2023

Path-Safe :Enabling Dynamic Mandatory Access Controls Using Security Tokens

James P. MacLennan
Wright State University

Path-Safe: Enabling Dynamic Mandatory Access Controls Using Security Tokens

A Thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Cyber Security

by

JAMES P. MACLENNAN
B.S.C.E., Wright State University, 2000

2023
Wright State University

WRIGHT STATE UNIVERSITY

COLLEGE OF GRADUATE PROGRAMS AND HONORS STUDIES

July 24, 2023

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY James P. MacLennan ENTITLED Path-Safe: Enabling Dynamic Mandatory Access Controls Using Security Tokens BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF Master of Science in Cyber Security.

Junjie Zhang, Ph.D.
Thesis Director

Thomas Wischgoll, Ph.D.
Interim Chair, Department of Computer Science and Engineering

Committee on
Final Examination

Junjie Zhang, Ph.D.

Krishnaprasad Thirunarayan, Ph.D.

Lingwei Chen, Ph.D.

Shu Schiller, Ph.D.
Interim Dean, College of
Graduate Programs & Honors Studies

ABSTRACT

MacLennan, James P. M.S.C.S, Department of Computer Science and Engineering, Wright State University, 2023. *Path-Safe: Enabling Dynamic Mandatory Access Controls Using Security Tokens.*

Deploying Mandatory Access Controls (MAC) is a popular way to provide host protection against malware. Unfortunately, current implementations lack the flexibility to adapt to emergent malware threats and are known for being difficult to configure. A core tenet of MAC security systems is that the policies they are deployed with are immutable from the host while they are active. This work looks at deploying a MAC system that leverages using encrypted security tokens to allow for redeploying policy configurations in real-time without the need to stop a running process. This is instrumental in developing an adaptive framework for security systems with a Zero Trust based approach to process authentication. This work also develops *Path-Safe* , a MAC security system that focuses on protecting filesystem access from unauthorized processes and malware. We show that our security system can mitigate real-world malware threats with low overhead and high accuracy.

Contents

List of Figures

List of Tables

Acknowledgement

I would like to take this opportunity to extend my sincerest thanks to my advisor Dr. Junjie Zhang. Without his guidance and insights, this work would not be possible. I would also like to express my gratitude to my committee members for their thought-provoking questions and for making my defense such a wonderful experience; thank you.

I would like to give special thanks to my wife and children. Your unwavering support and patience during this process have meant the world to me. I truly appreciate how blessed I am to have such a great support system at home.

Introduction

The use of the Linux OS in areas such as embedded computing, Industrial Control Systems (ICS), and the banking industry has made it a frequent target for malware development. Potentially more concerning is that established malware strains that historically attacked Windows-based platforms are now adding Linux OS support. A prime example of this is the Industroyer2 variant of malware that was used against Ukraine in April of 2022 to damage the power grid by targeting ICS computers throughout the country [14, 15]. The new versions of Industroyer2 are highly tailorable to different victim environments and can now also be used to attack victims using the Linux OS, utilizing common Linux commands such as `shred` and `dd` [14].

Another example is the ransomware family RansomExx. In late 2020, RansomExx strains were discovered targeting Linux. This marked the first known time that a major Windows ransomware variant expanded to Linux [19, 20, 18]. Such security challenges are exacerbated by the fact that older malware strains are being updated and put back into service. In the first half of 2022, there was a 254% increase in activity from a Linux Trojan called XorDdos [16, 17], a malware that was first discovered in 2014.

In light of the recent advancements in Linux-focused malware, this thesis investigates pitfalls with common malware mitigation strategies. It proposes a nascent methodology to address the changing threat environment for Linux hosts. In this chapter, we will first provide a brief overview of common strategies currently in place and some of the deficiencies in their design. Next, we present our contributions to enhancing security system design

and what our objectives are during this research. Finally, we provide an organizational overview of the remainder of this thesis.

1.1 Background

One mitigation strategy is to deploy intrusion detection systems such as antivirus tools, which, however, fall short of keeping up with new malware variants enabled by newer programming languages [20], sophisticated obfuscation techniques, etc. A more fundamental strategy is to enforce access control that can prevent malware from accessing sensitive information even if it infects the victim host. A salient example of this strategy is mandatory access control (MAC), which has been implemented by SELinux [12] and AppArmor [13]. These MAC systems group processes into domains and files into types and then define the specific access between the domains and file types [11], which is known as security policies. MAC allows defining fine-grained access control over kernel objects which can be used to prevent malware.

Nevertheless, these MAC systems lack the flexibility to update their policies, making it hard to adapt to new vulnerabilities or configuration requirements on the host. Specifically, security policies are managed by administrators rather than individual users. On the surface, this provides protection against malicious software or user misconfiguration of the security system. Practically, this makes policy updates even more cumbersome: they require updating the security policies, stopping the current running protection, pushing the updated policies, and redeploying the protections.

Other approaches, such as SECCOMP and eBFP [10], have been proposed to enforce access control by vetting system calls. Specifically, they restrict a binary's access to kernel objects by filtering the system calls exposed by the kernel. Therefore, these methods require benign binaries to be instrumented or run inside a container. Unfortunately, these methods fail to adapt to changes made in the underlying operating systems. Specifically, when

the system call interface changes, binaries need to be re-instrumented. These approaches require a detailed knowledge of every system call interface and how they interact with the underlying kernel objects.

1.2 Contributions

In order to offer flexible mandatory access control in real-time environments to adapt to evolving threats, we have designed, implemented, and evaluated a novel system, namely *Path-Safe* with the following objectives:

- Flexible: it eliminates the need to manually label kernel objects or to instrument binaries.
- Real-Time: it seamlessly adapts to policy changes without interrupting running processes and underlying operating systems.
- Lightweight: it incurs very low-performance overhead.

