ProPatrol: Attack Investigation via Extracted High-Level Tasks

Sadegh M. Milajerdi[1], Birhanu Eshete[2,⋆], Rigel Gjomemo[1], and V.N. Venkatakrishnan[1]

[1] University of Illinois at Chicago, Chicago IL 60607, USA
{smomen2,rgjome1,venkat}@uic.edu
[2] University of Michigan-Dearborn, Dearborn MI 48128, USA
birhanu@umich.edu

Abstract. Kernel audit logs are an invaluable source of information in the forensic investigation of a cyber-attack. However, the coarse granularity of dependency information in audit logs leads to the construction of huge attack graphs which contain false or inaccurate dependencies. To overcome this problem, we propose a system, called ProPatrol, which leverages the open compartmentalized design in families of enterprise applications used in security-sensitive contexts (e.g., browser, chat client, email client). To achieve its goal, ProPatrol infers a model for an application's high-level tasks as input-processing compartments using purely the audit log events generated by that application. The main benefit of this approach is that it does not rely on source code or binary instrumentation, but only on a preliminary and general knowledge of an application's architecture to bootstrap the analysis. Our experiments with enterprise-level attacks demonstrate that ProPatrol significantly cuts down the forensic investigation effort and quickly pinpoints the root-cause of attacks. ProPatrol incurs less than 2% runtime overhead on a commodity operating system.

1 Introduction

Targeted and stealthy cyberattacks (referred to as Advanced Persistent Threats (APTs)) follow a multi-stage threat workflow [5] to break into an enterprise network with the goal of harvesting invaluable information. APTs often utilize spear phishing and drive-by download to gain a foothold in an enterprise (initial compromise). After this step, APTs propagate to enterprise targets (e.g., Intranet servers) in pursuit of high-value assets such as confidential information.

Once APTs are detected, it is crucial to track the causal linkage between events in a timely manner to find out the attack provenance. Consequently, attack provenance may be used to detect affected entities within a host or across multiple hosts. As soon as an attack provenance is uncovered, a system analyst

⋆ The second author performed this work as a postdoctoral associate at the University of Illinois at Chicago.

can take immediate damage control measures, use it to make sense of past attacks or to prevent future attacks.

The state-of-the-art technique for provenance tracking is to use *kernel audit logs* to record information flow between system entities [7, 15] and then correlate these entities for forensic analysis. In particular, after an attack is detected, system analysts use the detection point as a seed to initiate *backward tracking* strategies to determine the root-cause of that attack, and *forward tracking* methods to find out the impacts of the attack.

Kernel auditing techniques interpose at the system call layer; therefore, they have acceptable runtime overheads but suffer from the *dependency explosion* problem. In particular, due to coarse nature of dependencies that manifest in audit logs, an entity may *falsely* appear to be causally dependent on many other entities. For instance, consider a browser process that has multiple tabs open, each receiving data from different socket connections. If the browser process writes to a file, then during forensic analysis, that file will look causally dependent on all the socket connections the browser has accessed up to the write operation. In case of a drive-by-download attack that exploits that browser, it becomes challenging for system analysts to pinpoint the origin of the attack among all the accessed sockets.

To mitigate the dependency explosion problem, researchers have proposed compartmentalization techniques to partition the execution of a long-running process to smaller units [18, 20, 21]. BEEP [18] and ProTracer [21] compartmentalize processes to low-level units based on iterations of event handling loops. MPI [20] compartmentalizes processes to high-level tasks based on source code annotations manually performed by developers. Unfortunately, these techniques rely on source code or binary instrumentation.

Our Work. In this paper, we present an approach (called ProPatrol), aimed at high-level activity compartmentalization to address the dependence explosion problem and to provide *units of execution boundaries* to aid forensic attack investigation. One of the main benefits of our approach is that does not require application source/binary instrumentation. The key insight in our approach is to leverage the execution compartments that are inherent to the design of certain Internet-facing applications (e.g., browsers, chat clients, email clients) in order to mitigate the dependence explosion problem during forensic analysis and are able to pinpoint true dependencies. Through a *combination of execution compartments and provenance*, we demonstrate how a cyber-analyst can perform precise forensic attack investigation. Starting with the choice of compartmentalized applications, our approach also includes an inference mechanism to identify the execution compartments implemented in these applications directly from their audit log traces.

Our approach does require enterprise users to be restricted to the use of compartmentalized Internet-facing applications. While this may seem stringent, recent trends [13] towards locked-down enterprise software (e.g., Windows 10 S) suggest that enterprises and software vendors desire this direction. In addition, modern applications are moving to a sandboxing-based architecture both for

security and performance purposes. Google Chrome, for instance, is a relevant example of such an application, while Firefox is moving in the same direction [1].

Results Overview. Using APT scenarios as case studies, we evaluated the effectiveness and efficiency of ProPatrol on five attack scenarios in an enterprise setting. ProPatrol successfully constructed forensic graphs on five distinct lateral movement attempts that target high-value assets in Intranet servers. We note that these attacks cover a broad surface of the APT landscape. More precisely, our evaluation covers the major APT attack vectors such as spear phishing, drive-by downloads, and classic web-based attack vectors such as CSRF and DNS rebinding. In all the attacks, lateral movement is attempted by initiating a connection to an Intranet server. In addition to covering a wide space of APT vectors, our evaluation also spans web browsers, email clients, and instant messaging clients —which are the common classes of applications targeted by APTs. Measured on the five attack scenarios for its runtime, on average, ProPatrol operates with an overhead of less than 2%. Most importantly, ProPatrol is able to detect the execution compartments responsible for the attacks correctly in all the cases, thus efficiently addressing the *dependency explosion* problem.

Outline. The remainder of this paper is organized as follows. In Section 2 we motivate the problem by showing the importance of execution partitioning for better forensic analysis and describe details of Provenance Monitoring techniques that are required for log collection. Section 3 discusses the details of our compartmentalization approach. In Section 4, we highlight implementation details. Evaluation of our approach appears in Section 5. Section 6 discusses related work. Finally, Section 7 concludes the paper.

