NUCAR
Technical Report TR-01
January 2006

Hunting Trojan Horses

Version 1.0

Micha Moffie and David Kaeli

Hunting Trojan Horses

Micha Moffie and David Kaeli
Computer Architecture Research Laboratory
Northeastern University, Boston, MA
{mmoffie,kaeli}@ece.neu.edu

Abstract

In this report we present *HTH* (Hunting Trojan Horses), a security framework for detecting Trojan Horses and Backdoors. The framework is composed of two main parts: 1) *Harrier* – an application security monitor that performs run-time monitoring to dynamically collect execution-related data, and 2) *Secpert* – a security-specific Expert System based on CLIPS, which analyzes the events collected by Harrier.

Our main contributions to the security research are three-fold. First we identify common malicious behaviors, patterns, and characteristics of Trojan Horses and Backdoors. Second we develop a security policy that can identify such malicious behavior and open the door for effectively using expert systems to implement complex security policies. Third, we construct a prototype that successfully detects Trojan Horses and Backdoors.

1 Introduction

Computer attacks grew at an alarming rate in 2004 [26] and this rate is expected to rise. Additionally, zero-day attack exploits are already being sold on the black market. Cases in which malicious code was used for financial gain were reported in the 2005 Symantec Internet Security Threat Report [31] (for the first half of 2005).

Symantec also reports a rise in the occurrence of malicious code that exposes confidential information. They further report that this class of malicious code attack represents 74% of the top 50 code samples reported to Symantec in 2005 [31]. A key characteristic reported was that six out of the top ten spyware programs were bundled with other programs.

It is difficult to guard against malicious code that exposes confidential information or tampers with information. These exploits may take the form of Trojan Horses or Backdoors that are installed without the user's consent. Moreover, freshly authored malicious code (i.e., zero-day attacks) can go undetected by even the most up-to-date anti-virus programs. Since Trojan horses and Backdoors may have very little immediate impact on the normal operation of a system, they may go undetected for a significant period of time, allowing the attacker a large window of opportunity.

We have developed a security framework that can uncover Trojan Horses and Backdoors, and defend against harmful activity. In this report, we describe HTH, a run-time monitor that comes coupled with new security policy tools.

In the following section, we present the motivation for our work. We review related work in section 3. In section 4, we introduce our security policy. In section 5, we explore the design space for HTH and discuss some of the design tradeoffs made in HTH. Sections 6 and 7 delve into the design and implementation of Secpert and Harrier. We evaluate HTH effectiveness in section 8. Initial assessments of HTH performance are described in section 9. We conclude in section 10.

2 Security Exploits

2.1 Security Exploits Examples

To establish the basis and motivation for our work we present some real world examples of malicious code exploits.

1. PWSteal.Tarno.Q is a Trojan horse that logs passwords and information typed into web forms. The downloader portin of the Trojan arrives as an email attachment. When the attachment is executed, the main part of the Trojan is downloaded from a fixed location. The Trojan creates a file and registers it as a browser helper for Internet Explorer (IE). The helper object is executed every time IE runs. The Trojan monitors a predefined set of web pages (such as those that contain strings like: bank, cash, gold, etc.), captures keystrokes and web forms submitted. The Trojan stores the information in several predefined files. Then the Trojan sends a unique ID (of the compromised computer) to the attacker (using a predefined http address) and periodically sends the collected information to a predefined url. [30]
2. The Trojan.Lodeight.A code tries to install malicious code on the compromised computer and open a Backdoor. When this Trojan is executed, it connects to one of two predefined websites and downloads a remote file and executes it (the remote file may be a Beagle worm). Then this Trojan opens a Backdoor on a TCP port 1084. [30]
3. W32.Mytob.J@mm is a mass-mailing worm which includes a Backdoor. The worm sends itself via email and uses a remote buffer overflow to spread through the network.

   The worm copies itself to a system folder and modifies the registry such that the worm is executed every time Windows starts. It collects email addresses and sends itself to some of those addresses (according to predefined characteristics). The worm starts an FTP server, connects to one of two predefined IRC channels, and listens for commands that allow the attacker to download files, execute files, restart the system or run other IRC commands [30].
4. As part of an adware program, the Trojan.Vundo presents the user with pop-up advertisements. One component of Trojan.Vundo (HTML code that exploits a Microsoft internet vulnerability) tries to download and execute a downloader component. If successful, this downloader component will create an executable file and save it in one or more directories. In addition, the downloader component will download (from a specified IP address) an adware component of the Trojan and cause it to execute (this component is a dynamically linked library and is injected into different processes). It will also modify the Windows Registry to execute itself upon startup. Once executed, the Trojan will

degrade Windows performance by decreasing the amount of virtual memory available, as well as displaying advertisements on the infected machine [30].

5. The Windows-update.com is a fake web site that exploits an internet explorer vulnerability to download and install Trojan horses. Using a vulnerable version of IE to access the fake windows site may cause the following: 1) an executable will be downloaded and executed on the computer. 2) the executable will run and download configuration information from a predefined website (lol.ifud.cc/63.246.131.30). And 3) connect to a third web site and choose one of many unknown custom Trojan Horse programs to download (depending on the configuration downloaded) [9].
6. W32/MyDoom.B virus is an executable file that can infect a Windows system. When executed, the virus attempts to generate files and add entries to the Windows registry. The virus modifies the registry to execute itself (at log in time) and to reference a Backdoor component. In addition, the virus downloads and installs a Backdoor. The Backdoor component (ctfmon.dll) opens a TCP port and can accept commands, execute additional code, or act as a TCP proxy. (US-CERT Alert TA04-028A) [32].
7. The Phatbot Trojan can be controlled by an attacker on a remote site (using a p2p protocol). The Trojan has a large set of commands which can be executed. A few of these commands include: stealing CD keys, running a command using *system(..)*, displaying system information, executing file from an ftp url and killing a process [9].
8. The Trojan Horse version of the Sendmail Distribution contains malicious code that is executed during the process of building the software. The Trojan forks a process that connects to a fixed remote server on port 6667. The forked process allows an intruder to open a shell running as the user who built the Sendmail software (CERT Advisory CA-2002-28) [33].
9. A Trojan horse version of TCP Wrappers can provide root access to intruders who are initiating connections with a source port of 421. Also, upon compilation of the program, this Trojan horse sends email to an external address. The email includes information which can identify the site and the account that compiled the program. Specifically, the Trojan sends the information obtained from running the commands *whoami* and *uname -a* (CERT Advisory CA-1999-01) [33].

The examples above include very recent examples of Trojans and Backdoors. Next, we use these and other examples to characterize Trojan Horses and Backdoors and uncover their common execution patterns.

2.2 Trojan Horses and Backdoors Characteristics

If we study the set of the Trojan Horses and Backdoors just discussed (as well as others malicious code examples), we can detect several distinct characteristics and behaviors:

1. Executables are downloaded and executed without user intervention.
2. The malicious code may create and/or update files in the file system (possibly the Windows registry) with *fixed* (i.e., hard-coded) values.
3. The code initiates a connection to a *fixed* remote host. The code may then download executables/data or upload private information.

4. The code allows a remote user to initiate commands or control the execution of the local host.
5. The code may degrade computer performance.
6. The malicious code may execute only under specific conditions (e.g., execute only on a specific port number).

To characterize a Trojan Horse or a Backdoor, one must consider the environment in which these exploits operate. From an attacker's point of view, his malicious program - the Trojan Horse - is operating in an unfriendly environment. First, a user can not control the malicious code, nor can the attacker until a connection is made. This means that the malicious code must be self-contained, and it must execute without any guidance from the user or the attacker. Second, the user or an anti-virus program may try to terminate the program as soon as it is detected; it is therefore beneficial for the program to hide and disguise itself as well as conceal its execution.

The common execution patterns of Trojan Horses and Backdoors are summarized below:

1. The malicious code is executed without user intervention.
2. The malicious code may be directed by the remote attacker once a connection is established.
3. Resources used by the malicious code, such as file names and network addresses, are hard-coded in the binary.
4. OS resources (Processes, memory) used by the malicious code may be consumed for the purpose of degrading performance.

In Table 1, we summarize the execution patterns exhibited by the different malicious examples.

Exploit Name	No user intervention	Remotely directed	Hard-coded Resources	Degrading performance
PWSteal.Tarno.Q	√		√	
Trojan.Lodeight.A	√	√	√	
W32.Mytob.J@mm	√	√	√	
Trojan.Vundo	√		√	√
Windows-update.com	√		√	
W32/MyDoom.B	√	√	√	
Phatbot	√	√	√	
Sendmail Trojan	√		√	
TCP Wrappers Trojan	√		√	

Table 1: Execution patterns exhibited by malicious code.

Table 1 shows how similar Trojan Horses and Backdoors behave. Many of those characteristic are unique to Trojan Horses and Backdoors and are exploited to distinguish good from malicious behavior. These unique behavior patterns are used as a basis for our security policy.

2.3 HTH Objective

Our main objective in this work is to complement anti-virus softwares by targeting unknown and zero-day attacks. We are particulary focused on Trojan Horses and Backdoor attacks. We aim to correctly identify and thwart these attacks before any harm comes ot the system, as well as reduce the number of false positive that typically occur in many firewalling systems.

3 Related Work

There are a number of approaches that can be followed to reduce a system's security risk to intrusion from malicious code. However, no prior work has specifically targeted Trojan Horses or Backdoors.

In this section we review related work in several related areas, including static and dynamic information flow systems, intrusion detection systems, and isolation and confining systems. We also discuss prior work in machine learning and data mining approaches for security.

3.1 Information Flow Systems

Information flow security systems have focused on language-based and static analysis mechanisms [2, 20]. These systems only allow the programmer to specify the policy. This means that the user puts his trust in the code developer and is not able to enforce his own security policy.

In contrast, RIFLE is an architectural framework for user-centric information flow security. This system can track information flow in all programs. This equips the user (in contrast to the programmer) with a practical way of enforcing any information flow policy [34]. Information tracking can also be used to defeat malicious attacks by identifying spurious information flows and restricting their usage [29].

Perl introduced a new taint mode. This mode enables Perl's interpreter to track all user input data (which is tainted) and restrict the actions the Perl program is allowed to perform on that input [24].

Valgrind [21] has been used to rewrite the binary during runtime to dynamically check for overwrite attacks [23]. It was also used to detect undefined value errors at the bit level [28]. The authors add a shadow bit for every data bit, indicating if the bit is undefined, and instruments the data during value creation. The MIT DOG project [36] uses binary rewriting to track user input (tainted data) very efficiently. DOG introduces an average slow down of 5.5 times compared to native execution.

Run time information systems are becoming more prevalent in security systems. Many of those runtime systems specialize in tracking one source of data such as user input, and develop security policies for common exploits. In HTH we consider different sources of data and dynamically track all of them to support our policy.

3.2 Intrusion Detection Systems

Program shepherding was introduced by Kiriansky et al. [13] and is used to enforce a security policy. Program shepherding thwarts attacks that change the control flow (such as buffer

overflow attacks) by monitoring the program's dynamic control flow.

System call monitoring is often used to detect malicious code [15] [14] [8] [5] [27] [3]. Monitoring can be used to differentiate between normal behavior which was recorded beforehand, and anomalous behavior [15]. The history of access requests can be also be used to dynamically classify programs on-line and execute them with appropriate privileges [3].

Software wrappers can be used to detect and remedy system intrusions [14]. These wrappers are software layers that are dynamically inserted into the kernel, and that can selectively intercept and analyze system calls at runtime. Using software wrappers in the kernel can significantly reduce the performance overhead associated with profiling, but offer less information on the call compared to the detailed information available in user space.