We have designed a new method of implementing MAC systems that utilizes security tokens to enable dynamic process-level security mitigations, fusing both static security policies and external sensor input. Our work builds the foundational elements that enable MAC security systems to adapt to the current threat environment and focuses on simplifying the definition of how to protect file and inode kernel objects. Our objectives are to provide a system that can modify access control in real-time and allow other decision-makers to be layered into the adjudication of access to monitored kernel objects. This establishes the framework to define adaptable layered protections not made solely on an immutable policy-based approach to kernel object access. To overcome deficiencies in signature-based approaches, *Path-Safe* focuses not on what is run but on what protected directories it has access to.

We also are proposing a MAC solution that does not rely on function signatures and kernel APIs but builds off the Linux Security Module (LSM) framework providing a simple-to-understand Linux kernel module. Our approach doesn't prevent creating a statically type enforcement system; it just allows the access to be changed dynamically and adapt to the current threat. Because we use access tokens, a per-process level analysis can be done, and a security manager could change access for two processes running the same binary. This enables other security systems, such as IPS/IDS systems or other dynamic monitoring algorithms to be used to feed the security manager's decision making.

To summarize, this paper has introduced the following contributions to moving forward the state-of-the-art MAC systems for the Linux OS. We have designed a new method of implementing MAC systems that utilizes security tokens to enable dynamic process-level security mitigations, fusing both static security policies and external sensor input. We have implemented our system and make it open source via https://github.com/jaypm007/Path-Safe. We have demonstrated the methodology of our *Path-Safe* MAC security system against real-world malware samples and benign applications, where the evaluation results have shown our system can stop all malicious attempts at a low overhead of nominally between 2 and 4%.

1.3 Organization

This thesis is organized into eight chapters. Chapter 1, Introduction, provides an overview of the malware landscape targeting Linux as well as the objectives for our novel approach to security system design. Chapter 2, Related Work, examines existing methodologies for malware mitigation and identifies deficiencies in their approaches. Chapter 3, Threat Model, defines the threat model we use to develop our MAC system and provides the analysis to simplify the model, isolating the critical threats that need to be addressed by our work. Chapter 4, System Design and describes the design methodology used to develop

Path-Safe in order to address the results of our threat model analysis. Chapter 5, Implementation, illustrates some of the implementations for the core components of our system. Chapter 6, Evaluation, provides a detailed analysis of how our system was evaluated for accuracy and we measured performance overhead. This chapter also provides the results of our system when subjected to live malware samples. Chapter 7, Limitations and Potential Solutions, presents some limitations of the current system and some potential solutions. Chapter 8, summarizes our key contributions to MAC design.

Related Work

This chapter provides a detailed look at some of the approaches used by related works to this thesis. Each section investigates a specific methodology used to develop host-based security systems. We will highlight some of the limitations of each methodology and discuss related works for each.

2.1 Mandatory Access Controls

Mandatory Access Control allows fine-grained access control over kernel objects which can be used to prevent Malware and come installed in many operating systems. Common Mandatory Access Control (MAC) implementations, such as SELinux and AppAmour, are designed so that security policies are not modifiable by the user. Security policies are deployed and managed by some form of security policy administrator and are immutable, while the MAC is managing access to system resources. When changes are required, this requires security policies to be updated and redeployed. These MAC systems utilize domain type enforcement(DTE) models, grouping processes into domains, files into types, and defining the specific access between the domains and file types [11]. MAC allows defining fine-grained access control over kernel objects which can be used to prevent Malware; however, they are typically designed so that security policies are not modifiable by the user. This is a core tenet of the most popular MAC systems available for Linux [12, 13]. Security policies are deployed and managed by some form of security policy administrator

and are immutable while the MAC is managing system resources [12, 13]. Unfortunately, when policy changes are required, this requires: updating the security policies, stopping the current running protection, pushing the updated policies, and redeploying the protections [12, 13]. This is problematic in a dynamic threat environment, where static policies cannot adapt to new vulnerabilities or configuration requirements on the host.

To support the current evolution of MAC security systems, complex labeling schemes have been developed to provide a comprehensive access model for kernel objects. Whether the MAC utilized a monolithic whole system type approach to labeling objects and files [12] or utilizes a per-process type labeling system [13], the complexity of defining DTE policies can lead to configuration errors, an unusable systems, or simply an abandoned security control. A fundamental problem with DTE-based protections is that they require a high-resolution understanding of the system resource required to operate a priori to deploying a policy. This creates pressure on application developers to understand multiple complex labeling systems that need to be updated as the software or operating system changes. This provides a security system that does not adapt to the current threat environment.

2.2 System call filtering

Other approaches such as SECCOMP and eBPF enable software developers to restrict a binary's access to kernel objects by filtering the system calls exposed by the kernel API. While these approaches do limit a processes' access to specific kernel objects, this puts a lot of responsibility on the software development team and requires a good understanding of what each system call exposes. Unfortunately, any system call function signatures and kernel APIs changes made to the Linux kernel require a developer to re-profile an executable.

Instead of relying on function signatures and kernel APIs, our approach is to use the LSM framework. System call function signatures and kernel APIs change and can make

it difficult to keep up(e.g., SECCOMP and eBPF). This also requires the developer to understand of how and what functions should be allowed to execute. Instead, we utilize LSM hooks that look at kernel object actions, such as file opening. This lives entirely in the kernel space and allows us to monitor the file actions we want to control. In the case of *Path-Safe* , we want to focus on file opening, renaming, and deleting files. Process whitelisting approaches, such as the UShallNotPass project, uses whitelists and access control lists (ACL) to control access to Cryptographically Secure PRNG (CSPRNG) resources via system call monitoring, have been proposed to control access to specific kernel APIs. This can prevent both malware and benign application from accessing encryption primitives but does not prevent access to the files on a host, preventing malware from destroying filesystem data. Pure white list implementations are also susceptible to process renaming techniques such as: using the prctl system call to modify /proc/status, or modifying the first command line argument to modify /proc//cmdline [9]. The *Path-Safe* system is immune to these types of techniques because we evaluate a process' access via security tokens that are pinned to the process before it is executed. Modifying a process' metadata in userspace will not influence access control to the underlying filesystem.