2 Background

2.1 Motivating Example

An enterprise network is typically composed of several employee machines and Intranet servers that host high confidentiality and high integrity assets. The network is often protected by a defensive perimeter consisting of firewalls and IDSs. In a typical setting, the employee machines may interact with external machines on the Internet, while the Intranet servers may receive connections only from inside the enterprise network. APTs typically exploit such connectivity of the employee machines to gain an initial foothold in the enterprise and subsequently perform lateral movement to reach high-value assets.

The most widely used APT attack vectors include sophisticated social engineering (e.g., spear phishing), browser compromises (e.g., via drive-by downloads), and web attacks (e.g., session riding) [12] that impersonate legitimate users of an enterprise host and connect to Intranet servers. Consider the following APT attack vector that highlights the need for precise provenance tracking.

Alice, an employee of an enterprise, has several tabs open on her browser. In one of the tabs, she is lured to a malicious website that contains a 0-day Java

exploit that targets an unpatched Java plugin inside her browser. The exploit instantaneously drops an executable file, which is executed and spawns a shell where the attacker can remotely enter commands. Using this shell, the attacker reads Alice's recent activities from her command history and notices a series of *git* commands to an internal GIT server. Next, the attacker executes a *git pull* command to retrieve the most recent documents and proceeds to slowly exfiltrate them to a C&C server that he controls. Alice is unaware of any of these actions.

This example showcases a drive-by-download APT attack vector [22], a common method used to gain an initial foothold in an enterprise. The next step is typically gaining control of the compromised local machine followed by further connections to other internal machines. When a step of this attack is detected, it is crucial to causally link it with the events of the initial infection and ultimately with the provenance of the input that causes the initial infection. For doing so, we need to deal with several challenges pertinent to provenance tracking, dependence explosion, abstraction of input/output, dynamics of applications, and performance issues for timely analysis. In particular, *dependency explosion* is one of the major hurdles to a fast and effective forensic investigation. This problem arises when a process receives several inputs from different sources within a short amount of time, while at the same time producing several outputs. In this context, the primary challenge is to associate the provenance of each input to the correct outputs. For instance, the average number of records generated by the audit logs is typically between 5,000 - 500,000 records per minute, only a minuscule portion of which is related to the attack [9].

2.2 Provenance Monitoring

In this section, we describe details of a provenance monitoring system which produces logs required for building a dependency graph that is used for post-attack forensics analysis. As the dependency graph is built based on information flow among system entities, we do not need to log all the system calls. Table 1 shows a summary of the most important system calls that are required for information flow tracking and provenance identification. In the table, we show different categories of system calls according to their purpose. Some system calls are responsible for the actual information flow between objects. For instance, when a new process is created via a `clone` system call, it inherits the file descriptors of its parent. Therefore, there is information flow from the parent to the child process.

A subset of the system calls (third row of Table 1) is responsible for initializing and setting up data structures rather than dealing with information flow directly. For example, the *socketpair* system call creates two sockets. *Preparatory* system calls initialize data structures, and in certain cases provide the provenance of the subsequent data. For example, by checking the *lseek* system call and considering file offsets, we only track specific offsets of a file to prevent unnecessary dependencies. *Termination* system calls deal with the destruction of objects.

Purpose	Relevant System Calls
Information Flow	*clone* (process), *fork* , *msgsnd*, *msgrcv*, *write*, *send*, *read*, *recv*, *exec*
Creation	*open*, *creat*, *dup*, *link*, *socket*, *socketpair*
Preparatory	*lseek*, *connect*, *listen*, *accept*, *bind*, *clone* (thread), *link*, *sendto*
Termination	*close*, *exit*, *exit_group*, *unlink*, *kill*

Table 1. System event types.

From	To	Relevant System Calls	Source	Destination
Process	Process	*clone* (process), *fork*, *vfork*, *rfork*, *msgsnd*	event caller	arg(s)
Process	Process	*wait*, *msgrcv*	arg(s)	event caller
Process	File/Socket	*write*, *pwrite*, *writev*, *pwritev*, *send*, *sendto*, *sendmsg*	event caller	arg(s)
File/Socket	Process	*read*, *recv*, *recvfrom*, *recvmsg*, *execl*, *execv*, *execle*, *execve*, *execlp*, *execvp*	arg(s)	event caller

Table 2. Information flow events.

Flow Types. Table 2 shows the details of information flow sources, destinations, and events. We summarize these details in the table by using only three types of objects (File, socket, process). As shown in the table, there are different kinds of information flow between system objects. These include: (*i*) from a process to another process initiated by events like *fork* and *clone*, (*ii*) from a process to a file/socket initiated by events like *write* and *send*, and (*iii*) from a file/socket to a process initiated by events such as *read*, and *receive*. In the last two columns of Table 2, we use *arg(s)* to indicate the argument(s) of system calls to refer to the object(s) that the *caller process* manipulates. In particular, depending on the system call, the argument type may be the *id* of a process, the *name* of a file, or a *descriptor* referring to a file/socket.

3 Approach

Approach in a nutshell. The goal of ProPatrol is to compartmentalize execution of long-running processes to smaller partitions by leveraging the high-level tasks extracted from audit logs. More precisely, after traces of an attack are detected, we want to perform forensic analysis and identify *who* initiated the connection (untrusted source versus legitimate local user), *how* it happened (history of the connection), and *what* system entities (processes, files, etc.) are affected. To answer these questions, we systematically follow the dependency between *system entities* (e.g., files, sockets, processes), which is constructed based on a system-wide provenance monitoring. This provenance monitoring is transparent to users, incurs negligible overhead, and does not require application instrumentation (details are discussed in 2.2). An overview of ProPatrol is shown

