3-2009

Information Leakage Detection in Distributed Systems using Software Agents

Yung-Chuan Lee
Southern Illinois University Carbondale

Stephen Bishop
Southern Illinois University Carbondale

Hamed Okhravi
Southern Illinois University Carbondale

Shahram Rahimi
Southern Illinois University Carbondale, rahimi@cs.siu.edu

Published in Lee, Y.-C., Bishop, S., Okhravi, H., & Rahimi, S. (2009). Information leakage detection in distributed systems using software agents. IEEE Symposium on Intelligent Agents, 2009. IA '09, 128-135. doi: 10.1109/IA.2009.4927510

Recommended Citation
Lee, Yung-Chuan; Stephen Bishop; Hamed Okhravi; and Shahram Rahimi. "Information Leakage Detection in Distributed Systems using Software Agents." (Mar 2009).

Information Leakage Detection in Distributed Systems using Software Agents

Yung-Chuan Lee, Stephen Bishop, Hamed Okhravi and Shahram Rahimi

Abstract— **Covert channel attacks utilize shared resources to indirectly transmit sensitive information to unauthorized parties. Current security mechanisms such as SELinux rely on tagging the filesystem with access control properties. However, such mechanisms do not provide strong protection against information laundering via covert channels. Colored Linux [20], an extension to SELinux, utilizes watermarking algorithms to "color" the contents of each file with their respective security classification to enhance resistance to information laundering attacks. In this paper, we propose a mobile agent-based approach to automate the process of detecting and coloring receptive hosts' filesystems and monitoring the colored filesystem for instances of potential information leakage. Implementation details and execution results are included to illustrate the merits of the proposed approach.**

I. INTRODUCTION

INFORMATION security has been researched to considerable depth in the ongoing quest to provide users and corporate entities a more secure computing environment. Although an extraordinary range of effective approaches have been developed to mitigate threats to information security, new threats appear daily. Within the realm of such threats, among the most difficult to detect and prevent involve covert channel, or side channel, attacks. A covert channel is a byproduct of shared resources like memory, network interfaces, and execution time on computing devices and can be created and accessed dynamically [19], [20]. Examples of information leakage can be found in [19], [23]. Because covert channels are created from shared resources, it is very difficult to detect and prevent their occurrences. Covert channel attacks are often employed to bypass conventional security mechanisms by an authorized insider while leaking oftentimes sensitive information between processes. It is common for such attacks to involve the transfer of data from highly privileged processes to processes which would otherwise lack the necessary permissions to access such data.

According to a 2006 Global Security survey by Deloitte, insider fraud and information leakage contribute 28 percent and 18 percent of internal breaches respectively [16]. The most critical factor of insider data leakage involves users with varying permissions and privilege levels as designated by their respective positions [12]. Kowalski et al., (2008) indicated that more than half of the insider data breaches occurred within organizations and that information was accessed through organizations' computers. Because internal security breaches are caused by legitimate and authenticated users, most conventional security measurements cannot effectively detect and prevent such activities.

Modern operating systems counter unauthorized accesses through the use of access control tags, or labels, applied to subjects (e.g. processes or users) and objects (e.g. files). These labels are compared with the permissions assigned to users attempting to access the labeled files. Access is then granted or denied depending on these permissions. While this mechanism provides effective access control in most situations, it is vulnerable to covert channel attacks. Such attacks enable laundering of the access control tags applied by the operating system, allowing for arbitrary tag removal or tag reassignment.

Colored Linux [20] provides an extension to SELinux based on data watermarking, or coloring. The approach in Colored Linux is to generate blind watermark signatures for all files on a filesystem based on each file's access control tag. These watermarks are then applied to all files. When file access is requested, Colored Linux examines the requested file's watermark and compares it to the file's tag. Discrepancies between the watermark and security tag indicate that an unauthorized modification to the tag has been made and appropriate measures can then be taken. If the watermark matches the tag, SELinux access control measures take over as usual. Furthermore, if adequately robust watermarking algorithms are used, attempts to remove a watermark will render the associated file's contents useless to the attacker.

Colored Linux was implemented primarily through modification of the SELinux kernel modules. The main advantage of their approach is that it does not need any knowledge of covert channels since the modification of the operating system is on filesystem kernel to monitor read and write accesses. On the other hand, while operating system-based coloring scheme works effectively in a "closed" system (i.e., a system in which every machine is running the modified operating system), it is not as effective in an "open" system (i.e., a system which is connected to machines with non-colored operating systems). Unless the borders of an open system are tighly controlled, an insider can distord watermarked files beyond recognition (e.g. using encryption) and

---

Yung-Chuan Lee is a PhD candidate and Computer Information Specialist with the Department of Computer Science, Southern Illinois University, Carbondale, Illinois, USA (email: ylee@cs.siu.edu).