2.3 Signature-Based Detection

A commonly used malware mitigation technique is signature-based detection. To develop a signature-based detection, static analysis is performed on a large corpus of malware samples to develop a repository of malware signatures. A signature-based system then monitors the binaries files on the host system and compares them against this collection of signatures, looking for malware on the victim's machine. The success of this type of approach relies heavily on the ability to develop a malware signature profile before it infects the host system. Unfortunately, signature-based malware mitigation techniques are susceptible to circumvention by intelligent malware. For example, Linux-based malware can leverage

runtime packing, a sophisticated obfuscation technique that can prevent static analysis and reverse engineering of a malware sample [9], or use newer programming languages [20] to add difficulty to signature development. The reliance on known malware samples creates an implicit delay for signature-based systems to adapt to new threats. This requires system maintainers to discover new malware variants, statically analyze and develop a signature, and then add the new signature database for each of the newly discovered variants. This process becomes cumbersome when you consider that upwards of 200,000 samples are discovered every month [9].

A signature-based approach also requires storage for signatures and can impose significant performance overhead if used for process monitoring [1]. Some solutions, such as RanDroid, propose offloading the static analysis to a remote server during software installation [7] in an attempt to lessen the burden of maintaining a local repository. This approach may remove the requirement of pushing new signatures to hosts, but it comes with many drawbacks. Remote processing requires an active network connection which is not always available. This is also susceptible to denial-of-service type attacks, where network resources are overwhelmed with traffic and cannot handle all network requests. This approach also lacks the ability to perform continuous system monitoring because it would be infeasible to continuously send a copy of all installed binaries to a remote server. Finally, using a remote server for validation cannot protect against a time-of-check time-of-use(TOCTOU) race condition, where a binary is modified after it has been approved and installed.

2.4 Behavioral-Based Detection

Another approach to malware mitigation is to develop behavior-based techniques which perform dynamic analysis of all running processes in order to monitor for malicious behavior on the host. A behavior-based approach requires training a behavioral model that

can discern the differences between benign and malicious activities on a host, such as: how files are encrypted, deleted, and removed [8]. Unlike a signature-based approach, a behavior-based approach may detect new strains of malware and even potentially detect zero-day attacks. Unfortunately, with any model-based system, there exists the possibility that benign applications will generate a false-positive and be detected as malware as well.

Other challenges to developing an accurate behavioral model exist. For example, when analyzing malware, running it at different privilege levels or network access may produce different behavior, making it difficult to execute all its code paths [9]. Even the specific API calls that are monitored may rely on specific kernel variants [8] and not generalize well to operating system API changes. Similarly, like its signature-based counterpart, the behavioral models need to be updated and redeployed to maintain a current and updated security posture.

Performance is also another area of concern with behavioral-based systems. Dynamic analysis requires both real-time monitoring of the kernel APIs and the subsequent model inference to make an informed decision. This can introduce significant processing overhead, resulting in malware that runs unobstructed until the system catches up and is not conducive to embedded systems that have fewer processing resources. Another concern is how similar benign and malicious actions can look, leading to both false positive and false negative defections.

2.5 Moving Target Defenses

Another area of research includes employing moving target defense (MTD) countermeasures. The goal of an MTD-based countermeasures are to provide controlled configuration changes to a system to increase an attacker's uncertainty [5]. In the case of ransomware, one method introduced in [3] is shuffling the file extension to prevent the discovery of target files. Routinely shuffling the extensions would increase the difficulty for ransomware algo-

rithms to adapt to the defense and discover the victim's files to encrypt [3]. Unfortunately, techniques like extension shuffling fail to prevent malware from accessing the underlying files and only require them to do a more thorough inspection of the files on the system (e.g., by looking at file header magic.)

Another proposed MTD-based countermeasure set to set traps for malware to expose themselves. Decoy resources are strategically placed in the filesystem and monitored to discover when malware attacks the filesystem [8]. For example, when malware opens a decoy file, the system can take appropriate measures to stop the attack. Of course, the effectiveness of such a technique relies on malware operating on the decoy files. This creates a potentially long delay in the system response, compromising many of the victim's files before the malware hits a decoy file [8].

Threat Model

In this chapter, we define and analyze the threat model used to identify the critical actions malware must execute to successfully attack a victim. We used this analysis to provide the motivation for the design objectives of our security system. For this thesis, we consider the following malware model with three typical malware activities including [6]:

1. Infecting the victim's host
2. Establishing a command-and-control(C2) channel with the attacker
3. Executing the attack

The following three sections will discuss each activity respectively.

3.1 Infecting the Victim's Host

For any malware to be effective, it must first infect a target host. A common vector for infecting a victim's host is social engineering, where a victim will infect their host by installing or downloading malware from malicious emails, SMS, and Instant Message links [2]. From a security system perspective, social engineering creates some interesting challenges. First, the user is presumed to have permissions to install or download the malware to the host. Second, the applications the user uses to download the malware in a social engineering attack can be considered benign. Essentially, in this scenario, the user

has accepted the risk of downloading or installing the malware. Any countermeasure must allow the user to inherit the risk of downloading malware but also must protect other system resources when the malware is executed.

Another vector for malware infection is Drive-by-Download, which refers to a malicious link or advertisement that, when clicked, downloads and installs malware [2]. Legitimate websites may host ads that malware authors have purchased through real-time bidding or directly buying ad space [23]. Some malware, such as the Ryuk ransomware family, use custom droppers installed from other malware to directly install their malware [2]. Others act like worms and self-propagate by sending spam via email or malicious SMS messages to addresses in the victim's address book. These attack vectors all require that an executable binary or malicious script be saved to disk before executing an attack on a host. Unlike with social engineering, the installation of malware is done surreptitiously from the user through executable.