Run time monitoring of untrusted helper applications was proposed by Goldberg et al. in [8]. The authors proposed to create a secure environment for untrusted helper applications by limiting program access to operating system resources.

Scott et al. developed a portable extensible framework for constructing a safe virtual execution system [27]. They demonstrated how to easily profile system calls and how a simple policy can be constructed. They present several policies that can track specific malicious behaviors.

Gap et al. [5] perform an analysis of many host-based anomaly detection systems. These systems monitor a process running a known program by tracking the system calls the process makes. They organize previously proposed solutions across three dimensions:

- Runtime information that the detector uses to check for anomalies. This includes system call number, as well as arguments and information extracted from the process address space, such as the program counter and return addresses.
- The atomic unit that the detector monitors, i.e. a single system call or a variable-length sequence of system calls.
- The history - the number of atomic units the detector remembers.

System call monitoring is so prevalent in Intrusion Detection Systems because it provides a lot of insight on program behavior. In addition, system call tracing can be done very efficiently and introduce limited overhead to program execution. As such system call monitoring is a key aspect of HTH.

3.3 Isolation and confining Systems

The Alcatraz system, presented in [16] is a isolation system that implements the idea of logical isolation. The actions of the program are invisible to the rest of the system until they are committed by the user. The Alcatraz system intercepts all operating system calls, and all file operations are redirected to a 'modification cache' that is invisible to the rest of the system.

Terra [6] is an architecture for trusted computing. Terra builds on a TVMM - a trusted virtual machine monitor - which allows terra to partition the platform in to multiple, isolated VMs (virtual machines). Each VM can be tailored to provide for a particular level of security and compatibility. This allows each application to run in its own VM, either as an "open box" VM with the semantics of a modern open platform, or as a "closed box" VM with dedicated, tamper-resistant, hardware accompanied by a tailored that can protect the privacy and integrity of its content.

Isolation and confining systems have the advantage of separating the execution affects of malicious code from the rest of the system. The main disadvantages of such approaches are several: Terra separates the execution into different virtual machines and thus, the sharing of data between several VM's may become more difficult and less intuitive. This problem does not exist is the Alcatraz system where the user can decide whether to commit the changes. However, if the user is to successfully identify malicious behavior he will need to be knowledgeable about the program's behavior. In other words, the user will need to be an expert.

3.4 Expert Systems

Expert systems such as CLIPS [12] [7], are designed to model human expertise and knowledge. They can be used to develop systems for diagnostics or consultation, and can eliminate the need for a human expert. Automatic intrusion detection systems can also benefit from an expert system tool. A case for using expert systems for emulating human security experts is presented by A. Chesla in [1].

Human security expert are very adaptive when analyzing new attacks and intrusions. An expert system will therefore need to be adaptive learn new security exploits. S. Wiriyacoonkasem et. al. [35] show how a neural network can be used to allow the expert system to learn from experience, and effectively improve the expert system.

Enhancing Intrusion Detection Systems with Expert Systems can improve the quality of the intrusions detected, i.e. it will be possible to detect more complex patterns which are currently detected by human security experts. In addition, the ability to model human knowledge may reduce the number of false positives as well as give advice to the non-expert user.

4 Security Policy

In this section we introduce our security policy. Based on the unique behavior of Trojan Horses and Backdoors described in section 2.2 we develop rules that determine which patterns are malicious. Our policy is composed of a set of rules, where each rule is designed to detect different type of malicious behavior. Our rules are designed to oversee different types of program behavior. We classify our rules into three categories where each category groups similar malicious execution patterns. Our categories include:

- Execution flow.
- Resource abuse.
- Information flow.

Each rule has an associated *severity* label which is assigned according to our confidence that the behavior detected is actually malicious. The label is intended as a additional guide to the user when he makes his decision to continue or kill the application. We distinguish three *severity* levels: Low, Medium and High. Low severity is used where our confidence that this code is malicious is low, Medium severity when our confidence is higher and High when we are most confident the code is malicious.

In the next sections, we give examples of our rules in each of the different categories. We do not present all the rules implemented but rather a representative set of rules.

4.1 Execution Flow

In our policy we monitor the execution flow of the program, which includes the invocation and execution of new processes. Our target is to detect malicious code being executed. It is more likely that a program is malicious if it (the program) is executing processes with hardcoded names present in the binary or if the process names originate from a socket (potentially a remote attacker). Our policy implements the following rules:

1. A rule that verifies the name of a newly created process is not hardcoded:

   if (new process) **and** (process name is hardcoded) **then**
   Warn user (Low);
   end if;

2. A rule that verifies that the name of a new process does not originate from a socket[1]:

   if (new process) **and** (process name originated from a socket) **then**
   Warn user (High);
   end if;

3. A rule that verifies the name of a newly created process is not hardcoded, and that this code is infrequent:

   if (new process) **and** (process name is hardcoded) **and**
   (code frequency is low) **and** (program started a while ago) **then**
   Warn user (Medium);
   end if;

Our policy takes into account how often code is being executed. If a code segment is rarely executed it may *reinforce* the suspicion of the presence of malicious code. Rarely can be defined as once during an execution or once across multiple executions. For example, malicious code such as the CIH/Chernobyl Virus execute on predefined dates in the year (CERT IN-99-03) [33]. In our policy, we increase the severity level from Low to Medium when a program name is hardcoded and this program is being executed rarely.

Our policy takes into account the origin of the program name. When the program name is hardcoded, we have less confidence in our warning since this may also occurs in trusted programs, therefore the severity level is Low. On the other hand, when the program name originates from a socket, we have more confidence that the program is malicious and label the warning High.

In the next section we present rules to counter Resource abuse.

[1] In our policy we assign a high warning to process names that originate from a socket. Future implementations may check if the socket name was hardcoded or provided by the user. This distinction is already made by other rules in our policy.

4.2 Resource Abuse

Resource abuse includes allocating and using different resources from the operating system with the purpose of draining the OS resources and impacting performance. Examples of resource abuse include:

- Executing numerous new processes
- Allocating a large amount of memory (such as the malicious code Trojan.Vundo does, as described above).

In our policy we monitor the number of new processes created, as well as the rate of creation of new processes. Our policy implements the following rules:

1. A rule that tracks the number of a newly created processes:

   if (new process was created) **and** (number of new processes created is high) **then**
   Warn user (Low);
   end if;

2. A rule that tracks the rate of newly created processes:

   if (new process was created) **and** (the rate of new created processes is high) **then**
   Warn user (Medium);
   end if;

In our policy, we monitor the creation and execution of new processes. We are more confident when the rate of the creation of new process is high and therefore assign it a higher severity label. We leave other types of resource management or resource abuse (e.g., memory allocation) to future implementations of our system.

In the next section we present rules that monitor the information flow.

4.3 Information Flow

Information flow includes the flow of information between the following different sources and targets:

- The user input (information source only).
- OS files and sockets (information source and target).
- The program binary (information source).
- The hardware (information source[2]).

We elaborate on the different data sources in section 5.1.

Next, we present several rules implemented in our policy:

[2]In current implementations, hardware is only a source of information, future implementation may include instructions that can write information to the hardware.

1. A set of rules that alerts the user when information is flowing from a file to a socket (Note, that the both the file name and the socket address May be hardcoded or given by the user.):

   if (information source is a file) **and** (information target is a socket) **then**

   if (user gave file name) **and** (hardcoded socket address) **then**
   Warn user (Low);
   end if;

   if (hardcoded file name) **and** (user gave socket address) **then**
   Warn user (Low);
   end if;

   if (hardcoded file name) **and** (hardcoded socket address) **then**
   Warn user (High);
   end if;

   end if;

2. A rule that notifies the user when information is flowing from hardware to a hardcoded file:

   if (information source is hardware) **and** (information target is a file) **and**
   (file name is hardcoded) **then**
   Warn user (High);
   end if;

In our policy, several more rules are implemented. Those rules are very similar to the ones presented above, their sources and/or targets are different and are therefore not presented here.

5 HTH Design

To be able to implement the policy described above and identify the different types of malicious behavior, we need to track and analyze dynamic program behavior. We separate the analysis and policy implementation from the tracking mechanism to allow for a flexible and independent development of each component. Figure 1 shows the HTH high-level software architecture. In sections 6 and 7 we describe the design and implementation of each component.

Next, we elaborate on the different data sources that need to be tracked for information flow. Then we discuss the design space for our implementation of the tracking mechanism.

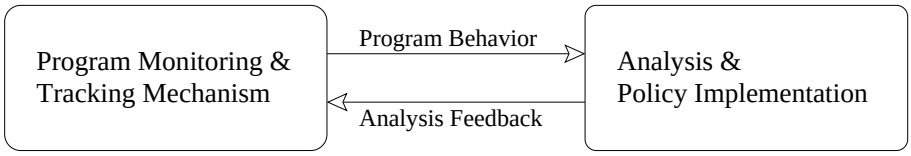

Figure 1: HTH software architecture

5.1 Data Sources

Our policy rules take into account the source of the data. We maintain enough information about each data source to enable our policy to make fine-grained distinctions. For example, we maintain more than just one 'Taint' bit. A single Taint bit only allows us to distinguish between two different data sources For example, a single bit can indicate whether the data was input to the program or not [24, 36, 23]), or if the data was defined or undefined [28].

We are interested in maintaining additional information. In particular, we would like to know the type and name of each resource. The following resource types (for data sources) are defined to support our policy:

- USER_INPUT
- FILE
- SOCKET
- BINARY
- HARDWARE

The USER_INPUT, FILE and SOCKET data sources types are self explanatory. The BINARY data source type is used to find hardcoded values. When the program itself or shared libraries are being loaded, the corresponding memory addresses are tagged BINARY.

The HARDWARE data source is used to tag data that originated from hardware. An example of this is the X86 *cpuid* instruction which stores processor identification information in the %EAX, %EBX, %ECX and %EDX registers. Although this is a simple example, future processors may hold more information in hardware. This information may include user secrets, hardware secrets, and information used for auditing.

We have several reasons for tracing the resource name:

1. it allows us to specify trusted resources (for example, trusted libraries such as libc.so),
2. we are able to give the user more information about the source or target of the information flow,
3. we are able to use the name during debugging [3].

The data sources are used for identifying the source of data. They are also used to identify the source of function arguments, such as strings or numbers. In particular, they can be used to identify the source of a resource name or address. For example, if a file is opened and the file name was hardcoded, the data source of the file name (the string itself) will be BINARY. If

[3]Note, for the security policy, it is only necessary to track the data source type and a minimal set of trusted resources.

the file name was given by the user, the data source would be USER_INPUT. In the rest of this report we will use resource ID (identifier) or origin to denote the resource file name or socket address, and use resource ID (origin) data source to denote the data source corresponding to the resource ID (origin).

Table 2 shows all the possible combinations of data sources and the resource ID (origin) data sources [4].

Data Source	*Resource ID*	Resource ID (Origin) Data Source
USER_INPUT	—	—
FILE	*File name*	USER_INPUT FILE SOCKET BINARY
SOCKET	*Socket name (address)*	USER_INPUT FILE SOCKET BINARY
BINARY	—	—
HARDWARE	—	—

Table 2: Data source combinations.

5.2 Design consideration and tradeoffs

In this section we motivate our design choices. We first describe several design alternatives and explain the tradeoffs associated with them. Next we examine which events we need to track to collect all the relevant information needed to support our policy and clarify the implications of tracking all those events. We end with presenting the design choices we made and the reasons for those choices.