Stephen Bishop is a Masters student with the Department of Computer Science, Southern Illinois University, Carbondale, Illinois, USA (email: sbishop@cs.siu.edu).

Hamed Okhravi is a PhD candidate and Graduate Research Assistant in Cyber Security with the Information Trust Institute (ITI) and Center for Reliable and High-Performance Computing (CRHC) at the Department of Electrical and Computer Engineering (ECE), University of Illinois at Urbana-Champaign, Urbana, Illinois, USA (email: hamed@crhc.uiuc.edu).

Shahram Rahimi is an Associate Professor and Director of the Undergraduate Program with the Department of Computer Science, Southern Illinois University, Carbondale, Illinois, USA (email: rahimi@cs.siu.edu).

978-1-4244-2767-3/09/$25.00 ©2009 IEEE

leak them to the outside using cross-border covert channels.

To overcome this drawback, we propose an information leakage detection (ILD) agent system to automate the processes of converting a regular machine to colored one. Benefits of such an approach involve the ability to modify and add detection capabilities in a modular fashion while simultaneously providing conditional deployment of such capabilities. With mobile agents, such dynamism can be realized with little or no administrative involvement. Furthermore, the distributed reporting potential of mobile agent networks can lend itself well to future analysis of information leakage, as well as the underlying covert channel techniques. The agent based approach also makes the coloring scheme effective in an open system which is a hybrid of machines running modified operating systems and comodity ones. Given comparable requirements for a small memory footprint and ease of integration with relatively low-level system constructs necessary to accomplish efficient filesystem monitoring, we have chosen Mobile-C [6] for our mobile agent platform, as it meets all of our requirements.

In Section 2, related works on information leakage prevention or detection are presented. Section 3 provides an overview of Colored Linux while our proposed Information Leakage Detection agent community is discussed in Section 4. Section 5 details several detection methodologies and their respective limitations. Section 6 provides the proposed strategies which will be implemented in our system. Section 7 lists the inter-agent communications present in our system. In Section 8, implementation details and results of the host-resident agents in our system are examined. Finally, Section 9 provides conclusions and future directions.

II. RELATED WORK

Since most of the studies in security community deal with preventing outsiders access, there are only a few literatures that have proposed methodologies to address the issue of information leakage though insiders. Alawneh and Abbadi introduced a mechanism to protect shared information among organizations via Trusted Platform Module (TPM) [1], [2]. By creating master controller and domains for TPM equipped devices, contents can only be accessed through the allowed devices. Takesue proposed a scheme to prevent information leakage through portable devices [22]. A modified i-node with 1-bit flag bit and 1-bit lock bit imposes authentications with integrated network location checking between storage devices and security server, a user can only access the files when she is inside the company and the authentication is succeeded. Change and Kim designed a system to prevent information leakage in ubiquitous computing environment [5]. The approach utilizes cryptographic algorithms and authentication methods in agents to secure the sensitive data during communications. Although these approaches present potential solutions to insider information threats, none of them examine the risk of covert channel attacks.

An overview of covert channel attacks are discussed in the following efforts [3], [18], [27]. A network-based storage covert channel based on IP time to live (TTL) field is designed in [21]. A link-layer network-based covert channel in the MAC protocol based on the splitting algorithm is proposed in [13]. Cabuk, et. al. have designed and studied network-based timing channels and mechanisms to disrupt such channels in [4]. A work by Wang and Lee [26] studies hardware-based (processor-based) timing channels and identifies two such channels in typical Simultaneous Multi-Threaded (SMT) processors.

There are also countermeasures proposed for known covert channel attacks such as information flow analysis techniques [24], time-domain anomaly [25], entropy-based approach [9], data-dependency scheme [17], and store-forward approach [10], [11]. However, most of these countermeasures work best for known covert channels. The problem, however, is that it is impossible to enumerate all covert channels in a real system. Hence, it is that unknown channels that pose the greatest threat to the security of the system.

Robust watermarking (coloring) offers a strong binding between data and its security tag and can detect information leakage from both insiders as well as through covert channel attacks. Further, our proposed approach to combine mobile agents with Colored Linux methodologies is novel.

III. COLORED LINUX

Information luandering through covert channel attacks is possible because the binding between the data and its security tag is loose; i.e., the security tag is usually appended to the end of the file as a bit stream. If such a file leaks through a covert channel, the tag becomes meaningless and easily removable. The insight behind Colored Linux is to make this binding strong by coloring (watermarking) data files. If the watermarking algorithm is robust, it is impossible for an attacker to remove the watermark without destroying the data itself.

Colored linux has a coloring algorithm, “brush”, for each file type (e.g. one for images, another one for text files, etc.). The set of all algorithms in the operating system is called the “brush set”.