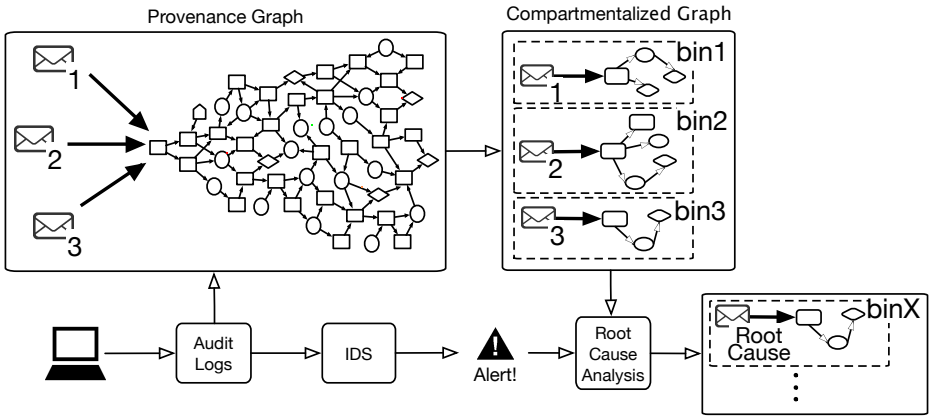

Fig. 3. Approach Overview.

in Figure 3. The provenance monitoring module constructs a dependency graph based on audit logs coming from enterprise hosts. Once an attack is detected, the compartmentalization module partitions long-running processes to smaller parts called Active Execution Units, where each Active Execution Unit relates inputs to outputs that are *truly dependent on those inputs*. For instance, in the case of a browser such as Google Chrome, each Active Execution Unit represents a single user-supplied URL. Once the Active Execution Units are determined, ProPatrol detects the root cause of the attack by performing a backward traversal from an attack point. In addition, it detects the affected objects by performing a forward traversal from the root cause. Thus, system analysts can quickly pinpoint the attack source and the affected system entities, which minimizes manual investigation efforts.

3.1 Attack Investigation

Solving the Dependency Explosion Challenge In prior approaches dealing with *dependency explosion*, a process is partitioned into smaller units of execution based on performing heavy code instrumentation or assuming that source code is available and software developers annotate it [18, 20, 21]. They use each unit to next correlate the provenance of received inputs to the produced outputs. On the contrary, we propose an approach that takes advantage of application compartments to learn a model through the analysis of the sequence of system calls it generates. Using this model, we define a partitioning scheme for applications, which assigns the provenance to the output objects of each Active Execution Unit.

In particular, we define an *active execution unit* as the segment of an application that processes an input or a set of inputs as a result of user activity. Examples of such activities include reading a new email, browsing on a new

website, and so on. We note that for the attacks that we deal with in this paper (e.g., drive-by download) we assume that such user activity is always present.

To use Active Execution Units to assign provenance, we need to be able to identify them from the system call traces when an Active Execution Unit starts and when it ends. Besides, for every system call that interacts with an object (e.g., a file or a process) and that appears between that start and that end, we associate the provenance related to the Active Execution Unit to that object. For instance, the *Active Execution Unit* of a browser such as Chrome is the website instance sending a request to Chrome's kernel process, while for an email client, the Active Execution Unit is the email that the user is currently reading.

3.2 Active Execution Unit Identification

Our methodology is based on an initial guided inference phase which exercises different applications with a variety of inputs. The inference is guided by an intuitive knowledge of what represents an active execution unit that might get compromised. For instance, for Google Chrome, active execution units are represented by visited websites, while for Thunderbird by the single emails. Such inference can be made with a high degree of certainty for several applications whose design and architecture are public knowledge, either because they are open source, or because of developer documentation.

When a new Active Execution Unit starts, some system events are generated by the part of application that is responsible for handling a new Active Execution Unit while others are commonly generated as a result of other application logic unrelated to this. Based on our observations, the latter represents a significant portion of the system calls and, during the inference phase, a source of 'noise' for correctly deriving the boundaries between Active-Execution-Units. Next, we propose a method to extract a sequence of system calls responsible for handling Active Execution Unit.

Using our previous definition, we can partition an application into several Active Execution Units, only one of which is active at any given time. To assign a system call to the correct Active Execution Unit, we must, however, be able to identify which Active Execution Unit is active when the system call is generated.

In general, the problem can be defined as follows: Given a stream of system calls arriving one at a time, and given a set of bins, each representing an Active Execution Unit, which is the active bin (that is the current active Active Execution Unit), to which a system call belongs?

We tackle this problem by creating an inference procedure, which exercises different activities to switch among Active Execution Units and user actions that cause new input to be received. More formally, for each application, we want to derive the following rules during such inference phase:

- Rule 1: $if(isObserved(S_k) \Rightarrow Bin_k = newBin();)$
- Rule 2: $if(isObserved(S_i) \Rightarrow \{k = non\_active; i = active;\})$
- Rule 3: $if(isActive(i))) \Rightarrow Bin_i = Bin_i \cup s_j$

Rule 1 deals with the creation of a new Active Execution Unit (e.g., a new tab, or a new email which comes under the user's focus). In this rule, S_k is a commonly observed sequence of system calls and their arguments when a new Active Execution Unit is created, and Bin_k represents a new empty bin. This sequence is typically manifested during the initialization of a new Active Execution Unit. Rule 2 deals with the switching tasks among different Active Execution Units. In this rule, S_i represents a commonly observed sequence of system calls when the user switches among Active Execution Unit, k and i represent the previous Active Execution Unit, which becomes inactive, and the newly activated Active Execution Unit, respectively. Rule 3 deals with assigning the current system call s_j to the currently active bin.

These rules are based on the key intuition that activities such as the creation of new Active Execution Unit or switching among existing ones are executions of the same code in an application and they usually manifest in the same system call sequences.

To derive the sequences S_k and S_i for each application, we run that application under different scenarios (e.g., open a tab, click on a link in an existing tab or open the link in a new tab, or check an email or open an email in a new window, etc.), with different actions and user inputs. For each creation or switching, we record its start by introducing a special event (e.g., a mouse click) and collect the traces of system calls and their arguments, together with additional information such as PIDs and TIDs (thread ids). Next, we compare the sequences and extract the longest common subsequence among all the traces.

More formally, given a set of system call traces, collected for the same type of activity repeated M times:

- $S_1 = (s_{11}, s_{12}, s_{1N})$
- $S_2 = (s_{21}, s_{22}, s_{2N})$
- ...
- $S_M = (s_{M1}, s_{M2}, s_{MN})$

We find the longest subsequence $S_L = (s_{l1}, s_{l2}, s_{lK})$ where each s_{li} is present in all the traces $(S_1, ..., S_M)$, and where for any two consecutive s_{li} and s_{li+1} in S_L, s_{li+1} follows s_{li} in each of the traces $(S_1, ..., S_M)$, possibly with other system calls between them. This subsequence represents a system call signature related to the specific activity, which is always present at the start of that activity. We use such subsequence as the 'boundary' between the different active execution units.