3.2 Establishing a C2 Channel

Many malware variants attempt to establish communication between the infected host and the attacker. For example, Cryptographic ransomware requires a clear C2 channel to provide victims with instructions, receive payment, and potentially key management. The methods and techniques used to establish C2 range from simply hard-coded IP addresses and domains of a C2 server to using dynamically generated domain generation algorithms (DGAs) [2]. Malware can exploit zero-day exploits, which are software vulnerabilities vendors have yet to discover, and establish this channel. Two examples of this are the WannaCry and BadRabbit ransomware strains which utilized the ExternalBlue zero-day vulnerability in the Window Server Message Block (SMB) service to gain C2 over its victims' machines [3]. After establishing a C2 channel, malware may attempt to conceal their C2 communications by leveraging network obfuscation through anonymizer services such

as Tor and Invisible Internet Project (I2P) or use bulletproof hosting sites [23]. Establishing the C2 channel is an extension of the control flow path the malware takes and requires the malware to have access to the underlying filesystem to perform many of the tasks that will be requested over the C2 channel.

3.3 Executing the attack

Any successful malware attack requires that the malware software be executed on the host's system and requires the same resources a benign application needs. For example, malware that is dynamically linked requires access to the shared libraries on a system. When the binary is executed, the resulting process needs the ability to open and read the shared libraries it needs in order to run. Some malware is statically linked, providing malware authors some reverse engineering protection and gains in portability [9]. When a binary is statically linked, it runs the risk of not being compatible with the infected machine's kernel application binary interface (ABI). Malware authors could mitigate the portability issues of statically linked binaries by using the libc system call wrappers. These wrappers enable the binary to maintain compatibility with the operating system's ABI and not expose the library calls used outside the libc library.

Once operational, a malware binary will execute its control flow path and begin its attack on the victim's host machine. File management is an essential aspect of all malware design, but perhaps none more pertinent than ransomware. To launch a successful ransomware campaign, attackers must effectively manage cryptographic keys, encrypt files and keep the victim's computer usable to allow for ransoms to be paid [6, 3]. Even after paying a ransom, an attacker could request a second ransom to prevent exposing the victim's data or even resell or release the data on a public site. Ironically, a good defense against data exfiltration is encrypting the data at rest(DAR) [2, 6].

A common operational goal of many malware attacks is destroying a victim's data

during an attack. For example, ransomware attackers must decide on the actions to take when a victim does not pay the ransom. Some ransomware variants delete a victim's encrypted files when they do not pay, showing that the ransomware is not an idle threat and needs to be paid [8]. Malware variants such as Industroyer2 destroy victims' files using utilizing common Linux commands such as `shred` and `dd` [14]. Malware may also attempt to circumvent path-based rules by attempting to rename a file using the mv command or issuing an inode_rename system call. For example, XorDDos attempts to rename system binaries to circumvent rule-based protections [16]. In both examples, we see that the adversary opens files and has the ability to rename or move files indiscriminately.

3.4 Threat Analysis

After considering the threat model described above, we see that malware is essentially an authorization problem. We can extrapolate from our model six kernel object actions malware needs to operate. They are as follows:

Unauthorized binary execution - Malware can be executed both directly by a victim and by vulnerabilities in benign applications. When any binary is initially executed, even before it is loaded into memory, the calling process must request for the kernel to initiate the process into memory through the exec family of system calls. It is at this point in the life cycle of the malware that the unauthorized access occurs and where a security system needs to evaluate that calling process, deciding what kernel resources it should have access to and permissions for.

Unauthorized file open - The previous sections establish many reasons why malware would need access to a host filesystem. For every file operation where malware creates, reads from, or writes to a file, it must first obtain a valid file descriptor provided by the kernel. This file descriptor may be requested through the use of many different

system calls, but at its core, the request for file resources to be opened needs to be adjudicated.

Unauthorized renaming an inode hardlink - We also see that malware may move or rename files for many reasons, including as a countermeasure to avoid current security monitoring. It is important to identify that moving or renaming a file is an inode operation, where the name of a file's inode hardlink is modified. Malware requests to access and modify inode hardlinks must be denied.

Unauthorized unlinking of an inode hardlink - To remove a file from the filesystem, an attacker must unlink the file's inode entry from the filesystem using an unlink system call. More thorough malware may first attempt to open the file and overwrite it multiple times to ensure that it is difficult to forensically recover the file from disk [1, 14]. Regardless of the malware's attack vector to destroy data, malware needs to have the ability to unlink a file to remove it from the filesystem. It should be noted that when the last hardlink is removed from an inode, the filesystem is free to reclaim the inode and reuse it. In such cases, the underlying data that it points to can also be modified, making recovery after a malware attack much more difficult.

Unauthorized linking of an inode hardlink - Every file must have one or more hardlinks, which are used to associate a file with a name. Unlike symlinks, every hardlink of a file points to the same inode and is not simply a reference to the original name. This distinction is important because allowing a new hardlink to be created on a file inside a protected directory would bypass any path-based protection. To counter this, we apply the same procedure to verify the creation of a hardlink that is performed during an inode rename operation.

Unauthorized use of benign applications - Malware's ability to utilize benign binaries in living-off-the-land and fileless malware attacks poses an interesting dilemma when designing a security system. When malware has unauthorized access to a system's

benign binaries, it can masquerade its behavior as non-malicious in nature. Many host many software applications and scripts utilize benign system utilities to perform their task. Malware's ability to leverage this access needs to be addressed by any host security system.

Our threat model analysis reduces the malware threat to these six unauthorized actions. This is advantageous moving forward with our MAC system design, as each action can be isolated and validated independently. By identifying and limiting the scope of the problem, we can also provide an intuitive policy design scheme, which enables users to quickly identify the critical directory paths of their system and limit the applications that have access to these critical data paths.

System Design

This chapter describes the system design of *Path-Safe* , a MAC security system that leverages encrypted security tokens to establish a sessions-based MAC enforcement model between the kernel and userspace processes. Our system monitors kernel resources by registering callback functions specific kernel object hooks provided by the Linux Security Module (LSM) framework, as seen in Figure 4.1. Utilizing the LSM framework enables us to not be dependent on the current system call interface. A key advantage of using the LSM framework is that *Path-Safe* is immune to new attack vectors introduced by changes in the system call interface with respect to filesystem access. The LSM framework also allows *Path-Safe* to focus only on the file actions that prevent unauthorized access to the filesystem enabling an efficient and thorough evaluation of all filesystem operations on the host.