The entire filesystem is colored during the initialization. Upon each access to a file, the color of the file is compared to its security tag. If there is a discrepancy between the two, it means that the file has been leaked through a covert channel and its security tag has been laundered. Note that Colored Linux knows nothing about the mechanism of covert communication. However, it can detect any leakage and prevent any furthure damage by taking appropriate measures.

The watermarking algorithms used in Colored Linux must be blind; i.e., they should be able to detect the watermark without needing the original file. This ensures that the security of the system is not endangered by storing the original uncolored file in the filesystem.

Colored Linux is implemented by modifying SELinux hooks. SELinux hooks are invoked whenever a resource is accessed in order to check the policy. By modifying these hooks, whenever a file is accessed, the color detecting algorithm is called to check the color of the file and compare it with its security tag. If it matches, the control is passed over to the SELinux engine. Otherwise, the process is terminated and appropriate logs are created. Colored Linux modules are called whenever an object is created, accessed, changed, or its tag is modified.

The assumption in Colored Linux is that there is a boundary beyond which covert communication is very difficult or impractical (a closed system). Every system inside that boundary is running color-aware operating system. Moreoever, color awareness is manually installed on all of the machines inside this boundary. ILD agent based system address these drawbacks by moving through hosts that are not color-aware and automating the coloring mechanism.

IV. ILD AGENT SYSTEM

Separation of powers and responsibilities in an agent community encourages flexibility and encapsulation. As such, our proposed agent system will be heterogeneous with members belonging to one of six principle archetypes, each adhering to unique roles and possessing distinct abilities. Figure 1 depicts the classifications of our Information Leakage Detection (ILD) Agent system and the respective agent ranks. All inter-agent communications will adhere to FIPA Agent Communication Language (ACL) specifications in order to maintain communication interoperability between different agent platforms. Properties and responsibilities of each type of agent are discussed in following subsections.

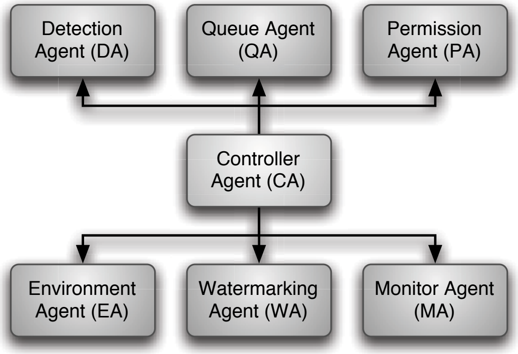

Fig. 1. Agent Classifications and Hierarchy

A. *Controller Agents (CA)*

Controller Agents are responsible for dispatching subordinate agents and coordinating their respective activities in a designated network. Additionally, Controller Agents will coordinate the remote installation of the necessary mobile agent environment and other required software packages on target hosts with Environment Agents. Multiple instances of controller agents can be dispatched to ensure proper coverage of large networks as well as to accomplish load distribution for the purposes of performance optimization.

B. *Detection Agents (DA)*

The main functionality of Detection Agents is to identify new hosts in the network and to verify the host's states. In our initial design, a host's state will refer to the presence or absence of SELinux and the Colored Linux infrastructure. Once determined, a host's state will be reported to the Controller Agent to aid in the identification of subsequent actions.

C. *Queue Agents (QA)*

To avoid overwhelming Controller Agents and to provide an orderly approach to dispatching agents to newly discovered hosts, Queue Agents will be useful. As stated above, when a Detection Agent identifies a new remote host, the host's state is reported to a Controller Agent. Rather than dispatching agents to a new host immediately, it may be preferred to defer such processing for some time, especially in the case when many such hosts are reported at once. In such cases, hosts are reported by Controller Agents to Queue Agents which prioritize hosts for subsequent processing by, and at the request of, Controller Agents.

D. *Monitor Agents (MA)*

Monitor Agents will perform active monitoring on the host filesystem through the *inotify* kernel subsystem to identify file write and creation operations. Details on the *inotify* kernel subsystem will be discussed in the next section. When a write operation or file creation operation takes place, Monitor Agents notify Watermarking Agents which can then perform watermark analysis of the file in question. As comparable capabilities are already present in Colored Linux hosts, Monitor Agents will only reside in non-Colored SELinux hosts.

E. *Watermarking Agents (WA)*

Similar to Monitor Agents, Watermarking Agents shall only be present on non-Colored hosts, as determined by Detection Agents. The responsibility of these agents is to watermark all files on a host's filesystem and to perform subsequent watermark analysis at the request of Monitor Agents.

F. *Permission Agents (PA)*

A central Permission Agent handles permissions issues involving Monitor Agents and Watermarking Agents with their target hosts. Specifically, the Permission Agent should ensure that such agents are given only those permissions necessary to perform their respective tasks. In addition, the Permission Agent ensures that all permissions necessary for agent environment installation by the Environment Agent are in place.