5.2.1 Static vs. runtime behavior tracking

A program can be analyzed statically or dynamically. Static analysis is performed at compile time, link time or post link time. Static analysis does not introduce any runtime overhead. Run time tracking on the other hand may impose significant overhead, but furnishes the monitor with all runtime information not available statically.

Accurate runtime information can provide more information to the analysis and may lead to a more accurate policy. A static tool may not be able to discover all the code being executed, and thus, limit the effectiveness of the policy [5]. For example, dynamically linked libraries are only loaded at runtime and may not be available prior to execution. Another example, is self-modifying code which cannot be analyzed statically.

[4]Our prototype, as well as any incomplete implementation may need to consider an UNKNOWN data source as well

[5]If all dynamically linked libraries are trusted, this may be less of an issue.

Code execution profiles, as well as real time data from the user or the network (which can be monitored and analyzed), are only available at runtime.

A run time system, although slower, has a significant advantage of having all the run time data available to it. Having such rich data set may lead to a more accurate analysis and reduce the number of false positives.

5.2.2 Source code vs. binary

Analyzing a program for vulnerabilities can be done using several different methods. We can choose to analyze source code or binary code. One key difference is that source code analysis has the advantage of maintaining the high-level semantics of the program behavior. Binary analysis introduces a *semantic gap* between the low-level behavior that can be observed (e.g., assembly instructions) and the high-level behavior the program exhibits (e.g., method calls).

Since the source code maintains high level semantics, analyzing it to discover the program behavior may be more accurate. The main drawback however, is that the source code needs to be available. This is usually not the case.

5.2.3 Events monitored at different abstraction levels

There are numerous events that need to be monitored to accommodate the information we need for our policy. We divide the events into 3 categories:

- Architectural (ISA) events - (instructions executed),
- OS (API) events - system calls, and
- Library (API) events - library routines

Events such as OS and Library calls allow us to collect information related to the program semantics and program information flow. Architectural events allow us to collect information related to program information flow and program frequency.

Dividing these events into categories emphasizes the need to accommodate different levels of abstraction in our system.

5.3 Design choices

HTH is a runtime monitoring system. This will allow us to maintain runtime information and enable us to perform detailed and accurate behavior analysis. Future research will look into developing hybrid approaches in which static analysis may be used to accelerate the runtime monitor.

In this initial implementation, our goal was to keep the monitor as lean, generic, and general as possible. Analyzing source code would limit our analysis to one particular language or, would require a specialized front-end for each language. Moreover, the source code may frequently be unavailable. Therefore HTH analyzes the program binary. This alternative does tie us to a specific architecture and OS (an executable format may bound us to a specific OS), though we have started to think about how to design this for portability.

Even though we could potentially track all events at every abstraction level (architectural, OS and library), we will benefit if we can reduce the number of events monitored to a generic

and preferably small number of events. Tracking all the libraries API may very well be intractable. Since HTH analyzes the program binary and has no access to the source code, we are only able to monitor calls that are made to shared objects. We do not assume that debug information is available in the binary and thus restrict ourselves to shared objects with a defined API can be monitored.

HTH will monitor architectural and OS events, as well as track selected library calls. The main reason for tracking a subset of library calls is to overcome the semantic gap introduced by working with only architectural and OS events.

5.3.1 HTH classification

HTH falls into the class of program monitors called *white box detectors*, as defined by Gao et al. [5]. A white box detector is a system that uses all information available to it including:

- system calls and system call arguments,
- program memory usage and,
- source code and binary code.

HTH is chosen to be a run time security monitor. We believe that this alternative is the most flexible and applicable for most users. In addition we believe that run time analysis can provide the best accuracy and minimize the number of false positives. Future work will consider the run time implications of such a system.

In the next sections we pesent Secpert the expert system which is responsible for analyzing the program behavior and Harrier which is the run time monitor which tracks all the events.

6 Secpert Design and Implementation

Secpert (Security expert) is the HTH component responsible for analyzing program behavior and implementing the policy. It is implemented as an expert system. In figure 2 we show a high level view of Secpert Expert system architecture.

Secpert is driven by program events. The events are used to analyze a program's behavior. Based on the policy adopted, the runtime behavior is monitored and a warning will be issued to the user if an exploit is detected.

We first describe the events Secpert will address and then describe our implementation using the CLIPS expert system.

6.1 Secpert Events

To implement the policy we have described early, Secpert is notified whenever an event occurs. To keep Secpert running efficiently, we only notify Secpert on predefined events and attach all relevant information to those event.

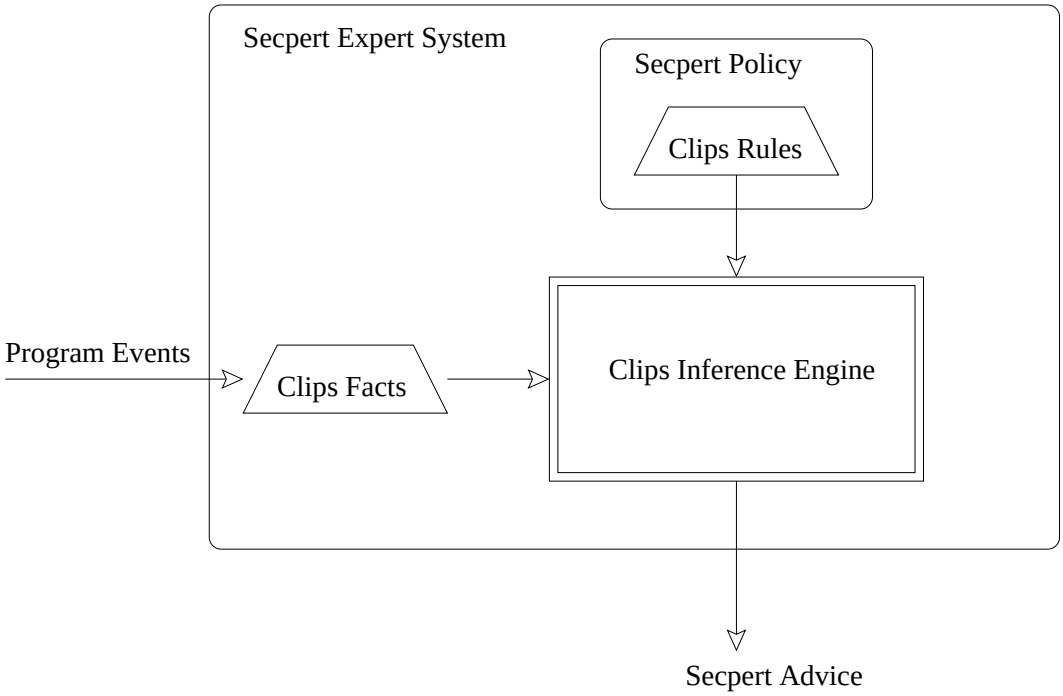

Figure 2: Secpert Expert System architecture

6.1.1 Events timing

In our current implementation, Secpert is notified whenever a relevant system call, socket call or library call is about to be executed. These types of events allow Secpert to focus on program behavior and mask irrelevant details (such as data flow). Since these events are usually rare, we should be able to analyze the event without a significant impact on performance (Secpert should never become a performance bottle-neck).

6.1.2 Events types

We categorize *interesting* calls into two types:

1. resource access, and
2. write to or read from a resource.

We attach all relevant information to each of these events. The information attached to a *resource access* includes the call number, resource name and type, as well as the resource ID (name) itself along with the associated resource ID data source. In addition, we also collect the time, code frequency, and code address.

The information attached to a *write to or read from resource* includes the call number, and information about the source of the data. This source information includes the source resource name and type, as well as the source resource ID data source. Similar information is sent regarding the target, including the target name and type, as well as information about the target name itself (i.e., the target resource ID data source). For target, we also provide the time, code frequency, and code address information.

6.2 Secpert Implementation

We have implemented our policy using the CLIPS expert system [7][12].

6.2.1 CLIPS

CLIPS is a tool that is used to build expert systems. It was designed to allow the user to develop software (expert system) that can represent human expertise and knowledge.

CLIPS can represents two types of knowledge: *heuristic knowledge* and *procedural knowledge*. Heuristic security knowledge is based upon a security expert experience. This knowledge can be represented by rules; mainly conditional if-then-else clauses. Procedural knowledge can be represented in CLIPS by functions, generic functions, and object oriented programming and is used to represent more *algorithmic* knowledge.

CLIPS has 3 main components [7]:

- Rules - The set of if-then-else statements which represent the (human) expert's knowledge.
- Facts - The data entered by the user to the expert system. This data represents all the information the user provides to the human expert.
- CLIPS inference engine - The engine that controls the execution of the rules.

CLIPS programming is data-driven. The inference engine will execute the rules according to the data, (or facts) provided. The CLIPS inference engine will choose to execute (or fire) the rule corresponding the the associated "if" condition that is satisfied. There may be more than one rule that will be fired. Note, that data is essential for the execution of the rules, and without the data the rules will not be executed.

The main advantage of using an expert system over other approaches (e.g., neural networks) is because an expert system has the ability to reason about its decision making. An expert system can give the user all of the information that was used to reach its conclusion.

6.2.2 Secpert - The Security Expert

Implementing Secpert using CLIPS consisted of implementing both the facts and the rules. The facts are asserted (inserted into the system) on predefined calls. During each of the events (as described 6.1.1 and 6.1.2) a new fact was created and inserted into the system. Each new fact contains all event-relevant data. An example of a fact being asserted in CLIPS is shown in appendix A.1.

We have implemented all the rules described in section 4 as CLIPS rules. An example implementation of an implemented rule is shown in appendix A.2.

When a rule is fired, it's output contains enough information for the user to understand the warning. An example of a fact being asserted in CLIPS is shown in appendix A.3.

7 Harrier Design and Implementation

Harrier is the HTH component responsible for tracking and monitoring program execution. The component is called Harrier after the *Northern Harrier* which systematically searches for prey (see Appendix C).

Harrier tracks the program's execution and generates events that are sent to Secpert for analysis. Harrier will generate events to support execution flow tracking, resource tracking, and information flow monitoring.

We start by describing the events tracked by Harrier, including the system calls monitored, the source/target of the data flow, the library calls (socket API), the architectural events monitored, and the code frequency. We then describe our implementationt.

7.1 System calls

Harrier tracks multiple system calls for several different reasons. We track the `execve` system call to facilitate execution flow. We also track the `clone` system call to enable resource abuse analysis. Whenever such a system call is issued, and just before it is executed, an event is generated and sent to Secpert. Harrier will interrupt the execution of the program and wait until Secpert analysis is done.

System calls such as `open`, `close`, `write` and `read` are tracked also. We track the `open` and `close` system calls in order to match future `reads` and writes to the same resource. In addition tracking these system calls allows us to find the data source of the resource id (source name).

Additional system calls such as `create`, `dup` and `socketcall` are tracked as well. When these system calls are executed, Harrier will generate events just as it does for `execve` and `clone`. But since Harrier is a prototype, not every system call is tracked.

7.1.1 Sources and targets of data flow

The `read` and `write` system calls represent entry point for information flow (read) as well as exit points (write). These system calls are tracked because they represent the source and target of the information.

Tracking these system calls is not enough to implement full information flow tracking, but is an essential step. When data is being read from a file or socket and stored in memory, Harrier will tag that data with the appropriate data source. This tag will be referred to whenever the data is used. For example, when data is being written to a file or a socket the tag of the data (i.e., the data source) is sent to Secpert for analysis.

