26401 303 303 303 303 Allocate 0xF04986FF 1024 <add-args>
89000:89001 M 10345 601 0 601 0 Free 0xFF98754A <add-args>
```

P: Process, F: Filesystem, N: Network, M: Memory

Figure 2: Sample events in the HolIDS prototype.

An Activity
`38450:38900 P 25631 0 0 0 0 Exec /usr/bin/up2date <add-args>`
`40039:41001 F 25631 0 0 0 0 Access Write /bin/ls <add-args>`
`45091:45092 P 25631 0 0 0 0 Exit <add-args>`

Corresponding Activity Signature
`P p u 0 ? ? Exec UpdateClass:U(P) <add-args>`
`F p u 0 ? ? Access Write ProtectedClass:P <add-args>`
`P p u 0 ? ? Exit <add-args>`

Figure 3: An example **activity** and corresponding **activity signature**.

We note that graphs are a more natural tool for modeling complex program behavior [] where multiple partial ordering of event signatures are allowable. To accomodate this complexity, we model such behavior using sets of activity signatures, where each signature captures one allowable partial ordering of event signatures. This simplification allows us to simplify the implementation complexity in the HolIDS system.

Definition 3.5 *A* **malicious activity signature** *captures the behavior of a system while a vulnerability is being exploited. This type of signature allows the detection of intrusions "at-entry" or at the time when a vulnerability is exploited.*

Definition 3.6 *A* **benign activity signature** *allows validating that ongoing activity matching this signature is non-malicious. This type of signature improves the detection accuracy of intrusions "post-entry" by reducing the number of false positives.*

Definition 3.7 *The* **anomalous events collection** *is a set of single-event entries that are apparently malicious by themselves. Eg: /bin/ls being modified.*

Definition 3.8 *A* **signatures database** *is a collection of activity signatures. There are two types of signatures databases,* malicious *and* benign.

3.1.1 Generating signatures

One of the key challenges in HolIDS is the generation and continuous updation of the signatures databases. The current HolIDS prototype consists of hand-coded signature sets for modeling both malicious and benign activity. However, we envision that the next version of the prototype can obtain automatically a large fraction of the benign activity signatures automatically. This can be accomplished by first running the system in a protected environment guaranteed to be free of malicious activity (similar to the techniques used in [] for learning normal program behavior) and monitoring for anomalous events. An automatic process for determining correlated events to form the activity and another process for deriving a generic signature that considers activity invariants can then be used.

To make our task of signature specification more efficient, we worked on classifying signatures and building generic classes of both vulnerabilities and benign activity sets. To obtain generic classes of software vulnerabilities, we used the classification proposed by Landwehr et. al. [?]. **[[Discuss class-genericity of malicious signatures, the set concept, etc.]]**

3.2 Two-level Detection

HolIDS employs a two-level approach by combining *at-entry intrusion detection* with *post-entry anomaly detection*. The at-entry detector flags intrusions at the time they exploit a system vulnerability to gain access, while the post-entry detector flags anomalous process and/or system behavor that deviates from known benign patterns. Both detection engines run continuously and in parallel. While the at-entry detector monitors all system activity including user and root, the post-entry detector only monitors activity of processes with effective UID root[1]

For monitoring the global system state continuously, HolIDS monitors and correlates events within the four subsystems. By observing ongoing system activity in an online fashion, HolIDS tracks relevant transitions in the global system state within the at-entry and post-entry detectors. The detectors themselves use associated signature databases to perform intrusion and anomaly detection respectively.

3.2.1 At-Entry Intrusion Detection

The *at-entry detector* focuses on monitoring the "entry points" for intrusions, which are the system vulnerabilities. To detect the exploitation of system vulnerabilities, it continuously monitors all events for matches with entries in the *malicious signatures database*. Any match found indicates an intrusion that will flag the system administrator.

To perform **[[types of vulnerabilities: Buffer overflows, Privilege escalation, Trojan horses, Race conditions, Covert and subliminal channels]]**

3.2.2 Post-Entry Anomaly Detection

The *post-entry* detector focuses on monitoring root activity alone in an effort to detect anomalous privileged-

[1] In Unix systems, superuser-created SUID binaries run with superuser privileges regardless of the invoking user.

mode behavior. For this, it monitors activity related to both root process as well as user processes running SUID binaries.

[[Rewrte:]] The post-entry detector uses the *bening signatures database* and the *anomalous events collection* to perform anomaly-based detection. The inner detector works by carrying out the following steps continuously for each event that it observes: (i) when an event is observed, advance state machines and/or create state machines (if required), (ii) check if the observed event is in the anomalous events collection; if no, exit (this is a benign event), else proceed, (iii) if there was a state transition in "atleast" one of the state machines due to this event, this event is benign because it is part of a benign activity signature; therefore, exit, (iv) this event is part of anomalous activity, flag as potential intrusion and notify administrator.

Further, feedback can be employed by backtracking intrusions [**?**] detected by the *post-entry* detector to improve the accuracy of the *at-entry* detector.

4 HolIDS System Implementation

HolIDS is deployed on an Intel x86 platform running Fedora Core 4 (Linux 2.6). Figure 4 shows the high-level architecture of this implementation that overviews the flow of information through the following four phases: first, a set of selected events in the system are *sensed* (steps 1-3 in the figure); second, the sensed events are *monitored* (steps 4-6); third, the monitored events are *analyzed* (steps 7-8); and fourth, the system administrator is *notified* of the status of the system (step 9).

During the *sensing phase*, selected system calls are intercepted by a dynamic instrumentation system, called Kprobe. The System Call Interceptor uses Kprobe to probe selected system-call invocations such as Exec in the kernel. It then creates the corresponding event using the information gathered at the interception point and logs these events into a FIFO queue (Event Buffer) to be retrieved later by the Syscall Sensor.