G. *Environment Agents (EA)*

Minimally, Watermarking and Monitor Agents require the necessary agent environment installed on a target host in order to reside and function there. Also, depending on the type of watermarking employed, certain watermarking-specific software dependencies which may not reasonably be accommodated by the Watermarking Agents themselves can exist. Environment Agents will be responsible for handling all such software dependencies without the intervention of the target host's administrator.

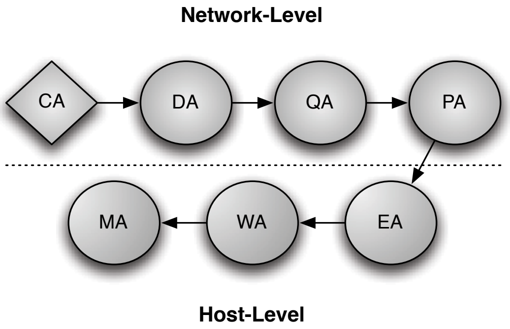

Fig. 2. Process flow of our proposed system.

V. DETECTION METHODOLOGIES

Detecting file "write" or "create" operations in a non-Colored host is the first step towards detecting potential information leakage. In this section, we examine four candidate methods and the feasibility and cost associated with performing such detection. One method is then selected and implemented in our proposed approach.

A. Memory Scanning

Memory scanning involves systematic scans of the target machine's memory space (accessible via the /dev/mem virtual device). In cooperation with information obtained from /proc, it would be possible to locate any given process' memory space in /dev/mem and scan that space for "write" calls. However, the time cost of a single scan of a system memory snapshot depends on the number of processes in the system. As the number of the processes increases, so does the scanning time. Thus, there is no guarantee that the scanning time will always be less than a given processes' execution time, making this method prone to missed detections.

B. Process Tracing

Process tracing works similarly to the previous method, with the exception that instead of scanning system memory, it will use the Process Trace, "ptrace," system call (as is used by the "strace" command) to attach to a process and monitor all system calls, including "write" (and "open" calls with the create flag set, as is needed by the Monitor Agent). Sharing the same concept as memory scanning, this method is susceptible to the same timing issue. Primarily, we must be aware of process creation and must attach to it with ptrace before the process issues any system calls (or terminates itself).

It is unknown how exactly this can be accomplished. A process polling method may be able to catch all process creations, however, this will dramatically decrease system performance. Therefore, the performance overhead makes this method less appealing.

C. Kernel-Level System Call Hooking

In order to maintain high system performance and mitigate the time cost associated with "write" and "create" operation detection, we explored the possibility and feasibility of detecting such operations via their respective system calls at the kernel level. While this method would allow for interception of *every* such system call easily and efficiently, several potential obstacles might prevent us from choosing this method. Machine architectures and kernel versions will surely differ throughout the network, and thus one pre-compiled kernel module carried as agent payload will not be injectable into every target host. It is certainly not reasonable to maintain pre-compiled modules for every possible architecture combined with every kernel version.

One potential solution would be to carry only the module source code as payload, build it on the target machine, and load it into the kernel. While this may overcome differences in system architectures to some extent, modules for newer kernel versions are written quite differently from those intended for use in much older kernels.

D. inotify Kernel Subsystem

The *inotify* kernel subsystem is a standard filesystem event notification service included in Linux kernels since release 2.6.13 three years ago [14]. This service enables a user to create applications from system libraries to monitor file operation events like read, write or delete on a set of specified files. By default, *inotify* imposes service limitations of 16384 maximum events per queue, 128 maximum instances per user and 8192 maximum watches per instance to conserve kernel memory. We are confident that these limitations will not present overwhelming obstacles to the initial implementation of our approach; however, a more comprehensive study on trade-offs between kernel memory and *inotify* limitations will be conducted in the future.

Hence, our file operation detection in Monitor Agent will utilize the *inotify* kernel service because it provides not only stability and performance but is also accessible through uncomplicated system libraries. Although *inotify* is not available in kernel versions prior to 2.6.13, our initial targeted platform will employ a fairly recent kernel version. We will later investigate the feasibility of dynamically building and installing *inotify* modules in older hosts.

VI. PROPOSED STRATEGIES

The following subsections illustrate the states and process flow of our system. Process flow is depicted in Figure 2. Each subsection explains the objectives of each step and how they can be achieved.

A. Host Discovery

In our proposed agent system, all operations begin with, and are coordinated by, the Controller Agent. Initially, it is assumed that all hosts in the network are clean, yet unknown. A Detection Agent is dispatched to scan the network for SELinux-based hosts. When the first such host is discovered, the Detection Agent determines whether or not the newly found host is "Colored." If the host is un-Colored, it is reported to the Controller Agent.