Tracking system calls is key to understand the program behavior. System calls are used to monitor program execution flow and resource usage. In addition some of the system calls are entry and exit points for the data flowing in the program. Although much of the program behavior can be extracted from the system calls, some of the behavior cannot. In the next section we present the motivation for tracking Library calls, and show how we can overcome the semantic gap introduced by working with system calls.

7.2 Library calls

Harrier tracks the `socketcall` system call and several of its sub calls such as `socket`, `bind`, `connect` and others. Several of the tasks that need to be done in order to open a socket (for instance a client) include calling additional routines that are implemented as a library API.

These routines include: *gethostbyaddr* and *gethostbyname*. These routines are important to track because they can tie the resource id (the socket name) to the connect call. In other words, the routines are used to track the data source of the resource id (socket name or address) from its origin to its usage in the `connect` system call.

As an example, an ordinary client may follow the following steps to connect to a server:

1. Open a socket, using the `socket` (sub) system call
2. Use the *gethostbyname* routine to get the host network address
3. Connect to the `server` socket using the host network address

Now, assume the server address is hardcoded. For instance if the client attempts to connects to `pop.mail.yahoo.com`, this string will be hardcoded and will be used as an input to the *gethostbyname* routine. Next, the host network address (which `pop.mail.yahoo.com` symbolizes) can be used to in the connect system call[6].

Our goal is to convey to Secpert the information that the string `pop.mail.yahoo.com` (the resource ID) was hardcoded. This will be done when the `connect` system call is executed. The `connect` call is being tracked by Harrier, since the network host address is one of its arguments, the data source of that argument can be found and sent to Secpert. The problem arises when we consider the source id of the host network address. We would like the source id to reflect the source id of the `pop.mail.yahoo.com` string, but in reality the network host address originates from the *gethostbyname* routine.

The *gethostbyname* routine resolves the host name to a network address. The resolution may be done using a local host file, a domain name server, or other methods. The network address returned may originate for example, from the local host file or from the domain name server.

Tracking data flow through any function which performs a translation will fail to tag the translated data with the same tag as the input data. Instead, the translated data will be tagged with its own new source. If the translation table resides in a binary, the tag of the translated data will be BINARY. If the translation table resides on a remote server, the translated data will hold the tag of the server (since the origin of the translated data originated from the server).

Our solution is to identify the *gethostbyname* routine and *short circuit* the data flow. Essentially, we will tie the hardcoded address in the example with that of the resulting host network address. We can do this by treating the routine as an atomic operation and copy the resource ID tag directly to the resulting host network address.

7.2.1 Bridging the Semantic Gap

The example shown above shows that occasionally the information flow is *outside* of the program and we need a different approach to profile it. We would run into similar problem if we wanted to profile the setting and reading of environment variables. Although we would prefer only to track system calls and keep the implementation simple and generic, this type of information flow can not be obtained through monitoring system calls alone.

[6]We ignore a step that may include a copy from the hostent struct to a sockaddr struct since in this step data being copied from one location in memory to another. This will be handled by the regular data flow mechanisms.

The semantic gap between the socket library API and the system calls is too wide. A new mechanism must be introduced to track the information flow at the appropriate abstraction level. For this reason we monitor a minimal set of library API functions.

7.2.2 Constraints imposed by tracking library API

Tracking library APIs may place some restrictions on Harrier. These limitations are imposed because we may need extra information in the binary to locate these routines. Usually the libraries are implemented as shared objects and the routines are therefore easy to find.

A problem may arise when the program is statically compiled without any debug information. Luckily, in the case of the *gethostbyname* routine, this should never occur. Trying to statically compile a client with the *gethostbyname* routine produces the following warning (gcc 3.4.2):
`warning: Using 'gethostbyaddr' in statically linked applications requires at runtime the shared libraries from the glibc version used for linking`.
This restriction is present to ensure that the program uses the local methods for host resolution.

7.3 Data flow

7.3.1 Data flow & Architectural events

Harrier monitors a program's architectural events to facilitate information flow tracking. Architectural events - such as instructions executed - generate values and move data to and from registers and memory. Harrier tags each register and memory location with one or more data sources. Note that a tag may contain multiple data sources - this indicates that the data is produced by multiple input operands originating from different sources.

Whenever an data producing instruction is executed, the tag of the newly assigned memory or register is updated. The new tag will hold a composite of the data sources of the source operands. The following are a few examples:

- The `mov %esp,%ebp` instruction moves data from the %esp register to the %ebp register. In this case the data sources of %esp will be assigned to be those of %ebp as well.
- The `movl $0x4, mem` instruction moves the immediate 4 to a memory location. In this case the data sources of the memory location are assigned the data sources of the immediate. In reality, the immediate has only one data source - BINARY, and this data source will be assigned to mem's memory location.
- The `add %ebx, %eax` instruction adds the %ebx and %eax registers and stores the result in the %eax register. In this case, the data sources of %eax will include all the data sources of both %ebx and %eax registers. (In effect, the resulting set of data sources will be the union of the two sets of data sources, where both %ebx and %eax have each a set of data sources.)
- The `cpuid` instruction assigns %eax, %ebx, %ecx and %edx a value (part of the cupid). Here, each of the registers will be assigned the HARDWARE data source

Harrier tracks instructions executed[7] and continuously updates the tags and data sources. This allows Harrier to track all the data flow inside the program, as well as data flow from the hardware. When a system call is made, the data sources of the arguments, as well as of the data itself is passed to Secpert for analysis.

HERE

7.3.2 Data flow & Loader events

Part of data flow tracking includes fining the hardcoded data. This data is embedded in the binary - the executable itself or libraries that are dynamically loaded.

Harrier monitors events where a binary (the executable or shared library) is loaded. In effect Harrier carries out a very similar operation to the operation it executes on a read system call. When the data is being read from a binary and mapped to memory, Harrier will tag that data with the BINARY data source.

Note, that the loader event can be classified as an OS or system call event. Whether this is implemented as an OS event or in a separate manner depends on the implementation (of Harrier). We emphasize this case since it is a special case of a file read and we need to carry out a slightly different operation.

7.3.3 Data flow & User input

User input needs to be tracked by Harrier. The read system call, using the STDIN file descriptor will read the 'regular' user input into a memory location. Harrier will identify this case and tag the memory location with the USER_INPUT data source.

Other channels of user input to the program may be the command line argument as well as environment and auxiliary variables (argc,argv,env,aux). All this information is stored on the initial stack as the program starts its execution. Harrier will tag all the initial stack with the USER_INPUT data source.

To summarize, Harrier tracks data flow and considers the following aspects of data flow:

- Data flow inside the program - instruction that read/write memory and registers
- Special instructions that read data from (and potentially write to) the hardware (CPU). CPUID is a current example. Attestation may be a future usage.
- Loaded binaries and shared libraries need to be tagged, since they may be a source of data.
- The initialized stack, which contains the command line arguments, the environment and aux variables are tagged.
- System calls that write data to or read data from memory are be handled as well.

[7] harrier is a prototype and as such does not track all X86 instructions. For example, Harrier does not track floating-point instructions at all. We also do not track control transfer instructions (jump, branch, etc..), although these instructions may allow implicit information flow. This is left for future work. Harrier does track enough information to provide significant protection, as the results show.

- Library calls where the data flows outside the scope of the program, are handled ('short circuit-ed').

7.4 Basic block frequency

In our policy we aim to distinguish code segments that are rarely executed. Such segments may be an indicator of a backdoor which may be seldom used. For example, the TCP Wrappers Trojan will provide root access to intruders who are connecting from a specific port, we assume that most of the time connections are 'normal', and relatively rarely are the intruder connections made. Another example of malicious code that is rarely executed is the CIH/Chernobyl Virus. This virus infects executables and is spread when an infected executable is executed. The CIH virus has several variants. Some are triggered every month on the 26th day, while other variants are triggered just on April 26th or June 26th. Once the CIH virus executes, it attempts to erase the entire hard drive and overwrite the system BIOS. (CIH affects only Windows 95/98 machines) (CERT IN-99-03) [33].

To support this, we need to count the code segments executed. This is done by counting *basic blocks*. A basic block (BB) is a sequence of instructions that end with a control transfer instruction - in other words, if the first instruction of the BB is executed, all other instructions in the the BB will be executed as well.

Harrier will count the number of times each BB was executed. Whenever an event is sent to Secpert, it will include the frequency count which represents the number of times the BB (which contains the event trigger) has executed.

Note that we are interested in the frequency of the application's code and not in code executed in trusted dynamic libraries (or shared objects). Assume for example an application that contains several functions that call the execve syscall and that one of those functions is malicious. Our goal is to differentiate the different functions and locate the one that exhibits the malicious behavior. when this application is executed, each call to execve will be diverted to a dynamically loaded library (libc.so), thus counting the frequency of the actual event - the execve system call - results in counting the BB of the shared library. This reduces the effectiveness of code frequency since we cannot distinguish different parts/functions of the original program.

To solve this problem, Harrier can track only the BB Frequency of the application itself – this is only done when the shared libraries are trusted. Harrier follows of the execution of BBs of the application itself and keeps track of the "last" BB executed in the application before entering the library code. This allows us to distinguish the code in shared objects from the original application and match the execve system call event to the application BB which originated the call path to the shared object. Figure 3 show a BB execution path being tracked by harrier. The figure shows the interface between the application and a shared object.

This scheme can be relatively easily overcome by an adversary, it is therefore used mainly to reinforce the warnings generated by Secpert.

7.5 Implementation

Harrier can be viewed as layer that virtualizes an application. At this level we can monitor all Library, OS and ISA abstraction levels as shown in figure 4. Harrier is implemented on

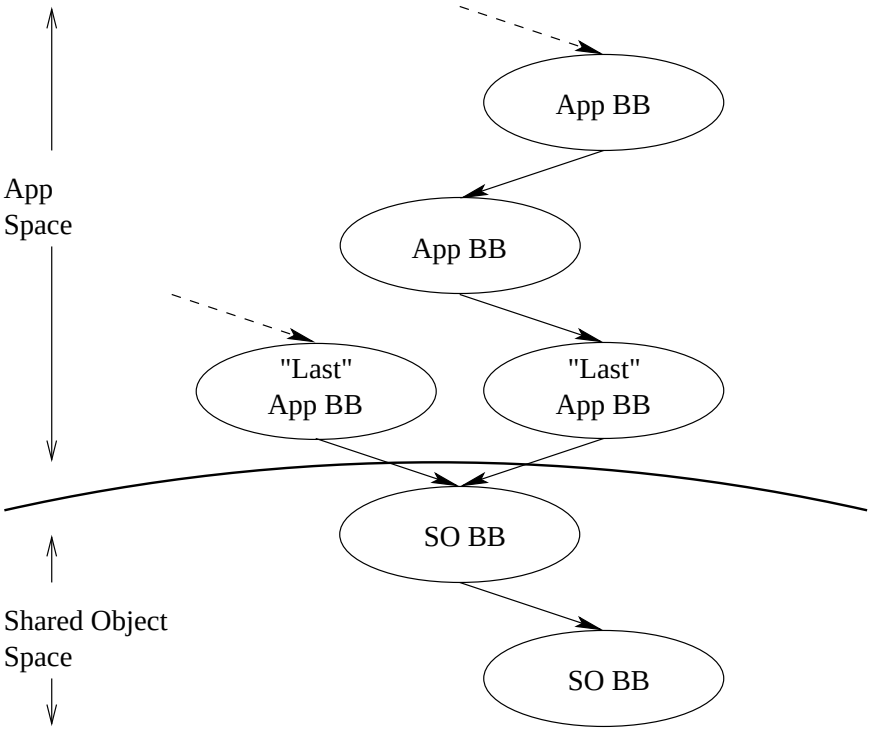

Figure 3: BB Execution path: from the application to a shared object

top of PIN [17]. Pin is a framework for dynamic instrumentation and supports Linux binary executables.