After such subsequences are learned for a number of different activities under different inputs, we introduce them in the rules previously described. In particular, every time a new Active Execution Unit's start is detected, we initiate a new bin. While this Active Execution Unit is active, we assign the connections that are created to receive input to that bin. When a switch to an existing Active Execution Unit is detected, we save the state of the current Active Execution Unit, in order to restore it once it becomes active again. If no subsequences can be identified, we conclude that the application is not suitable to be compartmentalized by our approach.

4 Implementation

4.1 Provenance Monitor

To trace the system calls, ProPatrol makes use of Systemtap [2], a very efficient Linux profiling tool designed to have near zero overhead. For each system call, we collect the timestamps, caller process id, group process id, system call name, and its arguments. We store these logs into a file to be analyzed further by the attack investigation module. Whenever a `fork` or `clone` appears in the logs, the new process or thread is monitored too.

4.2 Compartmentalization

4.2.1 Google Chrome

To isolate websites from each other, Google Chrome consists of multiple renderer processes which communicate with Chrome's kernel process. Google Chrome supports different models of how to allocate websites to the renderer processes [30]. However, by default, it creates a separate renderer process for each web page instance which user visits. Each renderer process communicates the jobs to the kernel process and receives responses via the `recvmsg` system call. Chrome's kernel process is responsible for networking and filesystem I/O tasks. These jobs include DNS requests, content download, reading and writing to the file system and so on. Consequently, ProPatrol associates the provenance of an input with a renderer process which has sent a request to Chrome's kernel process.

Active Execution Unit Selection. An active execution unit includes a renderer process and all the objects and processes initiated by the kernel in response to that renderer's messages. To be able to correctly assign and propagate the provenance, the attack investigation module must, therefore, associate each system call it receives with the correct active execution unit. We do this by taking advantage of the `recvmsg` system calls. In particular, when a `recvmsg` between the kernel and one of the renderer processes is found by the attack investigation module, we associate that system call, and all the subsequent system calls of the kernel and the renderer to the active execution unit related to that renderer. These system calls may include forking of new processes (e.g., plugins), writing to files, and so on. The new objects that are created or modified as a result of these system calls are associated with the provenance of the renderer. When a new `recvmsg` is 'seen' by the attack investigation module from a different renderer process, we switch to the active execution unit corresponding to that renderer.

4.2.2 Thunderbird

In the case of Thunderbird, each received email can be considered as a different sandbox associated with some provenance information related to the sender.

Thunderbird stores all emails in a single file called INBOX and when a user opens a specific email, this file is accessed at an offset corresponding to that email using the `read` system call.
Active Execution Unit Selection. An active execution unit in Thunderbird is defined as a set of objects to which information flows from Thunderbird as a result of reading an email. This set may include, files written by Thunderbird to the file system, browser processes forked by Thunderbird as a result of clicking on a link in an email and so on. In Thunderbird, each email is stored at a different offset in a single file, and Thunderbird uses this offset to access emails when prompted by the user. Therefore, when the attack investigation module finds a `read` system call into the INBOX file at a particular offset, it associates all the subsequent system calls with the active execution unit corresponding to the email at that offset.

4.2.3 Pidgin Chat Application

Pidgin is a chat application. Each active execution unit in Pidgin corresponds to a chat window and the objects to which information flows from that window. Similar to Thunderbird, interaction with each chat window corresponds to access to a file. However, Pidgin keeps separate files for each chat window.
Active Execution Unit Selection. *Pidgin*'s screen is separated into different chat windows each of which corresponds to a different file on disk. Therefore, an active execution unit is switched by the attack investigation module when it finds a `read` system call to the file associated with a chat window.

5 Evaluation

5.1 Enterprise Setup

To evaluate the effectiveness of ProPatrol, we simulate a set of attacks on an enterprise testbed of user workstations and Intranet servers. In particular, the Intranet consists of three Ubuntu user workstations and three Intranet servers. The Intranet servers include a GIT server used for collaborative coding within the enterprise, a web-based router, and a web server interfaced with a database that manages employees' personal information.

5.2 Graphs

To facilitate forensic analysis, ProPatrol produces visual graph representations to be used by analysts. In the Linux kernel, threads are implemented as processes that have the same process group. In the graph representation, we cluster the processes with the same process group (the process and its threads) together. The graphs depict processes and threads as ovals, sockets, and files, as well as information flow, labeled by numbers that show the sequence of events as they happened over time. Note that all the graphs we present in this section use these notations.

5.3 Summary of Results

Table 4, summarizes ProPatrol's compartmentalization capability on highlighting five different classes of attacks that target common applications such as browsers and email clients. These attacks include Remote Administration Tools (RAT) installation via an attachment on a spear-phishing email, drive-by download that exploits a Java plugin vulnerability, social engineering via an IM client, CSRF, and DNS rebinding. After the initial compromise in all these attacks, attackers pivot to one of the intranet machines that contain confidential information. At that point, an attack is detected, and system analysts use ProPatrol to find the root-cause of a connection to the corresponding sockets.

Application	**Attack**	**Root-Cause Detection?**
Email Client (Thunderbird)	RAT	✓
Browser (Google-Chrome)	Drive-by download	✓
IM Client (Pidgin)	Social engineering	✓
Browser (Google-Chrome)	CSRF	✓
Browser (Google-Chrome)	DNS Rebinding	✓

Table 4. Overview of attack investigation results.

5.4 Root-Cause Analysis

Below we present details of the five scenarios on which we evaluated ProPatrol.

5.4.1 Remote Access Trojan (RAT)

Setup. A RAT is a malicious binary that can execute several commands sent by the attacker. In this evaluation, we consider a spear-phishing email containing a RAT as an attachment. We assume that the user that receives this email is tricked into saving and executing the attachment. The attachment performs some malicious activity in the background without the user noticing it. In our evaluation, after it is downloaded to the user workstation, the RAT binary performs network scanning in the background. In this scenario, we used Nmap and a shell script that starts Nmap to mimic a RAT, which after being executed scans an internal IP address.