B. *Non-Colored Host Queueing*

When the first non-Colored, SELinux-based host is identified and reported by the Detection Agent, the Controller Agent shall create a Queue Agent and make it aware of the reported host. All subsequent host reports generated by the Detection Agent will also be forwarded to the Queue Agent. Hosts are enqueued, possibly with priorities, by the Queue Agent. At certain times, the Controller Agent will query the Queue Agent for a new host, which the Queue Agent will dequeue and forward to the Controller Agent.

C. *Permission Determination and Management*

Given a host report from the Queue Agent, the Controller Agent will create a Permission Agent and assign it to the new host. The permission agent (using standard Linux remote management facilities, as a mobile agent environment has not yet been installed on the target host) will attempt to determine if the proper permissions are in place for the successful remote installation of an agent environment on the target host, and for the proper operation of subsequently dispatched Watermarking and Monitor agents. If proper permissions have not been assigned, the Permission Agent is responsible for coordinating with the target host to establish the lacking permissions. Once this process has completed, the Controller Agent remotely installs (with the aid of a helper Environment Agent) the appropriate agent environment on the target host.

D. *Watermarking Target Hosts*

Following the successful installation of the agent environment on the target host, the Controller Agent dispatches a Watermarking Agent to the host. Within the host, the Watermarking Agent "colors" all files on the host's filesystem. Upon completion of initial coloring, the Watermarking Agent reports completion to the Controller Agent, and then awaits subsequent commands. Detection of a newly created file, or of write operations performed on an existing file, are reported to the Watermarking Agent by the Monitor Agent, prompting the Watermarking Agent to analyze and possibly color the new file. This process continues until the Controller Agent instructs the Watermarking Agent to terminate.

E. *File Creation and Write Monitoring*

Once the Watermaking Agent has reported successful completion of initial coloring to the Controller Agent, a Monitoring Agent is sent to the newly colored host. This agent will then use the methods described above in *Section 4c, and Section 5c* to detect and handle potential instances of information leakage.

VII. COMMUNICATIONS AMONG AGENTS

In our proposed agent architecture, communications among agents will follow the FIPA communicative act specification which is based on the Speech Act Theory to facilitate communication interoperability between different agent platforms [8]. The specification defines 22 composite and macro communicative acts to provide conversational actions such as *INFORM, REQUEST or PROPOSE*. Table I through VI illustrates the communication details of the processes mentioned in previous section. Figure **??** shows a sequence diagram of simultaneous message exchanges between agents in the proposed system.

TABLE I

CONTROLLER AGENT COMMUNICATIONS

From: **Controller Agent (CA)**

To: Detection Agent (DA)

- Ask the DA to notify CA when the first non-colored host is found. (*REQUEST-WHEN*)
- After first host found, ask the DA to notify QA whenever non-colored hosts are found. (*REQUEST-WHENEVER*)

To: Queue Agent (QA)

- Ask the QA to insert current non-colored hosts to its queue. (*REQUEST*)
- Retrieve the hosts in the QA's queue. (*REQUEST with INFORM*)

To: Permission Agent (PA)

- Request PA to prepare target host for agent environment installation. (*REQUEST*)

To: Watermarking Agent (WA)

- Ask the WA to watermark the host's filesystem and report the completion. (*REQUEST-WHEN*)

To: Monitor Agent (MA)

- Ask the MA to monitor the target host and notify the CA when information leakage occurred. (*SUBSCRIBE*)

To: Environment Agent (EA)

- Ask the EA to check for, and resolve, software dependencies on the target host which may inhibit the functionality of subsequently dispatched agents. (*REQUEST*)

TABLE II

DETECTION AGENT COMMUNICATIONS

From: **Detection Agent (DA)**

To: Controller Agent (CA)

- Confirm to CA that network scan to determine non-colored host is proceeding. (*AGREE*)
- Notify CA when the first non-colored host is found. (*INFORM*)
- Confirm to CA that notification to QA about non-colored hosts can proceed. (*AGREE*)

To: Queue Agent (QA)

- Ask the QA to insert current non-colored hosts in its queue. (*REQUEST*)

VIII. IMPLEMENTATION AND RESULTS

A. *Agent Environment*

In choosing an appropriate foundation for our agent community, we considered primarily the associated memory

TABLE III

QUEUE AGENT COMMUNICATIONS

From: **Queue Agent (QA)**
To: Controller Agent (CA)
• Confirm to CA that queue insertion has been performed. (*AGREE*) • Return the current hosts in queue to CA. (*INFORM*)
To: Detection Agent (DA)
• Confirm to DA that queue insertion has occurred. (*AGREE*)

TABLE IV

PERMISSION AGENT COMMUNICATIONS

From: **Permission Agent (PA)**
To: Controller Agent (CA)
• Confirm to CA to prepare the host for agent environment installation. (*AGREE*) • Notify CA of the result of host preparation. (*INFORM*)