Configuration for our security system is managed by creating policies. Policies allow *Path-Safe* to identify what directory paths require authentication and which binaries should have access to the specified directory paths. Utilizing a path-based policy construct allows us to develop highly interpretable configuration primitive for our system. Each policy contains a belf_dir_info structure that defines the directory path information of a protected directory and a linked list of ACLs containing approved binaries stored in a belf_dir_acl structure. The belf_dir_info structure consists of a character string representing the protected directory path, an integer hash value of that string, and a UUID. Using a hash value

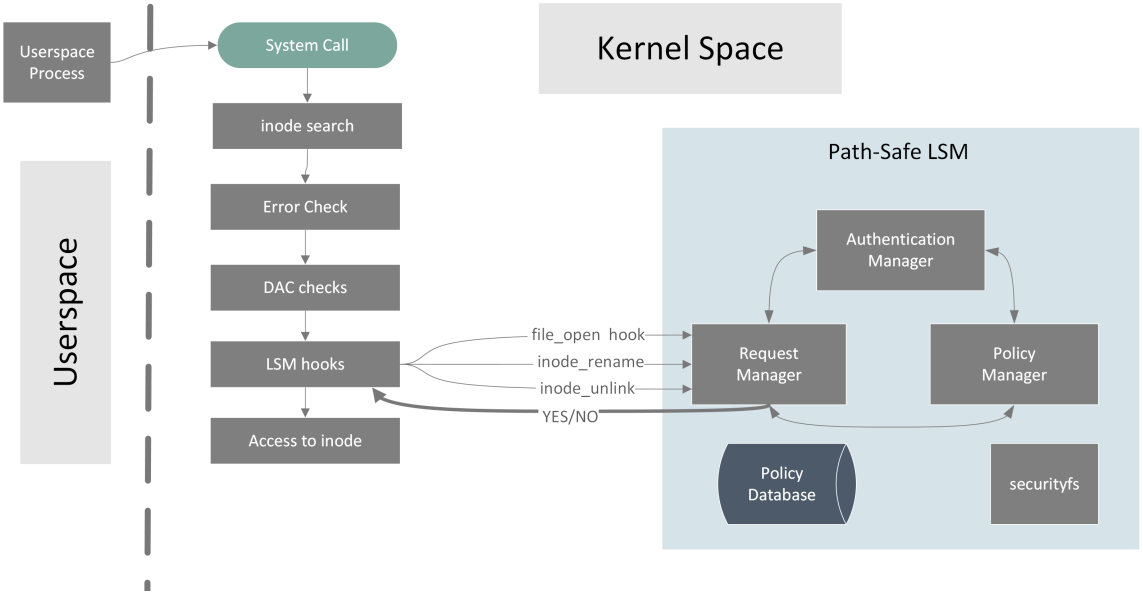

Figure 4.1: This figure shows the *Path-Safe* system architecture.

comparison is more efficient than making string comparisons, especially when searching through policies with common directory bases. Our system uses this UUID as a freshness label on the policy, providing a means to know if a token it is looking at has the latest version of the policies. Each policy also has an ACL list of binaries that can access the directory. This structure is a linked list with a binary_path name and an integer hash of the string. The remaining sections in this chapter look at the design decision made in order to address the unauthorized malware actions identified in Section 3.4.

4.1 Unauthorized Binary Execution

We see from our threat analysis that malware can be executed both directly by a victim and by vulnerabilities in benign applications. Our threat model also exposed scenarios where processes may masquerade as different processes after being executed in order to evade detection from name-based mitigation strategies. To prevent malware, or any binary, from inheriting all access and permissions of its parent process, we need to ensure that our system can monitor binary before execution control is given over by the kernel. The LSM

framework provides a creds_for_exec hook which is called directly before the transition of the process to load the requested binary program. This is the last opportunity to set the security credentials for a newly executed binary and is a good choice for *Path-Safe* to evaluate the protected paths that the process has access to.

We know that another attack vector is to use benign applications to perform malware and that external sensor input, for example, provided by an IDS, would inform the security system that a new threat has been discovered or a modification in the policy database occurred. *Path-Safe* addresses these by the use of a process session id. A session provides the security system to address new threats at a per-process granularity. For example, an IDS might detect a process that is being used by malware in one instance but benign in another. The notion of a session provides the ability to address this by preventing access from the offending process. Our security system is not constrained to a policy-only model and needs to ensure it can adapt to per-process events.

The ability to use manage sessions and modify permissions on the fly led us to use a token-based approach, where the underlying policy and security parameters could change independent of format when the process started. This flexibility allows the seamless integration of trusted security systems without the need to restart *Path-Safe* . We can even operate in an environment where the security manager is located remotely. To handle this, we use encryption to provide a layer of privacy, preventing other processes from learning what specific accesses a process has access to. *Path-Safe* uses AES-128 encryption algorithm in CBC mode by default. A boot session key is created during system initialization, and a random initialization vector (IV) is used to create the token that is installed as a security object in the process' credentials pointer. We use the random IV as the session ID (SID) for the process. The session key and security tokens generated by *Path-Safe* are not exposed to userspace and are the foundation for enabling remote security managers to work at a process level granularity. Our encryption process also provides protection from replay attacks; should a process discover a way to influence its security credential, it could not

simply use an old or otherwise discovered token.

4.2 Unauthorized File Open Access

Our threat model also establishes that malware needs to have the ability to open file descriptors to perform any file management operations on a file object. From the security systems perspective, we leverage encrypted security tokens to establish which processes are accessed to access which resources, and we do not make this decision based directly on the process name associated with the calling process. This prevents malicious processes from spoofing the process name in order to elevate its access level by modifying controllable process metadata elements from userspace. When any process makes a file open request, *Path-Safe* must adjudicate the request and does so by monitoring the LSM hook file_open. In many systems, a large portion of the file open requests will be made by benign applications operating in directory paths that are not protected by the security system. In order to optimize performance, the decryption of a process' security token is only performed if the file open requests are covered by one of the *Path-Safe* policies. In the case of file access, our system utilizes the Linux virtual filesystem (VFS) dentry objects to allow us to translate LSM hook data structures such as *struct file* into searchable objects in our database.