Attack Investigation. Using ProPatrol, we were able to find the root-cause of this RAT starting from the connections sent to Intranet servers. The sequence of system calls related to the active execution unit is processed by ProPatrol to create a causal graph depicted in Figure 5. Using this graph, a system administrator can easily infer the machine that has fallen victim to a malicious RAT that scans the network in the background.

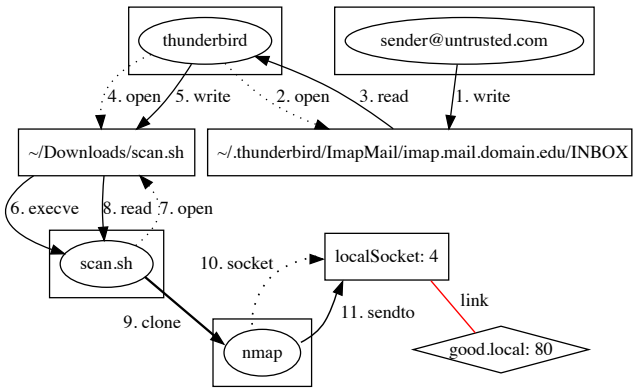

Fig. 5. Provenance graph for RAT detection scenario.

5.4.2 Drive-by Download

Setup. This attack exploits CVE-2012-4681, a vulnerability that allows a Java applet to bypass *SecurityManager* restrictions in Oracle Java Runtime Environment (JRE) version 7. We set up an external malicious web server that hosts a Java applet exploiting this vulnerability. Whenever a victim browser with the Java Plugin connects to the malicious web server, the attacker can execute arbitrary code on the victim's machine. Specifically, we install JRE version 7 on the user workstations inside our network and set up the Java plugin for the Google-Chrome browser. Then we conduct the attack on one of the user's machines. The attack proceeds as follows. The user opens Google-Chrome, and among other benign activities, he opens a tab connecting to a malicious web server. When the user workstation connects to the malicious web server, the attacker notices this event. Then using Metasploit, the attacker opens a remote shell on the user workstation. Next, as a lateral movement for accessing the Intranet servers, the attacker tries to steal the enterprise project's data from the Git Server. Using the remote shell, the attacker performs `git pull` to pull the latest codebase of the project on the Git Server. Finally, the attacker sends the codebase to the attacker's server.

Attack Investigation. A provenance graph generated by ProPatrol, starting from a backward traversal from the `git` connection to the internal git server, is visualized in Figure 6. The first edge is artificial, and we consider it to show that the socket connection on port 80 of `evil.org` is on a malicious website. `Chrome` is the initial process that is executed by the user opening Google-Chrome. Later, that process clones two threads (`Chrome_IOThread` and `Chrome_ProcesssL`) (edges labeled with 2 and 3). The `Chrome_IOThread` connects to the attacker's site and retrieves some data. The thread (`Chrome_ProcessL`) clones a set of different processes, threads, and applications for getting access to the remote shell (see edges 7-19). These intermediate steps are considered as internal mechanisms

of Metasploit and the Java exploit we are using. After accessing the remote shell at edge 21, the attacker enters a `git` command. As the Git server uses SSH protocol, an SSH process is cloned and connects to port 22 on the Intranet Git server `good.local` (edges 26-28).

For the attack that exfiltrates the code base of the Git server, using the ProPatrol, we were able to identify the root-cause starting from the point that the attacker performs lateral movements to internal servers.

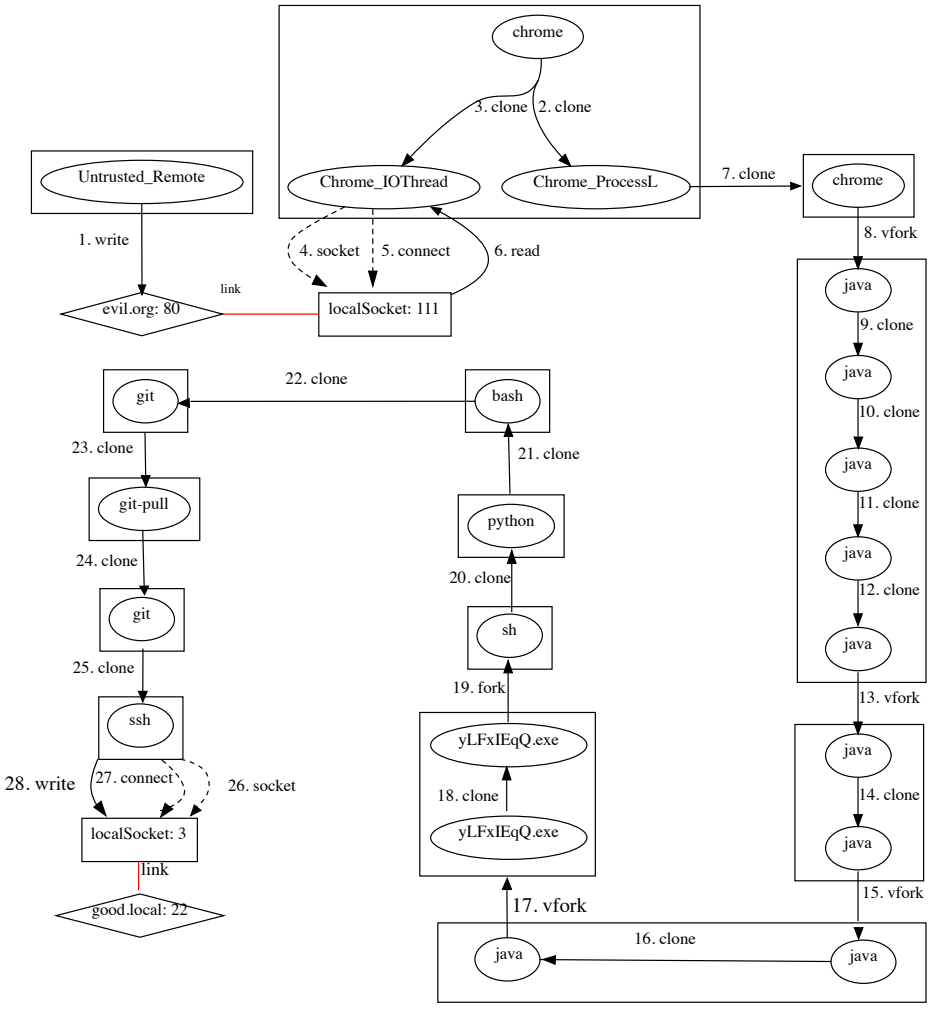

Fig. 6. Provenance graph for drive-by-download attack detection.