TABLE V

WATERMARKING AGENT COMMUNICATIONS

From: **Watermarking Agent (WA)**
To: Controller Agent (CA)
• Confirm with CA to perform watermarking operation. (*AGREE*) • Return the result of watermarking operation to CA. (*INFORM*)

TABLE VI

MONITOR AGENT COMMUNICATIONS

From: **Monitor Agent (MA)**
To: Controller Agent (CA)
• Confirm with CA to perform queue insertion. (*AGREE*) • Notify CA of the occurrence of information leakage. (*INFORM*)

TABLE VII

ENVIRONMENT AGENT COMMUNICATIONS

From: **Environment Agent (EA)**
To: Controller Agent (CA)
• Confirm with CA to perform environment checking and dependency resolution. (*AGREE*) • Notify CA of all resolved dependencies. (*INFORM*)

footprint as well as ease of access to system-level constructs. Mobile-C was hence accepted as our mobile agent framework due to its low memory footprint when compared to other popular agent architectures. In addition, being fully C-compliant enables Mobile-C agents to take direct advantage of the system calls provided by the Linux operating system. This is especially useful for our purposes as our Monitor Agent relies on the *inotify* system.

As a proof-of-concept, Mobile-C agents were developed to perform initial watermarking (coloring) of a portion of a filesystem in a Debian-based Linux operating system with security enhancement, i.e. SELinux, and to detect leakage of watermarked files within the colored filesystem. These agents implement the functionality of the Watermarking and Monitor agents previously described, i.e. the Host-level agents in our agent community, and identify the feasibility of our proposed system in whole.

B. Watermarking Algorithms

As different file types require different watermarking schemes, we focused on image files for our experiments. The watermarking algorithm utilized is the Dugad [7] algorithm as implemented in Peter Meerwald's watermarking library [15]. This algorithm has many nice properties, especially that of blindness, which is required for our system. Meerwald's library, in turn, depends on the NetPBM library for reading, writing, and converting images of a variety of formats.

C. Handling Dependencies

External dependencies, such as NetPBM, can be handled in several ways in mobile agent systems. Ideally, all necessary code can efficiently be carried with the agent itself. When this is not viable, the agent execution environment can be made to handle such dependencies. Mobile-C uses Ch, an embeddable, C99-compliant, C-language interpreter as its execution environment. Ch allows for the addition of user-defined packages, each of which may include header files, dynamically-linked libraries, scripts, and other resources required by users of the interpreter. In our case, these users are our Mobile-C agents.

While we aim to implement all watermarking functionality within agents, certain dependencies, such as NetPBM, cannot reasonably be accommodated by agents themselves and will therefore be added as separate packages to the Ch execution environment. For these purposes, an Environment Agent capable of retrieving, building, and installing into the execution environment packages which are needed by Watermarking Agents shall be employed. This will be helpful as new watermarking techniques and information leakage detection methods are developed which may require large and complex software suites to function.

D. Implementation of the Watermarking Agent

As described above, the primary role of a Watermarking Agent is to prepare a filesystem for information leakage detection by watermarking all files with a particular permissions tag. Such tags essentially identify the sensitivity of a file and are used in conjunction with permissions assigned to individual users. A user's permissions regulate which files are accessible by the user. Here, accessibility can relate to the ability of a user to read, write, or execute a file, or perform any combination of these actions. Information leakage via covert channels may result in the removal or modification of traditional permissions tags. The recipient of the leaked information may alter the tags in order to grant himself access to the information that he was not intended to possess. Watermarking embeds the permissions of a file within the file contents in such a way as to be (ideally) irremovable without rendering the file contents useless.

Functionally, the Watermarking Agent developed for our experiments initiates a complete scan of the target filesystem upon entry into a target host. It does not, however, indiscriminately watermark all files encountered. It could be the case that the filesystem, or portions of it, is already watermarked but the agent, agency, or supporting infrastructure was damaged or removed due to some unforeseen circumstance. Therefore, the Watermarking Agent will attempt to detect the presence of a watermark in all scanned files prior to watermarking. If a watermark is not detected, the file is watermarked immediately with a signature corresponding to the files permissions tag. Conversely, if a watermark is detected, the Watermarking Agent will compare the watermark with the file's permissions tag. If an inconsistency is found, the file is assumed to have been previously leaked, and is either quarantined in a secure directory or securely deleted.

Once the initial watermarking phase is complete, the Watermarking Agent will become dormant. A Watermarking Agent will be awakened upon receipt of signal from the Monitor Agent indicating that a new file has been created and will therefore need to be watermarked[1]. Algorithm 1 provides a broad representation of the operations performed by our Watermarking Agent.