4.3 Unauthorized Renaming an Inode Hardlink

We have also established that malware may want to rename the inode hardlink associated with a file. This may include renaming itself to an authorized binary as a countermeasure to avoid security monitoring systems. As we mentioned in the previous chapter, *moving* a file is an inode_rename hook provided by the LSM framework. Unlike the previous example, renaming a hardlink provides two *struct inode* pointers, one representing the original name and the other representing the new name to be modified. To ensure that an unauthorized

binary does not attempt to rename a file into or from a protected directory into an area the calling process can operate, we must look up both inode pointers. Similarly to the file open monitoring in the previous section, we only decrypt the calling process' security token when a policy match is hit.

4.4 Unauthorized Unlinking of an Inode Hardlink

Path-Safe prevents unauthorized binaries from having the capability to delete files in a protected directory. This is possible because the security module monitors every inode unlink request made by a running process via the LSM framework hook inode_unlink. Similarly to renaming a file, deleting a file is also an inode operation where the hardlink associated with the inode is removed. When the last hardlink associated with the inode is removed, the inode is released back to the filesystem and is available for reuse. When a calling process attempts to unlink an inode, the LSM hook provides a *struct inode* pointer. This pointer allows *Path-Safe* to look for a policy match before decrypting the process security token.

4.5 Unauthorized Linking of an Inode Hardlink

It is possible that advanced malware may attempt to circumvent path-based protections by creating a *shadow* hardlink to a file. Every file must have one or more hardlinks, which are used to associate a file with a name. Unlike symlinks, every hardlink of a file points to the same inode and is not simply a reference to the original name. This distinction is important because allowing a new hardlink to be created on a file inside a protected directory would bypass any path-based protection. To counter this, we apply the same procedure to verify the creation of a hardlink that is performed during an inode unlink.

4.6 Unauthorized Use of Benign Applications

The ability of malware to utilize benign processes provides an interesting challenge when developing a security system. There are use cases where benign binaries, such as *rm* or *shred*, may be appropriate to call from a binary. Namely, anytime the system has a legitimate reason to remove a file. Even in protected directories, authorized binaries should be allowed to be removed files only when appropriate. Our approach to this problem is to leverage the flexibility of our security token design and utilize external sensor input to enable either a policy bypass or make a policy more restrictive. The external sensor can be a physical sensor, such as a monitored server door relay or other tamper sensors. Another example would be a connected IDS system that detected a malware attack. In this situation, the IDS could inform a security system that it may need to adapt to the new threat and tighten up its security posture.

For this thesis, we demonstrate this capability by utilizing the security_fs pseudo-filesystem. *Path-Safe* exports a character device to userspace via the security_fs pseudo-filesystem named tripwire. This interface allows the host the ability to set a global status for the protected system. When the tripwire is set to active, our system will override all access to protected directories and prevent any further access to the protected directories on the host. The tripwire interface allows both read and write capabilities from userspace and is used to demonstrate adaptive threat capabilities. If, for any reason, the tripwire is activated, then all security validations for protected directories will fail.

Implementation

This chapter looks at some of the implementation details associated with the specific LSM hooks described above when *Path-Safe* is operational. Each section will look at a code excerpt of the LSM hook callback routine registered by the *Path-Safe* security module. This is then followed by a discussion on the policy look-up and security validation algorithms of our system.

5.1 creds_for_exec

```
1  struct belf_token* belf_req_auth_token(struct linux_binprm *bprm){
2      struct belf_dir_policy *test_policy;
3      char rbuf[NAME_MAX];
4      int ret = 0;
5      struct belf_token_data* tdata = NULL;
6      struct belf_token *token = NULL;
7
8      tdata = (struct belf_token_data*) kmalloc(sizeof(struct
       belf_token_data),GFP_KERNEL);
9      if(!tdata){
10         pr_err("BELF:belf_req_auth_token: ENOMEM");
11         return NULL;
```

```
12      }
13      tdata->tpid = current->pid;
14      tdata->policy_count = 0;
15      memset(tdata->dir_policies, 0, sizeof(tdata->dir_policies));
16      list_for_each_entry(test_policy,
17          &dir_policies, policy_list){
18              ret = belf_search_policy_for_acl(test_policy, bprm->filename
    );
19              if(!ret) {
20                  memcpy( &tdata->dir_policies[tdata->policy_count++],
21                          &test_policy->info, sizeof(struct belf_dir_info)
    );
22              }
23          }
24      token = belf_gen_token(tdata, sizeof(struct belf_token_data));
25      kfree(tdata);
26      return token;
27  }
```

Listing 5.1: The *Path-Safe* system compiles a list of policies that determine which protected directories a newly executed process will have access to. The size of each token is constant to prevent onlookers from guessing how many policies a process may access.

Path-Safe evaluates every process when it is initialized using the creds_for_exec hook. When the creds_for_exec hook is executed, our system creates a belf_token_data data structure using the belf_req_auth_token function, seen in Listing 5.1, which contains: the process id, the number of policies the process is authorized to access, and an array containing a copy of the specific policies. The process id provides our system

the ability to verify that the token belongs to the current process and prevents a process from using another process' token. The policy count and dir_policy array act as a lookup table. During token creation, *Path-Safe* searches the policy database ACL lists, looking for a name that matches the file name that the current process is trying to execute, and adds a copy of each matching policy to the token's dir_policies array. The array is set at a fixed size so that the encrypted token is also at a fixed size. The consistent size prevents side-channel attacks where an adversary attempts to identify which system binaries have access to protected areas of the host. An adversary could use this information, for example, to rank the worthiness of a system binary to exploit.