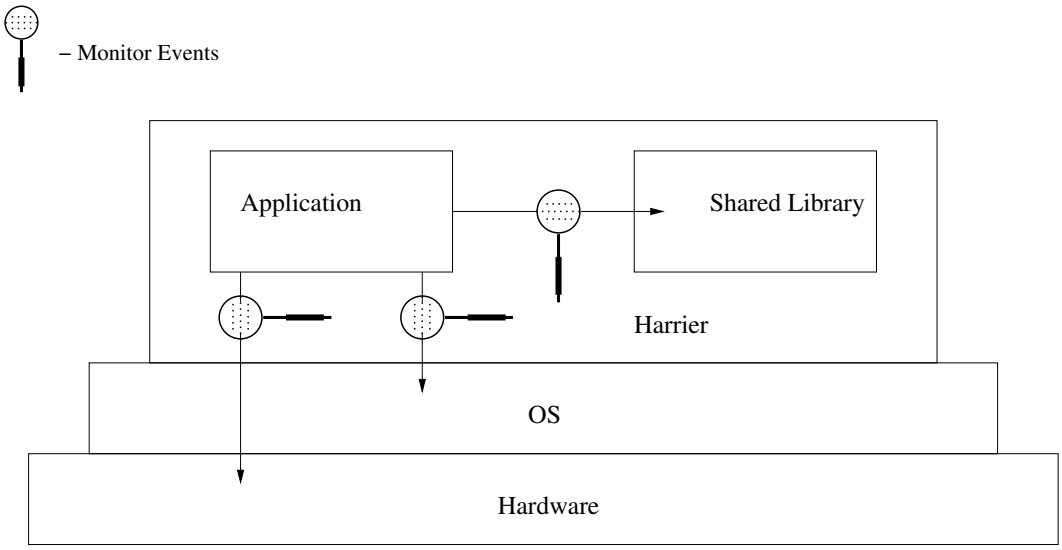

Figure 4: Harrier virtualizes the application execution

Dynamic instrumentation is done 'on top of' the operating system and 'below' the appli-

cation. It therefore allows us to access events in all the different abstraction levels. We are easily able to monitor libraries API, OS system calls and architectural events.

In order to support our policy, we need to track data flow, code frequency and monitor events such as system and library calls. To gather all this information we instrument the application in different granularities:

- Instruction,
- Basic Block - where we conceptually instrument on Basic Block boundaries,
- Routine - where we instrument on function call or return,
- (image) Section - where we identify the different sections in a binary, and
- Image - where we act when an binary is loaded or unloaded.

In table 3 we summarize for each policy rule which instrumentation granularity is used and what information is gathered. The table is divided according to the abstraction level in which the information is gathered.

Policy rule	Instrumentation granularity	Information gathered
Architectural events		
Information Flow	Instruction	Data Flow (reg/mem mem/mem, reg/reg)
Information Flow	Instruction	Hardware Information (CPUID)
Code Frequency	Basic Block	BB frequency
OS (API) events		
Execution Flow	Instruction	System Calls (execve)
Resource Abuse	Instruction	System Calls (clone)
Information Flow	Instruction	System Calls (IO read/write)
Information Flow	Section	Binary load
Information Flow	Image	Binary load
Information Flow	Instruction	Initial stack location
Library (API) events		
Information Flow	Routine	'Short Circuit' Data Flow (getHostByName)

Table 3: Information gathered in different instrumentation granularities

In figure 5 we show how Harrier uses pin to instrument the code. On the left side, the original code is shown in assembly. On the right side, we show how Pin instruments the original code and inserts new calls to the analysis functions Track_DataFlow, Collect_BB_Frequency and Monitor_SystemCalls. Note that the Track_DataFlow function is called before each instruction that moves or computes data, the Collect_BB_Frequecy function is called before each new basic blocks is executed and the Monitor_SystemCalls function is called before the system call (int 80 instruction).

Figure 6 shows the high level software architecture of Harrier. On the left side of the picture we show the main modules of Harrier instantiated from the 'main' in PinInstrumentor.

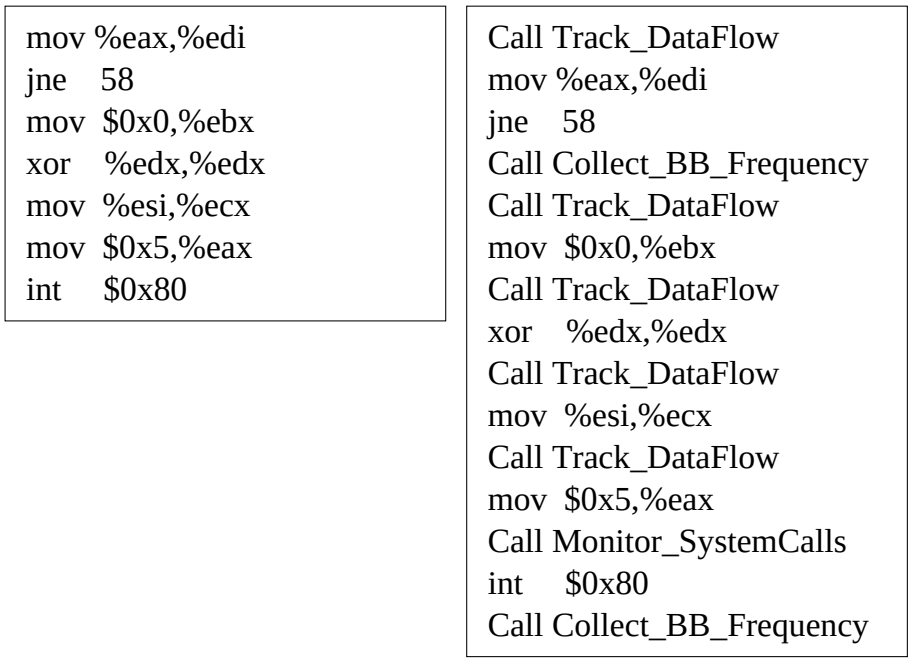

Figure 5: Harrier instrumentation example

Each of the modules instruments and handles a different aspect of the application (system calls, routines, instruction data flow, data flow and code executions). The figure shows the interaction between the different modules and the data structures used to track data flow and BB frequency.

The system calls are the events which we use to update Secpert with. The figure shows how the PinSysCallEventGenarator interacts with the EventAnalyzer. This module is responsible to format and send the events to Secpert (on the left side of the figure). It then waits for a response from Secpert and updates the user when necessary.

8 Security Accuracy Evaluation

8.1 Micro Benchmarks

In this section we present micro-benchmarks that are used to show all the different aspects of the policy as well as the implementation. The benchmarks test for malicious code as well as for trusted behavior.

8.1.1 Execution Flow

In table 4 we show our micro benchmarks for testing execution flow. The table shows results for 4 different benchmarks. All the benchmarks call *execve* with the program name originated from different sources: user input for the "User input" benchmark, hardcode program name for both the "Hardcode" and "Infrequent execve" benchmarks and a socket source for the "Remote execve".

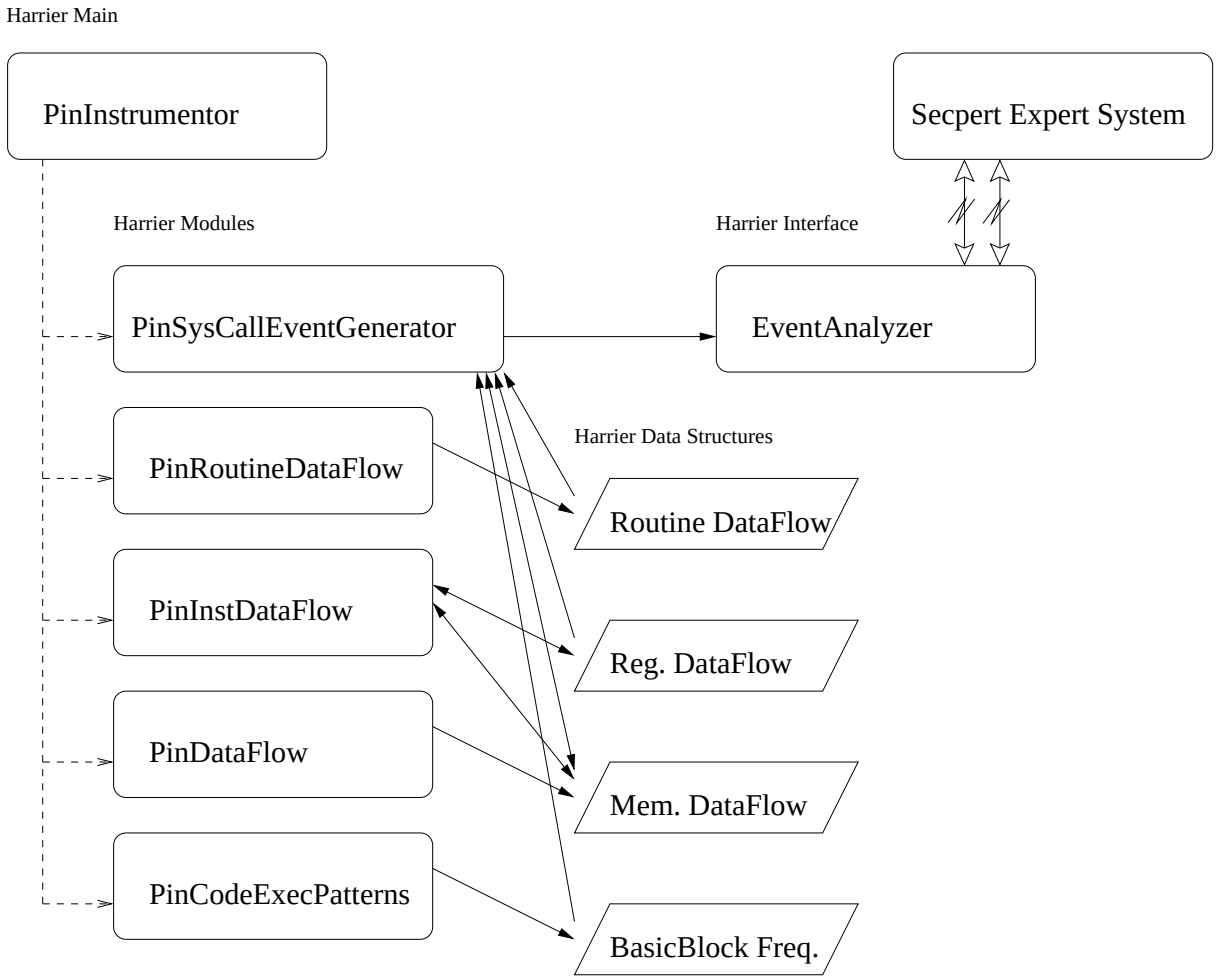

Figure 6: Harrier software architecture

Micro Benchmark	
execve	
User input	√
Hardcode	√
Remote execve	√
Infrequent execve	√
√	Correctly classified benchmark

Table 4: HTH Micro benchmarks - Execution Flow

The Infrequent execve benchmark is similar to the Hardcode, except the program is sleeping for a while to simulate a malicious code where the execve is executed infrequently. In the case of user input, HTH did not produce any warning and the benchmark was correctly classified as not malicious.