E. Implementation of the Monitor Agent

While the Watermarking Agent effectively binds a files permissions tag to its content, it does not compare the watermark to the permissions of a user attempting to access the file. This task is the responsibility of the Monitor Agent. The Monitor Agent serves the primary role of monitoring the target filesystem for any file "creation" or "write" operations and notifying the Watermarking Agent of such events for subsequent processing. As stated above, the file operation monitoring is achieved via the *inotify* kernel subsystem. Algorithm 2 represents the Monitor Agent operations.

F. Results

Regardless of the type of covert channel through which information is leaked, the detection methods of [20] effectively prevent any disassociation of the leaked information content from its designated permissions from being used by the recipient of the leaked information. If permissions are altered during leakage, they will no longer match the information's embedded watermark. Likewise, if the information itself is altered, then the watermark will no longer be valid. Therefore, to test our proof-of-concept implementation, permissions alteration and content alteration of monitored files were performed. Initial tests have been conducted on an Intel-based machine with Linux kernel version 2.6.24 and SELinux security extensions enabled.

Algorithm 1 Watermark(Directory D)

```
while $D$ has children do
    $d_i \leftarrow$ child i of D
    if $d_i$ is a directory then
        Watermark($d_i$)
    else
        boolean $w$ = DetectWatermark($d_i$)
        if $w$ = TRUE then
            Compare watermark of $d_i$ with permissions tag
            if Watermark does not match tag then
                Quarantine or Securely Remove $d_i$
            end if
        else
            Watermark $d_i$ with signature = permissions tag
        end if
    end if
end while
return
```

Algorithm 2 Monitor()

```
W $\leftarrow$ inotify event descriptor
for all Target directories $d_i$ do
    Add inotify watch descriptor for "write" and "create"
    operations within $d_i$
end for
loop
    f $\leftarrow$ Read event from event descriptor W
    Pass f to Watermarking Agent for Analysis
end loop
```

First, a Watermarking Agent was introduced into our test environment and performed initial watermarking of a portion of the filesystem. As this was a proof-of-concept, these initial tests were conducted only on image files using image watermarking algorithms found in [15]. Correctness of the applied watermarks were then manually confirmed using [15]. Next, a Monitor Agent was introduced and was tested for functionality by creating new files, and writing to existing files, in the monitored portion of the filesystem. In all cases, these operations were correctly detected and communicated to the Watermarking Agent for subsequent analysis.

With the correctness of the applied watermarks and the detection capabilities of the Monitor Agent confirmed, correct detection of instances of information leakage was tested. Here, cases of security tag alteration and information content alteration where specifically tested. In the first case, a scenario involving leakage of information with a high-

---

[1] For future work, the Watermarking Agent shall be made able to detect valid changes of permissions tags, and re-watermark files accordingly.

level security classification to a user with lower-level security permissions (i.e. write-down) was simulated by changing the SELinux security tag of files without altering the files' content (and therefore not updating the files' watermarks). In these cases, the Watermarking Agent correctly identified a mismatch between the security tag and the embedded watermarks and quarantined the offending files.

To test the second category of leakage, in which information does not change with respect to security classifications but is instead leaked in such a way as to alter the *contents* of the information itself, monitored files were edited in some small ways (changing a few pixels in watermarked image files and thus destroying the watermark) without altering the SELinux security tags. In these cases, the Watermarking Agent correctly identified the invalid watermarks and quarantined the modified files.

Having correctly identified all tested instances of simulated information leakage, it has been shown that the Agent-based Information Leakage Detection system described here is viable and warrants further research and continuing development.

IX. CONCLUSION

The primary benefit of an Agent-based Information Leakage Detection system lies in the ability to modify and add detection capabilities, modularize those capabilities, and then conditionally employ such capabilities at the discretion of a central control mechanism (in our system, the Controller Agent). The use of mobile agents as described in this paper, and in general, reduces the per-host administrative complexity as once the initial agent environment is properly installed and configured, all further necessary actions are performed by the agents themselves. Additionally, mobile agents are able to provide unique reporting capabilities that, for the purposes of our research, may benefit the analysis of information leakage and the underlying covert channels through which information has been leaked.

While the information leakage detection approach detailed here is based on the work of [20], future work in this area may lead to the inclusion of techniques aimed at detecting and blocking covert channels prior to the occurrence of information leakage. Given the highly varied nature of covert channelling methods, detecting *all* such methods is likely a matter for which a solution can only be obtained through the liberal use of techniques rooted deeply in the field of artificial intelligence.

REFERENCES

[1] M. Alawneh and I.M. Abbadi, "Preventing information leakage between collaborating organisations", *In Proceedings of the 10th international Conference on Electronic Commerce*, vol. 342, pp. 1-10, 2008.

[2] M. Alawneh and I.M. Abbadi, "Preventing Insider Information Leakage for Enterprises", *The Second International Conference on Emerging Security Information, Systems and Technologies*, pp. 99-106, 2008.