5.2 file_open hook

```
1
2  static int belf_file_open(struct file *file){
3      struct task_belf *bsp = belf_cred(current_cred());
4      struct belf_dir_info* dinfo = NULL;
5      char rbuf[PATH_MAX];
6      char* req_path_name = dentry_path_raw(file->f_path.dentry, rbuf,
       PATH_MAX);
7      dinfo = belf_search_policies_for_resource(req_path_name);
8      if( dinfo ){
9          if(!bsp->bp_token){
10             return 1;
11         }
12     return belf_validate_token(bsp->bp_token, dinfo);
13     }
14     return 0;
```

```
15 }
```

Listing 5.2: *Path-Safe* registers the belf_file_open with the LSM framework to monitor every file open request made on the host system.

Path-Safe monitors every file open operation by registering the function belf_file_open, shown in Listing 5.2, with the LSM framework to prevent unauthenticated processes from opening files inside protected directories. The LSM file_open hook is triggered every time a process requests a file descriptor for opening a file and provides *Path-Safe* with a *struct file* pointer to the file object being requested. This allows us to get the raw path name from the file dentry and examine the permission flags the process is requesting for the file.

Our system searches through the policy database looking for a policy match where the requested file object is protected by the policy and the requested permissions are allowed. If this criterion is met, the process authentication token will then be decrypted and evaluated for access. By searching the policy database first, benign requests will not incur the decryption and encryption overhead associated with token decryption operations.

5.3 inode_rename hook

```
1 int belf_inode_rename(struct inode *old_dir,
2     struct dentry *old_dentry,
3     struct inode *new_dir,
4     struct dentry *new_dentry){
5     int ret = 0;
6     /* Get path for old_dentry and check if we are authorized to move it
      */
7     if(belf_validate_dentry(old_dentry)){
8         return 1;
```

```
9       }
10      /* Get path for new_dentry and check if we are authorized to move it
        */
11      ret = belf_validate_dentry(new_dentry);
12      return ret;
13  }
```

Listing 5.3: *Path-Safe* registers the belf_inode_rename function with the LSM framework to monitor every inode rename request made on the host system. An inode rename requires validating two dentry objects to ensure the new or old names are not covered by a policy.

Moving or renaming a file is an inode operation, where the name of an inode hardlink is modified. *Path-Safe* monitors every inode rename attempt by registering the function belf_inode_rename, shown in Listing 5.3, with the LSM framework to prevent unauthenticated renaming of files inside protected directories. The LSM inode_rename hook is triggered every time a process attempts a renaming operation and provides *Path-Safe* with two *struct dentry* pointers. The inode_rename hook operates on a set of inode and dentry objects representing the original name and new name being requested. The inode_rename hook must check both dentry objects independently to verify if the source or destination names are inside a protected directory. The Security Manager validates the calling authorization token for the calling process for each dentry that needs authorized access.

5.4 inode_link

```
1  int belf_inode_link (struct dentry *old_dentry,
2      struct inode *dir,
3      struct dentry *new_dentry){
4      int ret = 0;
```

```
5     /* Get path for old_dentry and check if we are authorized to move it
      */
6     if(belf_validate_dentry(old_dentry)){
7         return 1;
8     }
9     /* Get path for new_dentry and check if we are authorized to move it
      */
10    ret = belf_validate_dentry(new_dentry);
11    return ret;
12 }
```

Listing 5.4: *Path-Safe* registers the belf_inode_link function with the LSM framework to monitor inode link requests made on the host system. Adding a hardlink to an inode also requires validations. We need to check the original hardlink and new hardlink paths to verify if they need authorization from *Path-Safe* .

Path-Safe monitors every inode link attempt by registering the function belf_inode_link, shown in Listing 5.4, with the LSM framework to prevent creating unauthorized hardlinks for files inside protected directories. The LSM inode_link hook provides *Path-Safe* with two *struct dentry* pointers. The inode_rename hook operates on a set of inode and dentry objects representing the original name and new name being added to the inode. The inode_link hook must check both dentry objects independently to verify if the source or destination names are inside a protected directory. The security module validates the calling authorization token for the calling process for each dentry that needs authorized access.

5.5 inode_unlink

```
int belf_inode_unlink(struct inode *dir, struct dentry *dentry){
    int ret = 0;
    ret = belf_validate_dentry(dentry);
    return ret;
}
```

Listing 5.5: *Path-Safe* registers the belf_inode_unlink function with the LSM framework to monitor every inode unlink request made on the host system.

Path-Safe monitors every inode unlink attempt by registering the function belf_inode_unlink, shown in Listing 5.5, with the LSM framework to prevent unauthenticated unlinking of files inside protected directories. The LSM inode_unlink hook is triggered every time a process attempts an unlink operation and provides *Path-Safe* with a *struct dentry* pointer to the file the calling process is requesting to be unlinked from the filesystem. The inode_unlink hook searches the policy database for a matching entry and validates the process' authorization to operate on the dentry object.

5.6 Validating a *dentry* Object

```
static int belf_validate_dentry(struct dentry *d){

        char name[NAME_MAX];
        struct task_belf *bsp = belf_cred(current_cred());
        struct belf_dir_info* dinfo = NULL;

        /* Let's use the helper to extract the path name from a dentry*/
        char *p = dentry_path_raw(d, name, NAME_MAX);
        if (IS_ERR(p)) {
```

```
10                  pr_err("belf_name: Dentry Name error\n");
11                  return 1;
12          }
13
14          dinfo = belf_search_policies_for_resource(p);
15          if( dinfo ){
16                  if(!bsp->bp_token){
17                          pr_err("Belf validate dentry: No token available
    \n");
18                           return 1;
19                  } /*Just checking that we can fail an open */
20                  return belf_validate_token(bsp->bp_token, dinfo);
21          }
22
23          return 0;
24  }
```

Listing 5.6: All monitored filesystem operations are approved based on the success of validating access to the underlying dentry structure. This is handled by the function belf_validate_dentry.

The core implementation details of *Path-Safe* revolve around validating a process' access to operate on the dentry structure the file operations are to be performed on. *Path-Safe* registers hooks that provide the means to isolate the dentry objects that reference the file being operated on and is handled by the belf_validate_dentry function seen in Listing 5.6. This process can be further broken down into two distinct tasks, policy lookup and security token validation. The following two sections take a closer look at the implementation used for this work.