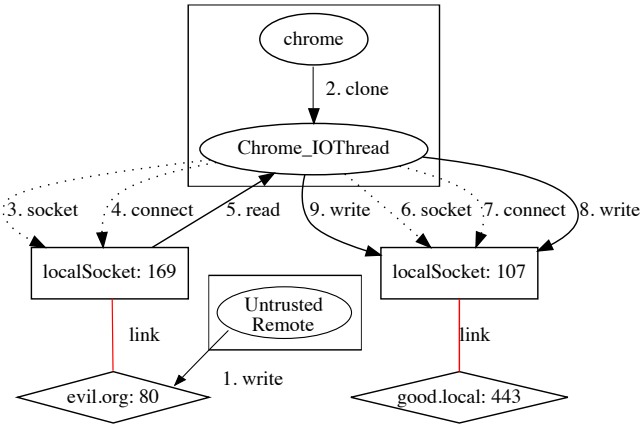

Fig. 7. Provenance graph for web attack scenario involving CSRF and DNS rebinding.

5.4.3 CSRF and DNS Rebinding

In this class of attacks, we demonstrate how we investigate CSRF and DNS Rebinding using ProPatrol. We combined these two because of the similarity of the attack vectors.

Setup. Our setup involves two malicious external web servers, one for CSRF and one for DNS Rebinding. The user workstation runs Google-Chrome with multiple open tabs. Some of the open tabs are connected to Intranet servers' sites. The user next uses one of the tabs to browse to one of the malicious websites, causing the browser to retrieve a page. Finally, the retrieved page sends a request to the Intranet server.

For CSRF attack, we tested many different scenarios. These include: (*i*) retrieving a page that contains a hyperlink to an Intranet server and a user clicks on it to access the Intranet server, (*ii*) retrieving a page that contains an element addressed by internal addresses and a JavaScript code snippet that checks the availability of those elements for port-scanning the enterprise network, and (*iii*) a JavaScript code snippet sending a malicious GET/POST request to the webpage of the internal router having a CSRF vulnerability for changing the password.

To evaluate ProPatrol against DNS Rebinding attacks, we set up a malicious external site containing a web server and a DNS server implemented with *Dnsmasq*. The DNS server has two IP addresses registered for the domain name of the web server, i.e., the IP address of the web server and the IP address of the Intranet server. When the user browser connects to this site and tries to resolve the domain name, the first IP it receives is the IP address of the web server. As a result, the browser connects to the web server and loads a webpage containing a JavaScript code that keeps requesting resources from the Intranet web server. These requests are blocked because Same-Origin-Policy prevents ac-

cessing the contents hosted on other origins. After a while, the user's browser connects to the DNS server one more time and tries to resolve the domain name again. This time, it is resolved to the Intranet web server's IP address, and the Same-Origin-Policy is circumvented —enabling the attacker's script to read the response from Intranet server and send it to the attacker's machine.

Attack Investigation. As shown in Figure 7, ProPatrol has detected the root-cause starting by a backward traversal from the attacker's attempt to send a malicious request to the Intranet server. The first edge is artificial, and we consider it to show that the socket connection on port 80 of `evil.org` is on an untrusted site. `Chrome` process is the initial process that is executed by the user opening Google-Chrome. Later that process clones a thread named `Chrome_IOThread`. This thread creates the local socket 169 and connects to the attacker's site. Then edge numbers 6 to 9 are events related to making a connection to the port 443 of the Intranet server `good.local`. Event numbers 8 and 9 transfer some untrusted information to the socket on the intranet server. Note that ProPatrol did not correlate the attack to the other valid requests going to Intranet servers in other applications or other tabs of the browser.

5.4.4 Instant Messaging Client

Setup. To demonstrate how ProPatrol forensically investigates attacks targeting Instant messaging clients, we considered the *Pidgin* IM client. Pidgin maintains individual conversation history in separate files for each contact of a user. We add a google account in the Pidgin that contains a list of added buddies. Then we start chatting with some of them. For each buddy, there is a chat communication that is stored in a separate file from the other conversations.

Attack Investigation. The provenance graph for detecting an attack that happens via an IM client is shown in Figure 8. In the chat communication with username2, username1 receives a chat messages with a link to a vulnerable Intranet server. When username1 clicks on the malicious link, a Google-Chrome process is forked by pidgin, and a connection to the Intranet server is initiated. At this moment, the active execution unit corresponds to the chat window with username2. Therefore, ProPatrol detects username2 as the root-cause of this attack.

5.5 Effectiveness

Table 9 shows the effectiveness of ProPatrol pertaining to event volume reduction. In this table, the second column shows the number of system calls generated by each application for its initialization which is the duration from the start of the application until it loads completely. Execution of each one of these applications could be compartmentalized to smaller bins depending on user activities. For instance, a new bin is created when a user opens a new tab in Chrome, or opens a new chat window in Pidgin, or reads a new email in Thunderbird. The third column shows the number of events assigned to each bin on average.

Application	Initialization Syscalls	Average Bin Syscalls	Average active execution unit Syscalls
Google-Chrome	200K	14K	< 50
Thunderbird	91.5K	8K	< 20
Pidgin	20.5K	1K	< 15

Table 9. Effectiveness of attack summaries.

For example, in the case of Google Chrome, if a user opens 10 tabs, the size of provenance graph would be about $200K + 10 \times 14K$. In any attack, typically only one bin is responsible for the attack, and ProPatrol successfully identifies it. The fourth column shows the final number of events that ProPatrol shows to the system analyst after detecting the root-cause. As evidenced by this table, ProPatrol can achieve orders of magnitude reduction in event volume.