8.1.2 Resource Abuse

In table 5 we show our micro benchmarks for resource abuse. Both benchmarks frequently call *fork*. The loop forker has one main thread that calls fork. Each of the children executes an infinite loop and sleeps. The tree forker benchmark implements a simple loop where *fork* is being called and both the child and parent continue with the loop. This creates a tree of processes, where each process calls fork, produces 2 processes (parent and child) and both continue with the loop.

Micro Benchmark	
fork	
loop forker	√
tree forker	√
√	Correctly detected malicious behavior

Table 5: HTH Micro benchmarks - Resource Abuse

In both cases HTH can detect when the number of processes reach a certain threshold as well as a rate.

8.1.3 Information Flow

Table 6 shows the results for our information flow benchmarks. To test the information flow we consider several different sources, as discussed in section 5.1: Binary, File, Socket and Hardware.

In each section of the table, we evaluate benchmarks with different information flow sources and targets. In addition the benchmarks test different sources for the file names or socket names (source ID data sources). These may be hard-coded, given by the user or even obtained from a remote socket.

The benchmarks which use the Hardware source make use of the CPUID X86 instruction. All the benchmarks that use sockets where tested twice: once as a socket client and the other a socket server. Note, some of the tests show trusted behavior and are correctly classified.

8.2 Running trusted programs

In this section we evaluate several trusted programs. Our aim is to find out how often to we get false positives.

Some trusted programs that are not "well behaved" such as software update programs should be caught by HTH. In such a case the user is able to allow the operation to continue. A summary of the results are shown in table 7.

Next we describe each exploit and present the output of our tool.

8.2.1 ls

No warning is issued. Our tool does detect that "." is opened and the origin is binary (hard-coded).

Micro Benchmark	
Binary → File	
User filename	√
hardcode filename	√
remote filename	√
Binary → Socket	
User address	√
Hardcoded address	√
File → File	
User input, User Input	√
User input, Hardcoded	√
Hardcoded, User input	√
Hardcoded, Hardcoded	√
File → socket	
User input, User Input	√
User input, Hardcoded	√
Hardcoded, User input	√
Hardcoded, Hardcoded	√
Socket → File	
User input, User Input	√
User input, Hardcoded	√
Hardcoded, User input	√
Hardcoded, Hardcoded	√
Hardware → File	
User filename	√
Hardcode filename	√
√	Correctly classified benchmark

Table 6: HTH Micro benchmarks - Information Flow

8.2.2 column

In this test, we executed the 'column a b c' command. HTH did not produce any warning - as expected. Harrier did detect that the data printed to the screen originated from all three files (a b and c) and that those filenames are originated from the user (command line).

This test was the one that prompted the need to track the source of command line arguments. We needed to add the source 'USER_INPUT' to all command line arguments.

8.2.3 make

We executed make on the makefile used to compile Harrier.

Aplication Name	
ls	√
column	√
make	√
g++	√
awk	√
pico	√
tail	√
diff	√
wc	√
bc	√
xeyes	√

√	Correctly identified any good behavior
√	Partially or Inaccurately identified inappropriate behavior

Table 7: HTH Success in *not* warning when running well behaved programs

In our first test we executed 'make' when harrier was already made, so nothing should execute. No warning were issued from HTH. When running HTH, It found the open syscall of the 'makefile' file which was originated from /usr/bin/make, no warning was issued for this since nothing 'bad' was done with the content of this file.

When we tested 'make clean' (when the Harrier was build): HTH issued a warning [Low] for a hardcoded execve system call: '/bin/sh' was hardcoded.

When we executed 'make' (when Harrier was not built) HTH issued several warnings (Low) for executing g++. It identifies this commands as hardcoded as well as originated from the user. It seems that 'make' tries to find g++ in different directories and that those directories originates from the PATH environment variable (and therefore tagged as USER_INPUT).

8.2.4 g++

For this test we called g++ with two test files: (g++ test.cpp DataFlow.C) HTH issued warnings for executing a program that were hardcoded. The warnings ([Low]) were generated for executing 'cc1plus' and 'collect2' executables. (The 'as' executable was tracked as well, but its source ID data source was wrong).

In this case HTH generated a warning for a trusted application. This warning was generated since the policy specifies this behavior as malicious.

8.2.5 awk

To test awk we executed the following command: 'awk '/ifdef/' syscall_names.C'. HTH did not produce any warning. Harrier did detect that the data printed to the screen originated from the syscall_names.C file and that that file was given by the user.

8.2.6 pico

In this test we monitored the pico editor. Our test includes writing several characters (user input) and then saving the buffer to a new file called 'a.txt' (given by the user).

Harrier wrongly identified the source of the data as well as the source of the filename ('a.txt') and issued the following warning:

```
Warning [HIGH]   Found Write call to a.txt
    The Data written to this file is originated from the BINARY:("/usr/bin/pico")
    Moreover, it seems that the name of the file: a.txt originated
    from a BINARY: ("/usr/bin/pico")
```

This warning was generated because our prototype is incomplete. A complete dataflow tracking tool would correctly identify the data sources avoiding this warning all together.

8.2.7 tail

We also tested the tail utility. We have executed '/usr/bin/tail PinInstrumenter.C' and monitored it with HTH. Harrier detected that the file PinInstrumenter.C was given by the user and that the data printed to the screen originated from that file. HTH did not produce any warning.

8.2.8 diff

No warning is issued. Harrier detected that the output originated from both files and that the files were given by the user.

8.2.9 wc

We tested the wc utility. HTH detected that the output originated from the input file (as well as the binary) and that the input file was given by the user. HTH did not produce any warning.

8.2.10 bc

bc is a command line calculator. We used it simply to add two numbers. HTH did not issue any warning. Harrier did detect that some of the output originated from the user input (bc echo's the expression).

8.2.11 xeyes

When we tested xeyes, HTH generated several false warnings, Those include a write to a socket (on the local host), where the data originated from X11 libraries (libX11.so, libXrender.so), or where the data originated from binaries such as the xeyes binary itself or other hard coded binaries (xlcDef.so) All the warning generated were of Low severity.

False positives are bound to occur, HTH monitors the program behavior and basically tries to distinguish good from malicious behavior patterns. There is no accurate definition for good vs. malicious behavior: a malicious behavior for one user may behave as expected for another.

The current policy has strict rules since the policy assumes Secpert only monitors a single execution of the program. Some of the rules can be relaxed if Secpert would be extended to monitor multiple executions of the program as well as several programs. For example, if Secpert would monitor parent/child programs, the warning produced when g++ executed cc1plus and collect2 would be delayed to the point where they misbehave. If those executables would behave properly no rule would be fired at all. In other words, cc1plus and collect2 are part of the g++ program and are run as separate threads, if we would monitor all the related threads, Secpert would be able to make smarter decisions and not be limited by current implementation.

Future work will look at these, as well as additional ways to reduce the number of false positives.

8.3 Real Exploits / Proof of concept

In this section we evaluate several malicious programs found on the web. Several of these programs are a "proof of concept" exploits - they shows how to exploit a vulnerability.

A summary of the effectiveness of our policy is shown in table 8.

Exploit Name	
ElmExploit	√
nlspath	√
procex	√
grabem	√\
vixie crontab	√
pma	√
superforker	√

√	Identified malicious behavior
√\	Partially or Inaccurately identified behavior

Table 8: HTH Success detecting Real exploits

Next we describe each exploit and present the output of our tool.

8.3.1 ElmExploit

This exploit - Electronic Mail for UNIX (Elm) Expires Header Buffer Overflow Exploit [22] - creates an email and sends it to a given user.

Summary: Found the creation of the crafted email.

```
Warning [HIGH]   Found Write call to tmpmail
    The Data written to this file is originated from the
        BINARY:("/proj/arch1/mmoffie/PIN/RealExploits/a.out")
```

```
    Moreover, it seems that the name of the file: tmpmail originated from a
        BINARY: ("/proj/arch1/mmoffie/PIN/RealExploits/a.out")
```

HTH did not find the act of sending of the email. The exploit calls: `system("/bin/cat ./tmpmail | /usr/sbin/sendmail -t")` but HTH did not produce a warning for a hardcoded execve. This is because the system() function executes a command specified in the string by calling '/bin/sh -c' with the string. It then returns after the command has completed.

HTH does find the relevant event:

```
CLIPS_write: (assert (system_call_access
        (system_call_name SYS_execve)
        (resource_name "/bin/sh")
        (resource_type FILE)
        (resource_origin_name  "/lib/tls/libc.so.6")
        (resource_origin_type  BINARY)
        (time 108)
        (frequency 1)
        (address "80488e5")
```

but because Secpert trusts the libc.so library and the '/bin/sh' string data source is from libc.so the event is filtered out.

8.3.2 nlspath

The nlspath [4] vulnerability is an old exploit that uses the NLSPATH environment variable to exploit setuid root programs that are based on libc (libc version 5.2.18, 1997) and get root access. The exploit assigns the NLSPATH environment variables a string that was hardcoded in the exploit itself, it then executes '/bin/su'. This exploit did not give root access (on RedHat 9).

HTH found the call to execl with the hardcoded '/bin/su' string.

```
Warning [LOW]    Found SYS_execve call ("/bin/su")
    ("/bin/su") originated from ("/proj/arch1/mmoffie/PIN/RealExploits/a.out")
```

Future implementation should consider tracking data flow via environment variables similar to the way we track library socket calls.

8.3.3 procex

The procex [4] exploit can be used to disclose some information (contents of setuid application's environment data) to unprivileged users.

The exploits calls execl twice while reading from '/proc/pid/environ' (where pid stands for the process id). Executing this exploit although printing information did not appear to disclose any privileged information. HTH did find the two call to execl. Both are hardcoded:

```
Warning [LOW]    Found SYS_execve call ("/bin/ping")
    ("/bin/ping") originated from ("/proj/arch1/mmoffie/PIN/RealExploits/a.out")
Warning [LOW]    Found SYS_execve call ("/bin/ls")
    ("/bin/ls") originated from ("/proj/arch1/mmoffie/PIN/RealExploits/a.out")
```

8.3.4 grabem

grabem [10] is a simple program that can log user's password from the console. During grabem execution it asks for lab#, username and password and stores the user name and password in a file (that was created if didn't exist beforehand). The file name is .exrc%.

HTH detects that data is being written to the hardcoded file. It does not detect however that the data originated from the USER, Nor does Harrier detect the system() function call which turns the console echo on or off.

```
Warning [HIGH]   Found Write call to .exrc%
    The Data written to this file is originated from the
      BINARY:("/proj/arch1/mmoffie/PIN/RealExploits/a.out")
    Moreover, it seems that the name of the file: .exrc% originated from a
      BINARY: ("/proj/arch1/mmoffie/PIN/RealExploits/a.out")
```

8.3.5 Vixie crontab

Vixie crontab exploit [11] exploits a buffer overflow in crontab to get root. The exploit basically writes hardcoded data (code) into a file './Window' and executes '/usr/bin/crontab'. Running the exploit did not get root.

HTH detects both the data being written to the file and the execution of crontab:

```
Warning [HIGH]   Found Write call to ./Window
    The Data written to this file is originated from the
      BINARY:("/proj/arch1/mmoffie/PIN/RealExploits/a.out")
    Moreover, it seems that the name of the file: ./Window originated from a
      BINARY: ("/proj/arch1/mmoffie/PIN/RealExploits/a.out")
```

If we allow HTH to continue:

```
Warning [LOW]    Found SYS_execve call ("/usr/bin/crontab")
    ("/usr/bin/crontab") originated from
      ("/proj/arch1/mmoffie/PIN/RealExploits/a.out")
```

8.3.6 pma

pma [10] - Poor's Man's Access, is a daemon which lets a remote attacker issue shell command from remote.