During the *monitoring phase*, the Syscall Sensor, which operates in user mode, retrieves the events from the queue. We note that it is essential to the accuracy of the HolIDS system that events in the buffer are retrieved before the buffer overflows; otherwise, some important events might be lost. Next, the events from the Syscall Sensor and potentially other third-party sensors are dispatched to one of the process, memory, networking, or file system sensors based upon the events' subsystem classification. The subsystem sensors may perform some functions (e.g., aggregation, filtering, and escalation) on the events in accordance to a policy before submitting them to the Central Analysis Engine.

During the *analysis phase*, the events will be analyzed in parallel by a number of analyzers. The current implementation of the HolIDS system includes the Root Misbehavior Detector and Root Penetration Detector. The Malicious Activity-Signatures and Benign Activity-Signatures contain malicious and benign activity signatures, correspondingly, which are represented as state machine objects that references the current state with a pointer to an event. The behavior of the activities in the system is maintained with hash tables. Initally, the subsystem and event type for the first event of each activity-signature is hashed. The values in the hash table contains a linked-list of all the associated state machines. As new events are being analyzed, state machines with matching first events will be instantiated with new values. Those already instantiated with the matching next event will be updated and linked to a new hash. The termination status of the state machine detemines the state of the system and (i.e., the detected malicious and anomalous activities) whether or not the system administrator is alerted.

5 Evaluation

[[Several evaluation dimensions: What signatures we have in database (good and bad), specific attack/non-attack descriptions, detection accuracy improvement (metric? maybe false+/- rates, maybe specific attacks/non-attacks listing), runtime overhead, others?]]

6 Related Work

[[Refer to vulnerability classification [?].]]

[[Refer to Porras,Ilgun work.]]

[[Refer to Jha/Giffin]]

Hofmeyr et. al. point out the importance of monitoring previliged process execution (rather than user process) execution by monitoring system call invocations [**?**]. Similar to their work, HolIDS restricts detection to root domain compromises, but we note the importance of

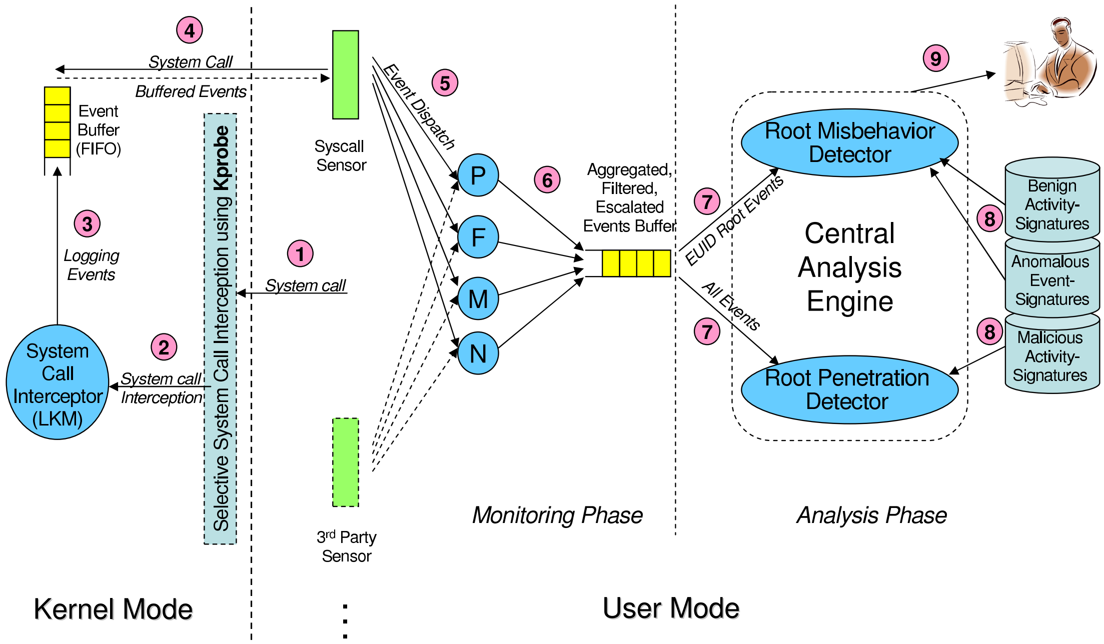

Figure 4: HolIDS system implementation architecture.

monitoring user domain activity in order to detect the privilege escalation points in user-level processes.

Sekar et. al. [?] proposed intrusion prevention systems that model and enforce correct program behavior through system call interception at program runtime. The high-level specifications for restricting program behavior are similar in spirit to the concept of anomalous event-set in our work. **[[Refer to vulnerability-specific predicates [?]]]**

Our work uses techniques similar to those used by Bhatkar et. al. in their recent dataflow anomaly detection work [?]. The authors model correct behavior of individual programs through system call interception in a sterile environment, combined with run-time validation of program behavior. In contrast, HolIDS allows combining the behavior of multiple interacting programs in an arbitrary manner. Further, it combines at-entry exploit detection and post-entry anomaly detection in a unifying framework.

[[Refer to backtracking work [?].]]

Researchers have investigated both static [?, ?] and dynamic [?, ?] analysis techniques for the detection of race condition (TOCTTOU) vulnerabilities in software. We leverage their findings to create generic signatures that allow detecting intrusions that exploit this class of vulnerability.

[[Refer to domain and type enforcement (DTE) [?].

Also, The Classes and Sets in good signatures are similar in function to capability-based systems [].]]

7 Conclusions and Future Work