5.6 Performance Overhead

Table 10 shows the performance overhead introduced by ProPatrol and the time required for generating the attack graphs. As shown in the second column of Table 10, we calculate the average time for a single system call per scenario in microseconds. The third column shows the overhead (in percentage) by the monitoring infrastructure (which includes Systemtap and the provenance graph building module), which on average is 1.1.%. We set up the graph generation module on a 32 bit Ubuntu OS, Quad-Core 2.4 GHz Intel Xeon Processor with 10 GB RAM. The time (in seconds) this module took for highlighting the root-cause of an attack is shown in the fourth column of Table 10 showing a very minimal overhead. Overall, both the graph generation module and Systemtap

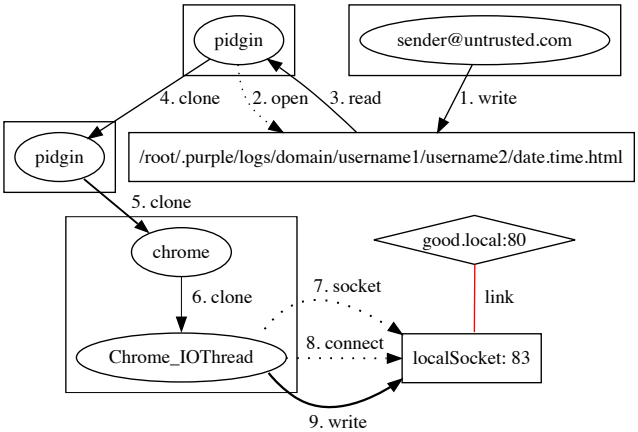

Fig. 8. Provenance graph for an attack scenario that targets an IM client.

Scenario	Avg Event time (µs)	Provenance Monitor Overhead (%)	Graph Generation time (sec)
RAT	8.87	0.72	0.12
Drive-by download	16.37	1.57	0.001
Social engineering	9.87	0.16	0.0004
CSRF	8.10	1.14	0.03
DNS Rebinding	15.71	1.88	0.08

Table 10. Overhead for the Provenance Monitor and Graph Generation Time.

incur negligible overhead due to the coarse-grained provenance tracking underlying ProPatrol.

6 Related Work

6.1 System-wide Provenance Collection

SPADE [6] and PASS [25] are operating system level provenance systems. SPADE hooks into the audit subsystem in the Linux kernel to observe program actions whereas PASS intercepts system calls made by a program. Both of these systems observe application events such as process creation and input/output, which is then used to find out the relationship between data sets. LineageFS [28] modifies the Linux kernel to log process creation and file-related system calls in *printk* buffer. A user-level process reads this buffer periodically to generate lineage records. Similar approaches to collect provenance are Hi-Fi [27] and LPM [3] — these are kernel level systems that track the provenance of system objects. While they provide a secure and application-transparent way of collecting provenance, they do need provenance awareness at the application level in order to counter the dependence explosion problem. Moreover, SLEUTH [9] and HOLMES [23] use kernel audit logs for real-time attack detection and forensics, which could benefit from the light-weight compartmentalization approaches such as ProPatrol to improve accuracy.

6.2 Information Flow Tracking

Some past work (such as [29,33]) proposed information flow tracking at processor-level with manufacturer support. Some others (e.g., [14, 26] perform binary rewriting at runtime to instrument machine code with additional instructions that update shadow memory. Xu et al. [31] employ source code transformation by instrumenting C code with additional code that can handle flow tracking. Being fine-grained techniques, they offer good precision in tracking the source of enterprise activity. However, all these approaches impose a high overhead. For instance, [14] imposes a 3.65x slowdown factor. Another line of work also uses techniques to decouple taint tracking from program execution [4, 11, 17, 24].

In the coarse-grained tracking front, Backtracker by King et al. [15] is one of the first works in this area that introduced the notion of dependency graphs.

The same authors extended Backtracker in [16] with support for multi-host dependencies, forward tracking and correlating disconnected IDS alerts. To reduce the size of audit logs, different methods [8, 10, 19, 32] are proposed leveraging graph abstraction, garbage collection, or compactness techniques.

6.3 Execution Partitioning

Execution partitioning techniques are proposed for dividing the execution of long-running programs into smaller units, resulting in a better forensic analysis. BEEP is a closely related approach to ProPatrol. BEEP is based on the notion of independent units whereby a long-running program is partitioned into individual units by monitoring the execution of the program's event-handling loops, with each iteration corresponding to the processing of an independent input/request. An essentially backward forensic tracing system, BEEP, is suitable for programs that tend to have independent loop iterations. Ma et al. [21] introduced ProTracer, a lightweight provenance tracing system that only captures system calls related to taint propagation. ProTracer records the history of objects by logging important events. It utilizes an instrumentation technique called BEEP [18] for partitioning an execution into smaller units. BEEP [18] and ProTracer [21] use training and code instrumentation to divide execution to multiple iterations of the main loop in a program. Another related work, MPI [20] relies on users to annotate the application's high-level task structures to enable semantic-aware execution partitioning.

7 Conclusions

In this paper, we presented ProPatrol, as a compartmentalization approach for doing more accurate and timely root-cause analysis. ProPatrol uses a lightweight provenance monitoring system to effectively perform forward/backward tracking. Our evaluation shows that the tracking system operates with a very minimal overhead of less than 2%. We demonstrated in an enterprise setting that ProPatrol is able to detect the root-cause of a broad class of APT vectors such as spear phishing, drive-by downloads, RATs, CSRF, and DNS Rebinding attacks.

Acknowledgements

This work was primarily supported by DARPA (under AFOSR contract FA8650-15-C-7561) and in part by SPAWAR (N6600118C4035), and NSF (CNS-1514472, and DGE-1069311). The views, opinions, and/or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense, National Science Foundation or the U.S. Government.