We have compiled the daemon on linux and the attacker on a sun machine [8]. The attacker can connect and execute (remotly) shell commands such as ls, pwd cd and more. The server (the daemon) will create two special files using `("/bin/mknod %s p; /bin/mknod %s p")` where part of the name is hardcoded ("inpipe","outpipe") and part is the pid. It will then create a shell process (system) and redirect the shell's input and output to the new special files: `sprintf(sbuf, "csh -i <%s >%s 2>&1 &", iname, oname); system(sbuf);` It then

[8] We have reduced the number of threads in the pma daemon (to two) in order to simplify the monitoring. No other modification was made.

continues with two threads: one reads from the outpipe and writes to the socket (attacker) and the other reads from the socket (attacker) and writes to the pipein shell.

HTH monitors both threads simultaneously. HTH does not warn when the system() function call is called (libc.so is trusterd).

HTH monitors the daemon server, It detects the deamon openning a hardcoded socket server (it is hardcoded because we use LocalHost, the port is given by the user) HTH also detects the accept, the open of the in and out pipes (that were redirected to the shell - using system command)

HTH detects the daemon writing to the pipe a hardcoded prompt:

```
Warning [HIGH]   Found Write call to inpipe32425
    The Data written to this file is originated from the
      BINARY:("/proj/arch1/mmoffie/PIN/RealExploits/pma/pmad")
    Moreover, it seems that the name of the file: inpipe32425 originated from a
      BINARY: ("/proj/arch1/mmoffie/PIN/RealExploits/pma/pmad")
```

pma then compares the password given from the attacker and returns "ok" or "nope". In the first stage, a password is compared and if it is o.k. the deamon prints "echo ok" to the inpipe so that the outpipe will read "ok" (and this is what the attacker will see)

```
Warning [HIGH]   Found Write call to inpipe32425
    The Data written to this file is originated from the
      BINARY:("/proj/arch1/mmoffie/PIN/RealExploits/pma/pmad")
    Moreover, it seems that the name of the file: inpipe32425 originated from a
      BINARY: ("/proj/arch1/mmoffie/PIN/RealExploits/pma/pmad")

Warning [HIGH]       Found Write call
     Data Flowing From: outpipe32425
     To: gateway:36982 (AF_INET)
     source filename was hardcoded in:
       ("/proj/arch1/mmoffie/PIN/RealExploits/pma/pmad")
     This program has opened a socket for remote connections. i.e. it is a
       server with the address: LocalHost:11116 (AF_INET)
     the server address was hardcoded in:
       ("/proj/arch1/mmoffie/PIN/RealExploits/pma/pmad")
```

HTH then detects data flowing from sockets to inpipe and back from outpipe to socket (command, result).

In one of the tests we made, we tested the bb frequency:

```
Warning [HIGH]       Found Write call
     Data Flowing From: gateway:37047 (AF_INET)
     To: inpipe32666
     This program has opened a socket for remote connections. i.e. it is a
       server with the address: LocalHost:11111 (AF_INET)
     the server address was hard oded in:
```

```
      ("/proj/arch1/mmoffie/PIN/RealExploits/pma/pmad")
    target file-name was hardcoded in FILE:
      ("/proj/arch1/mmoffie/PIN/RealExploits/pma/pmad")
    This code is rarely executed...

Warning [HIGH]       Found Write call
    Data Flowing From: outpipe32666
    To: gateway:37047 (AF_INET)
    source filename was hardcoded in:
      ("/proj/arch1/mmoffie/PIN/RealExploits/pma/pmad")
    This program has opened a socket for remote connections. i.e. it is a
      server with the address: LocalHost:11111 (AF_INET)
    the server address was hardcoded in:
      ("/proj/arch1/mmoffie/PIN/RealExploits/pma/pmad")
    This code is rarely executed...
```

8.3.7 superforker

The superforker exploit [10] is a extreme version of the classic fork() denial of service attack. The program continuously forks processes as well as continuously opening files and writing garbage data to it.

Executing this exploit on the machine will consume all its processes extremely fast - the user will not be able to create any new processes any more.

HTH produce the following warnings for creating the hardcoded files, note the file names where created randomly (using characters that were hardcoded):

```
Warning [HIGH]   Found Write call to ..LDNwGMuqQoHTRcbYCeah...
    The Data written to this file is originated from the
      BINARY:("/proj/arch4/mmoffie/PIN/RealExploits/a.out")
    Moreover, it seems that the name of the file: ..LDNwGMuqQoHTRcbYCeah...
      originated from a BINARY: ("/proj/arch4/mmoffie/PIN/RealExploits/a.out")
```

HTH produced the following warnings when the number of processes increased:

```
Warning [LOW]    Found several SYS_clone calls
     This call was frequent
```

And the following warnings when the number continued to increase very fast and passed another treashold:

```
Warning [MEDIUM] Found several SYS_clone calls
     This call was very frequent in a short period of time
```

8.4 Macro Benchmarks

In this section we show the effectiveness of HTH on more 'real world' applications. We modified a few real applications to include malicious code and test HTH.

8.4.1 pwsafe

pwsafe [25] is a password database manager. It is used to model malicious code that sends user's private information to a remote attacker (i.e. pwunsafe).

pwsafe is a unix command line program. To test it, we have created a simple database which contain a couple of entries in a '/.pwsafe.dat' database file. In our test case we first execute `pwsafe -help` - which prints a help message, and then `pwsafe --exportdp` which prints out the database content.

When running HTH on pwsafe no warnings are produced. Harrier does detect the write to STDOUT, but inaccurately identifies one of data source of the data as STDIN.

Next, we added malicious code which sends the data (passwords) to a hardcoded server. When we run HTH on the malicious version of pwsafe, we detect that data is sent to the hardcoded socket and produce the following warnings.

```
Warning [LOW]        Found Write call
     Data Flowing From: /lib/libcrypto.so.4
     To: duero:40400 (AF_INET)
     target (client) socket-name was hardcoded in:
       ("/proj/arch4/mmoffie/PIN/MacroBenchmarks/pwsafe/pwsafe-0.2.0-mod/pwsafe")

Warning [LOW]        Found Write call
     Data Flowing From: /usr/lib/libreadline.so.4
     To: duero:40400 (AF_INET)
     target (client) socket-name was hardcoded in:
       ("/proj/arch4/mmoffie/PIN/MacroBenchmarks/pwsafe/pwsafe-0.2.0-mod/pwsafe")
```

Note that we do not correctly detect all the sources of the data: One of the sources should have been the database '/.pwsafe.dat' file. Another source, the socket itself should not be a source at all. The warnings generated are of data flowing from shared objects (/lib/libcrypto.so.4 and /usr/lib/libreadline.so.4) to the socket.

8.4.2 mw2.2.1

mw2.2.1 [19] is a perl script that is used to look up a word at the Merriam-Webster web site. The script is used to test how well HTH handles resource abuse.

Note that while mw2.2.1 is a perl script, HTH monitors the perl executable. In other words, Harrier monitors '/usr/bin/perl' (version 5.8.0) which is itself running the script. We have turned off the dataflow tracking since we were monitoring perl and not the script directly. Turning off data flow enabled Harrier to run much faster and eliminated false positives associated with executing perl instead of the script. HTH did not produce any warning on the original scripts.

We have modified the script to fork more than 20 children. HTH produced the following warnings:

```
Warning [LOW]    Found several SYS_clone calls
     This call was frequent
```

and if the script was allowed to continue:

```
Warning [MEDIUM] Found several SYS_clone calls
     This call was very frequent in a short period of time
```

HTH was able to detect the simple scenario of resource abuse even though it was running the script indirectly.

8.4.3 Tic Tac Toe

Ultra Tic Tac Toe [18] is a simple, console based Tic Tac Toe game. We use this game to model a Trojan which writes malicious code into a file and then executes it.

When running HTH on Tic Tac Toe no warnings are produced. We have then modified the game to write a hardcoded value to a file. Next, we change the file permissions to allow the user to execute it and execute the file[9] on his behalf (from within the game). The file name is hardcoded in the game.

When the modified game was executed, HTH produced the following warning:

```
Warning [HIGH]   Found Write call to ./malicious_code.txt
    The Data written to this file is originated from the
      BINARY:("/proj/arch4/mmoffie/PIN/MacroBenchmarks/uttt/uttt-
      0.11.0.micha/src/ttt")
    Moreover, it seems that the name of the file: ./malicious_code.txt
      originated from a BINARY:
      ("/proj/arch4/mmoffie/PIN/MacroBenchmarks/uttt/uttt-0.11.0.micha/src/ttt")
```

and when we allowed it to continue:

```
Warning [LOW]    Found SYS_execve call ("./malicious_code.txt")
    ("./malicious_code.txt") originated from
      ("/proj/arch4/mmoffie/PIN/MacroBenchmarks/uttt/uttt-0.11.0.micha/src/ttt")
```

9 Performance Evaluation

Harrier's main performance bottle neck is caused be tracking the data flow. Currently, the prototype implementation is very naive. The data structures used to implement data flow are not as efficient and are therefore relatively slow. Since we need to instrument almost every instruction and track the data flowing (between the registers and memory and registers), each such instruction introduces additional overhead caused by accessing the new data structures.

In our next step we will consider different alternatives to accelerate data flow. Those alternatives may include hardware support such as RIFLE's [34] architectural and micro-architectural extensions, or more efficient binary instrumentation methods and data flow tracking such as DOG [36] which was developed in MIT. Initial work with Winnie Cheng from MIT is already on the way.

[9] In our test case we just write a string into a file and execute that file, the execution fails since the file is not in a executable format.

10 Conclusions and Future work

In this work we show how we can identify common malicious behavior, patterns and characteristics of Trojan Horses and Backdoors. We developed a security policy that can correctly identify such malicious code segments, and constructed a first prototype which demonstrates the feasibility of our approach.

Results show that HTH is capable of finding Trojan Horses and Backdoors, thus achieving the most important goal. HTH does however generate false positives. This issue may be solved in several different ways including creating custom policies based on the user responses, as well as expanding Secpert policy to allow HTH to monitor a program across different sessions as well as monitor different programs simultaneously.

Additional work has to be done to show that HTH can run with an acceptable slow down.

Future work can expand and improve HTH in several different directions:

1. Adding hardware support for improving the performance of HTH such as [34].
2. Considering hybrid approaches in which static analysis may direct the runtime monitor. This may allow us to benefit from less runtime overhead (some of the analysis can be done statically) while having access to all dynamic information.
3. Expanding the ideas presented in order to identify distributed malicious attacks. An example of this type of attacks are bot networks. These new types of treats have increased during the first half of 2005 and are likely to dominate the new threat landscape [31]. Tracking complex distributed behavior patterns across the network may allow us to effectively detect such malicious bots.
4. Add new rules to support different types of resource abuse such as memory or network abuse.
5. Analyze the data downloaded or uploaded from the network or files and incorporate this information in current and new rules. For instance, if we can analyze and detect what the type of a downloaded file is (.gif, .doc or .exe) we can incorporate this to our policy. The detection itself does not need to be based on the suffix, analyzing the content itself may be more accurate.
6. Cross session - expanding the rules to take into account a program's behavior during several different executions. This will allow us to be more accurate and reduce false positives. We will need to save all the information between two consecutive executions. If we do this, we will be able to create more fine grain rules, for instance, when data is downloaded to a file we will be able to see how that file is being used in later executions instead of immediately producing an error. We can then replace the rule that will warn that we are saving data to a hardcoded file with a set of rules that track (potentially in later executions) how that file us being used.
7. Simultaneous sessions - adding support to concurrently monitor different executions on one machine, and introducing new rules and policy to detect interactions between the different programs.
8. Reduce the number of false positives. This may be done using user feedback and an adaptive policy. In addition extending HTH and Secpert to track programs across differ-

ent sessions as well as tracking multiple programs simultaneously will reduce the number of false positives.