[3] S. Cabuk, "Network Covert Channels: Design, Analysis, Detection, and Elimination", PhD Thesis, Purdue University, 2006.

[4] S. Cabuk , C. Brodley, and C. Shields, "IP covert timing channels: Design and detection", *In proceedings of the 2004 ACM Conference on Computer and Communications Security*, pp. 178-187, 2004.

[5] H. Chang and K. Kim, *Design of Inside Information Leakage Prevention System in Ubiquitous Computing Environment*, Lecture Notes in Computer Science, Springer Berlin, vol. 3483, pp. 128-137, 2005.

[6] B. Chen, H.H. Cheng, and J. Palen, *Mobile-C: A Mobile Agent Platform for Mobile C-C++ Agents*. Software - Practice and Experience, John Wiley and Sons, vol. 36, no. 15, pp. 1711-1733, 2006.

[7] R. Dugad, K. Ratakonda, and N. Ahuja, "A New Wavelet-based Scheme for Watermarking Images". *In Proceedings of the International Conference on Image Processing*, vol. 2, pp. 419-423, Oct. 1998.

[8] FIPA communicative Act Library Specification. "Foundation for Intelligent Physical Agents", 2000. http://www.fipa.org/specs/fipa00037/

[9] S. Gianvecchio and H. Wang, "Detecting covert timing channels: an entropy-based approach", *In Proceedings of the 14th ACM conference on Computer and communications security*, pp. 307-316, 2007.

[10] M.H. Kang, I.S. Moskowitz, and D.C. Lee , "A network Pump", *IEEE Transactions on Software Engineering*, pp. 329-338, 1996.

[11] M.H. Kang, I.S. Moskowitz, and D.C. Lee , "The Pump: A Decade of Covert Fun", *In Proceedings of the 21st Annual Computer Security Applications Conference*, pp. 352-360, 2005.

[12] E. Kowalski, D. Cappelli, and A. Moore, "Insider Threat Study: Illicit Cyber Activity in the Information Technology and Telecommunications Sector", Tech. Report, National Threat Assessment Center and Carnegie Mellon Univ., CyLab, January 2008.

[13] S. Li and A. Ephremides, "A covert channel in MAC protocols based on splitting algorithms", *IEEE Wireless Communications and Networking Conference*, vol. 2, pp. 1168-1173, 2005.

[14] R. Love, "Kernel korner: intro to inotify". *Linux Journal*, vol. 8, November 2005.

[15] P. Meerwald, *http://www.cosy.sbg.ac.at/˜pmeerw/Watermarking/*.

[16] A. Melek and M. MacKinnon, "2006 Global Security Survey. Research Report", Deloitte, 2006. http://www.deloitte.com

[17] P.M. Melliar-Smith and L.E. Moser , "Protection against covert storage and timing channels", *In Proceedings of the Computer Security Foundations Workshop IV*, pp. 209-214, 1991.

[18] J. Millen, "20 years of covert channel modeling and analysis", *In Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pp. 113-114, 1999.

[19] National Computer Security Center. "A Guide to Understanding Covert Channel Analysis of Trusted Systems", NCSC-TG-30, November 1993, http://www.radium.ncsc.mil/tpep/library/rainbow.

[20] H. Okhravi and S. Bak, "Colored Linux: Covert Channel Resistant OS Information Flow Security", University of Illinois, Urbana-Champaign. 2008.

[21] H. Qu, P. Su,and D. Feng, "A typical noisy covert channel in the IP protocol", *In Proceedings of the 38th Annual International Carnahan Conference on Security Technology*, pp. 189-192, 2004.

[22] M. Takesue, "A Scheme for Protecting the Information Leakage Via Portable Devices", *The International Conference on Emerging Security Information, Systems, and Technologies*, pp. 54-59, 2007.

[23] H. Tanaka, "Information Leakage via Electromagnetic Emanation and Effectiveness of Averaging Technique", *Information Security and Assurance*, pp. 98-101, 2008.

[24] C. Tsai, V. Gligor and C. Chandersekaran, "On the Identification of Covert Storage Channels in Secure Systems", *IEEE Transactions on Software Engineering*, vol. 16, no. 6, pp. 569-580, 1990.

[25] C. Wang and S. Ju, "Searching covert channels by identifying malicious subjects in the time domain", *In Proceedings of the Fifth Annual IEEE SMC Information Assurance Workshop*, pp. 68-73, 2004.

[26] Z. Wang and R.B. Lee, "Covert and Side Channels Due to Processor Architecture", *In Proceedings of the 22nd Annual Computer Security Applications Conference on Annual Computer Security Applications Conference*, pp. 473-482, 2006.

[27] S. Zander, G. Armitage, P. Branch, "Covert Channels and Countermeasures in Computer Network Protocols", *IEEE Communications Magazine*, vol. 45, issue 12, pp. 136-142, December 2007.