References

1. Multiprocess firefox. https://developer.mozilla.org/en-US/Firefox/Multiprocess_Firefox
2. Systemtap. https://sourceware.org/systemtap/
3. Bates, A., Tian, D.J., Butler, K.R., Moyer, T.: Trustworthy whole-system provenance for the linux kernel. In: 24th USENIX Security Symposium (USENIX Security 15). pp. 319–334 (2015)
4. Chow, J., Garfinkel, T., Chen, P.M.: Decoupling dynamic program analysis from execution in virtual environments. In: USENIX 2008 Annual Technical Conference on Annual Technical Conference. pp. 1–14 (2008)
5. Corporation, M.: Apt1: Exposing one of china's cyber espionage units. Tech. rep. (2013)
6. Gehani, A., Tariq, D.: Spade: support for provenance auditing in distributed environments. In: Proceedings of the 13th International Middleware Conference. pp. 101–120. Springer-Verlag New York, Inc. (2012)
7. Goel, A., Po, K., Farhadi, K., Li, Z., De Lara, E.: The taser intrusion recovery system. In: ACM SIGOPS Operating Systems Review. vol. 39, pp. 163–176. ACM (2005)
8. Hassan, W.U., Lemay, M., Aguse, N., Bates, A., Moyer, T.: Towards scalable cluster auditing through grammatical inference over provenance graphs. In: Network and Distributed Systems Security Symposium (2018)
9. Hossain, M.N., Milajerdi, S.M., Wang, J., Eshete, B., Gjomemo, R., Sekar, R., Stoller, S., Venkatakrishnan, V.: SLEUTH: Real-time Attack Scenario Reconstruction from COTS Audit Data. In: 26th USENIX Security Symposium (USENIX Security 17). pp. 487–504. USENIX Association, Vancouver, BC (2017)
10. Hossain, M.N., Wang, J., Sekar, R., Stoller, S.D.: Dependence-preserving data compaction for scalable forensic analysis. In: 27th USENIX Security Symposium (USENIX Security 18). pp. 1723–1740. USENIX Association, Baltimore, MD (2018)
11. Ji, Y., Lee, S., Downing, E., Wang, W., Fazzini, M., Kim, T., Orso, A., Lee, W.: Rain: Refinable attack investigation with on-demand inter-process information flow tracking. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 377–390. ACM (2017)
12. Johns, M., Winter, J.: Protecting the intranet against "javascript malware" and related attacks. In: Detection of Intrusions and Malware, and Vulnerability Assessment, pp. 40–59. Springer (2007)
13. Keizer, G.: Enterprises to get locked-down Windows 10 in six months. https://www.computerworld.com/article/3232749/microsoft-windows/enterprises-to-get-locked-down-windows-10-in-six-months.html, [Online; accessed 08-October-2018]
14. Kemerlis, V.P., Portokalidis, G., Jee, K., Keromytis, A.D.: libdft: Practical dynamic data flow tracking for commodity systems. In: ACM SIGPLAN Notices. vol. 47, pp. 121–132. ACM (2012)
15. King, S.T., Chen, P.M.: Backtracking intrusions. In: ACM SIGOPS Operating Systems Review. vol. 37, pp. 223–236. ACM (2003)
16. King, S.T., Mao, Z.M., Lucchetti, D.G., Chen, P.M.: Enriching intrusion alerts through multi-host causality. In: NDSS (2005)
17. Kwon, Y., Wang, F., Wang, W., Lee, K.H., Lee, W.C., Ma, S., Zhang, X., Xu, D., Jha, S., Ciocarlie, G., et al.: Mci: Modeling-based causality inference in audit

logging for attack investigation. In: Proc. of the 25th Network and Distributed System Security Symposium (NDSS) (2018)
18. Lee, K.H., Zhang, X., Xu, D.: High accuracy attack provenance via binary-based execution partition. In: NDSS (2013)
19. Lee, K.H., Zhang, X., Xu, D.: Loggc: garbage collecting audit log. In: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. pp. 1005–1016. ACM (2013)
20. Ma, S., Zhai, J., Wang, F., Lee, K.H., Zhang, X., Xu, D.: Mpi: Multiple perspective attack investigation with semantics aware execution partitioning. In: USENIX Security (2017)
21. Ma, S., Zhang, X., Xu, D.: Protracer: Towards practical provenance tracing by alternating between logging and tainting (2016)
22. Mathew, S.J.: Social engineering leads apt attack vectors. http://www.darkreading.com/vulnerabilities-and-threats/social-engineering-leads-apt-attack-vectors/d/d-id/1100142?
23. Milajerdi, S.M., Gjomemo, R., Eshete, B., Sekar, R., Venkatakrishnan, V.: HOLMES: Real-time APT Detection through Correlation of Suspicious Information Flows. In: Proceedings of the IEEE Symposium on Security and Privacy. IEEE (2019)
24. Ming, J., Wu, D., Wang, J., Xiao, G., Liu, P.: Straighttaint: Decoupled offline symbolic taint analysis. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. pp. 308–319. ACM (2016)
25. Muniswamy-Reddy, K.K., Holland, D.A., Braun, U., Seltzer, M.I.: Provenance-aware storage systems. In: USENIX Annual Technical Conference, General Track. pp. 43–56 (2006)
26. Newsome, J., Song, D.: Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software (2005)
27. Pohly, D.J., McLaughlin, S., McDaniel, P., Butler, K.: Hi-fi: collecting high-fidelity whole-system provenance. In: Proceedings of the 28th Annual Computer Security Applications Conference. pp. 259–268. ACM (2012)
28. Sar, C., Cao, P.: Lineage file system. Online at http://crypto. stanford. edu/cao/-lineage. html (2005)
29. Suh, G.E., Lee, J.W., Zhang, D., Devadas, S.: Secure program execution via dynamic information flow tracking. In: Acm Sigplan Notices. vol. 39, pp. 85–96. ACM (2004)
30. Team, T.C.: Chromium Project Process Model. https://www.chromium.org/developers/design-documents/process-models, [Online; accessed 05-October-2018]
31. Xu, W., Bhatkar, S., Sekar, R.: Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In: Usenix Security. pp. 121–136 (2006)
32. Xu, Z., Wu, Z., Li, Z., Jee, K., Rhee, J., Xiao, X., Xu, F., Wang, H., Jiang, G.: High fidelity data reduction for big data security dependency analyses. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 504–516. ACM (2016)
33. Yin, H., Song, D., Egele, M., Kruegel, C., Kirda, E.: Panorama: capturing system-wide information flow for malware detection and analysis. In: Proceedings of the 14th ACM conference on Computer and communications security. pp. 116–127. ACM (2007)