References

[1] A. Chesla. Intrusion prevension and expert systems. *Information Systems Security Association*, pages 6–10, Apr. 2004.

[2] D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.

[3] G. Edjlali, A. Acharya, and V. Chaudhary. History-based access control for mobile code. In *CCS '98: Proceedings of the 5th ACM conference on Computer and communications security*, pages 38–48, 1998.

[4] Security Focus. Security focus exploits. http://www.securityfocus.com/.

[5] D. Gao, M. K. Reiter, and D. Song. On gray-box program tracking for anomaly detection. In *Proceedings of the 13th USENIX Security Symposium*, pages 103–118, San Diego, CA, USA, Aug. 9-13 2004.

[6] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine based platform for trusted computing. pages 193–206. Proceedings of the nineteenth ACM symposium on Operating systems principles, 2003.

[7] J. C. Giarratanoq. *CLIPS User's Guide*, Mar. 31 2002. http://www.ghg.net/clips/CLIPS.html.

[8] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications (confining the wily hacker). In *Proceedings of the 6th Usenix Security Symposium*, San Jose, CA, USA, 1996.

[9] LURHQ Threat Intelligence Group. Lurhq. http://www.lurhq.com/.

[10] hoobie. hoobie security exploits. http://www.hoobie.net/security/exploits/.

[11] insecure. Insecure exploit world. http://www.insecure.org/sploits.html.

[12] A. Kazarov and Y. Ryabov. Clips expert system tool: a candidate for the diagnostic system engine. ATLAS DAQ, NoteNumber 108, Version 1.0, Dec. 11 1998.

[13] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *Security '02: Proceeding of the 11th USENIX Security Symposium*, San Francisco, August 2002.

[14] C. Ko, T. Fraser, L. Badger, and D. Kilpatrick. Detecting and countering system intrusions using software wrappers. In *Proceedings of the USENIX Security Conference*, pages 145–156, Jan. 2000.

[15] A. P. Kosoresow and S. A. Hofmeyr. Intrusion detection via system call traces. *IEEE Softw.*, 14(5):35–42, 1997.

[16] Z. Liang, V. N. Venkatakrishnan, and R. Sekar. Isolated program execution: An application transparent approach for executing untrusted programs. pages 182–191. 19th Annual Computer Security Applications Conference, Dec 2003.

[17] C. K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation (PLDI)*, Jun. 2005. Chicago, IL.

[18] Benjamin Miller. Ultra tic tac toe. http://ttt.sourceforge.net/.

[19] mw2.2.1. A special-purpose simple script that looks up a word from merriam-webster site. http://www.cpan.org/scripts/index.html, http://www.cpan.org/authors/id/Z/ZH/ZHOUXIN/mw2.2.1.

[20] A. C. Myers. Jflow: practical mostly-static information flow control. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241, New York, NY, USA, 1999. ACM Press.

[21] N. Nethercote and J. Seward. Valgrind: A program supervision framework. In *Electronic Notes in Theoretical Computer Science*, volume 89, pages 1–23. Elsevier, 2003.

[22] Neworder. Neworder. http://neworder.box.sk/.

[23] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *The 12th Annual Network and Distributed System Security Symposium*, Feb. 3-4, San Diego, CA, USA, 2005.

[24] perldoc.perl.org. Perl 5.8.7 documentation, perlsec - perl security. http://perldoc.perl.org/perlsec.html.

[25] pwsafe. pwsafe password database. http://nsd.dyndns.org/pwsafe/.

[26] Bruce Schneier. Attack trends 2004 and 2005. In *ACM Queue vol. 3, no. 5.* ACM, Jun. 2005. http://acmqueue.com/.

[27] K. Scott and J. Davidson. Safe virtual execution using software dynamic translation. In *ACSAC '02: Proceedings of the 18th Annual Computer Security Applications Conference*, page 209, Washington, DC, USA, 2002. IEEE Computer Society.

[28] J. Seward and N. Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *USENIX 2005 Annual Technical Conference*, pages 17–30, Apr. 10-15, Anaheim, CA, USA, 2005.

[29] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 85–96, New York, NY, USA, 2004. ACM Press.

[30] Symantec. Symantec security response. http://securityresponse.symantec.com/avcenter/, 2004-2005.

[31] Symantec. Symantec internet security threat report, trends for january 05 - june 05, 2005.

[32] United States Computer Emergency Readiness Team. Us-cert. http://www.us-cert.gov/.

[33] Carnegie Mellon University. Cert. http://www.cert.org/.

[34] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. Rifle: An architectural framework for user-centric information-flow security. In *MICRO 37: Proceedings of the 37th annual International Symposium on Microarchitecture*, pages 243–254, Washington, DC, USA, 2004. IEEE Computer Society.

[35] S. Wiriyacoonkasem and A.C. Esterline. Adaptive learning expert systems. In *Southeastcon 2000*, pages 445–448, Apr. 7-9 2000.

[36] Q. Zhao, W. W. Cheng, B. Yu, and S. Hiroshige. Dog: Efficient information flow tracing and program monitoring with dynamic binary rewriting. http://pdos.csail.mit.edu/6.824/reports/beiyu.pdf, 2005.

Appendix

A CLIPS implementation details

A.1 CLIPS facts

In this section we show an example of a fact (or event) that is asserted in the expert system.

The fact asserted is a resource access type event (as described in 6.1.2). As can be seen, the execve system is being called, its resource name and type are specified ("/bin/ls/" and FILE type) as well as information on the resource ID. The resource ID data source contains the origin of the "/bin/ls/" string, which in this case is the "execve.exe" test program itself and its type: BINARY. This means that the string "/bin/ls/" was hardcoded. The execution time, frequency and address of the execve call in the program are given as well.

```
CLIPS> (assert (system_call_access
        (system_call_name SYS_execve)
        (resource_name "/bin/ls")
        (resource_type FILE)
        (resource_origin_name
                  "/proj/arch4/mmoffie/PIN/MicroBenchmarks/execve/execve.exe")
        (resource_origin_type  BINARY)
        (time 33)
        (frequency 1)
        (address "8048403")
        ) )
```

A.2 CLIPS execve rule

In this section we show an implementation of one rule in CLIPS. We show the implementation of the execution flow rule as described in section 4.1.

The following rule (defrule) defines the rule. It has two parts, the "if" condition part, which is the part before the `=>` and the "then" part, which is the part after the `=>`.

The condition part, matches an *execve* system call (new process) and checks whether the process name (the origin name and type) is hardcoded or originated from a socket. We use the `filter_binary` and `filter_socket` functions to filter out trusted binaries and sockets. In our prototype we trust the libc and ld-linux shared objects. We do not trust any sockets although our implementation does support this.

In the "then" part, we set the `warning` variable according to the policy and warn the user with all relevant information including the binaries or socket from where the process name originated from.

```
(defrule check_execve "check execve"
    ?execve <- (system_call_access
        (system_call_name ?sys_name)
        (resource_name $?name)
        (resource_type $?type)
```

```
        (resource_origin_name $?origin_name)
        (resource_origin_type $?origin_type)
        (time ?time)
        (frequency ?freq)
        (address ?addr))
    ?resolution <- (resolution (status RESOLVE))
    (system_call_name (name ?sys_name))
    (test (eq ?sys_name SYS_execve))
    (test (or
       (not (empty-list (filter_binary $?origin_type $?origin_name)))
       (not (empty-list (filter_socket $?origin_type $?origin_name))) ))
=>
    (bind ?suspicous_binaries (filter_binary $?origin_type $?origin_name))
    (bind ?suspicous_sockets  (filter_socket $?origin_type $?origin_name))

    ; for exec from binary:
    (bind ?warning 1)  ;  low
    (if ( and (< ?freq ?*RARE_FREQUENCY*)
              (> ?time ?*LONG_TIME*))
        then
          (bind ?warning 2)  ;  medium
    )
    ; for exec from socket
    ( if ( not (empty-list ?suspicous_sockets))
        then
          (bind ?warning 3)  ; High
    )

    (print-warning ?warning)
    (printout t "Found " ?sys_name " call " ?name crlf)

    ( if ( not (empty-list ?suspicous_binaries))
        then
          (printout t ?*TAB* ?name " originated from " ?suspicous_binaries crlf)
        else
          (printout t ?*TAB* ?name " originated from " ?suspicous_sockets crlf)
    )

    ; I need the following to give feedback ..
    (if ( and (< ?freq ?*RARE_FREQUENCY*)
              (> ?time ?*LONG_TIME*))
        then
            (printout t ?*TAB* "This code is rarely executed..." crlf)
    )
    (retract ?execve ?resolution)
```

```
    (assert (resolution (status STOP)))
)
```

A.3 CLIPS fires the execve rule

This sections shows CLIPS output when the execve rule (shown in appendix A.2) is fired. The execve rule is fired when three facts match the "if" part of the execve rule. f-43 is the fact shown in A.1. Facts f-42 and f-5 are implementation specific facts: f-42 is a fact used to keep track of CLIPS state, and f-5 fact is asserting that the string "SYS_execve" is a system call.

The output shown, is generated by the execve rule, and shows the warning and the reason for the warning.

```
FIRE    1 check_execve: f-43,f-42,f-5
Warning [LOW]    Found SYS_execve call ("/bin/ls")
    ("/bin/ls") originated from
          ("/proj/arch4/mmoffie/PIN/MicroBenchmarks/execve/execve.exe")
```

B Secure binary

In this section we introduce a concept we call a *Secure Binary*.

A *Secure Binary* is a binary that can be dynamically or statically verified to uphold the rules, and will be consider *safer* but not *safe* with respect to malicious code such as Trojan Horses and Backdoors.

A *Secure Binary* must conform to the following rules:

1. There is no hard-coded data in the binary - any data used by the program should be an input to the program.

This rule is very effective against a lot of the Trojan horses described above which usually contain some hardcoded resource name. Moreover, it is even effective against TCP wrappers Trojan which gives root access to intruders that connect via port 421 (where 421 is hardcoded).

But this rule will be too hard to maintain, even a simple loop end condition will be required to be given as input. Essentially this rule separates the algorithm from its instantiation.

Thus, we will need to relax the rule as follows:

1. There is no hard-coded data in the binary used towards a resource name/type or resource content - in other words, No file name or socket name my be hardcoded. In addition, when writing to such a resource the data must never be hardcoded.

This rule will not prevent the TCP wrappers Trojan from giving root access to intruders that connect via port 421, but will prevent it from sending identifying information to the intruder.

We argue that a *Secure Binary* is a *safer* binary and is **less likely** to contain a Trojan horse or Backdoor.

C Northern Harrier

The Northern Harrier shown in Figure 7 is a small hawk which systematically searches an area for pray.

Figure 7: A female harrier
http://www.mbr-pwrc.usgs.gov/id/framlst/i3310id.html