5.7 Policy Lookup

```
1  struct belf_dir_info* belf_search_policies_for_resource(char*
       request_path){
2      struct belf_dir_policy *test_policy;
3      int ret = 0;
4
5      list_for_each_entry(test_policy, &dir_policies, policy_list){
6          ret = belf_file_in_path(test_policy->info.dir_path, request_path
       );
7          if(!ret){
8              return &test_policy->info;
9          }
10     }
11     return NULL;
12 }
13 int belf_file_in_path(const char* policy_path, char* request_path){
14     int ret=0;
15     char pbuf[PATH_MAX];
16     char rbuf[PATH_MAX];
17     struct path pp_path;
18     char* req_path_name;
19     char* real_policy_path;
20     int policy_dir_len, rp_len;
21
22     /* Check if the path exists and/or is mounted */
23     /* We also want to avoid symlink tricks so we find the real path */
```

```
24    /* If this changes outside we can't help it but at lease we can
      protect
25        what the link points to */
26    ret = kern_path(policy_path, LOOKUP_FOLLOW|LOOKUP_DIRECTORY, &
      pp_path);
27    if(ret){
28        //pr_err("BELF ERR: %s doesn't exists of isn't a directory",
      policy_path);
29        return 1;
30    }
31    /* Let's get the path strings */
32    real_policy_path = d_path(&pp_path, pbuf, PATH_MAX);
33    req_path_name = request_path;
34
35    if(!req_path_name) return 1;
36    /* We need to know if the policy path is a root directory of the
      requested path */
37    policy_dir_len = strlen(real_policy_path);
38    rp_len = strlen(req_path_name);
39
40    /* There is no need to do a memcmp as we can't be a root directory
      */
41    if(rp_len < policy_dir_len) return 1;
42
43    /* Check that we have a '/' in the path */
44    if((rp_len > policy_dir_len) && (req_path_name[policy_dir_len] != '/
      ') ){
```

```
45          return 1;
46      }
47      return memcmp(req_path_name, real_policy_path, strlen(
        real_policy_path));
48  }
```

Listing 5.7: *Path-Safe* policy look up is handled by iterating through a list of policies, checking if the current dentry is covered the polices.

The first task for dentry validation is identifying the specific policies that the dentry object is covered by (i.e., requiring a token validation.) Looking at the implementation seen in Listing 5.7, we see that belf_search_policies_for_resource takes a raw path string provided by the belf_validate_dentry function and iterates through the policy database. This raw path is extracted from the dentry object and compared to the policy-specific path. The kern_path kernel function is used to resolve any files attempting to access a file via a symlink, mitigating any process attempting to gain access to a protected directory via a symlink, given the focus on intuitive policy definitions the number of policies managed by *Path-Safe* is generally much smaller than that of other MAC system. This allows us to use the Linux kernel's implementation of a linked list for this work. Finally, for each policy in our list, we evaluate if the current dentry path is a child of the policy's protected path. This determination is made using the following checks: i) is the current policy path mounted, ii) is the policy path the root filesystem, iii) and if the dentry path is a child of the policy path, as seen in the belf_file_in_path function listed in Listing 5.7.

5.8 Validating security tokens

```
1  int belf_validate_token(struct belf_token* token, void* data){
2      struct belf_token *local_token = NULL;
```

```
3     struct belf_token_data *tdata = NULL;
4     struct belf_dir_info* dinfo = data;
5     int i=0;
6     int rc = 1;
7
8     /* Let's decrypt our token for validation checks */
9     local_token = belf_decrypt_token(token);
10    if(!local_token){
11        pr_err("Belf:belf_validate_token: ENOMEM");
12        return -ENOMEM;
13    }
14
15    tdata = (struct belf_token_data*)local_token->data;
16    /* Let's check that we have the right token */
17    if(current->pid != tdata->tpid){
18        pr_debug("Token is not for this process\n");
19        goto out;
20    }
21
22    /* Let's check if the tripwire is hit */
23    if( get_tripwire_status()){
24            goto out;
25    }
26
27    for( i; i < tdata->policy_count && i < MAXPOLICIES; i++){
28        if(dinfo->hash == tdata->dir_policies[i].hash){
29            rc = 0;
```

```
30              goto out;
31          }
32      }
33
34  out:
35          kfree(local_token);
36          return rc;
37  }
```

Listing 5.8: To validate a security token, *Path-Safe* decrypts the token in place and searches the authorized policy list for a match. It also checks the tripwire interface to ensure that policy-based access is still permitted.

Path-Safe validates access to resources based for a process based on the process' authentication token. Token validation happens after the request manager has identified the specific policy for which the process needs to be authenticated. Looking at Listing 5.8, the first step in token validation is to decrypt the current process security token and verify that the process id of the token matches the calling process. This check is to ensure that the process is not reusing an old token or that the token hasn't been tampered with. The token contains a dir_policy array structure that contains a list of policies that the process has been authorized to access. Rather than making a more costly string comparison, our algorithm compares the calculated hash value of the protected directories as the key to match. The hash is calculated at policy creation, reducing the overhead associated with the *Path-Safe* system. Finally, additional security inputs to the system are assessed. For this work, the tripwire interface acts as our external input.

Evaluation

In this section, we first demonstrate the accuracy of the *Path-Safe* against our threat model. Second, we show the performance overhead analysis and the effectiveness of utilizing security tokens to manage file system access. Finally, we show the step-by-step analysis of *Path-Safe* protecting a host against three real-world malware samples.

6.1 Accuracy Analysis

To evaluate the accuracy of *Path-Safe* at protecting directories from unauthorized access to our filesystem, we use benign applications to generate filesystem events that we can use to measure the following metrics for each of the *Path-Safe* LSM file and inode hooks:

True Positive (TP) Authorized Application is granted access to a protected directory

False Positive (FP) Unauthorized Application is granted access to a Protected Directory

Table 6.1: Accuracy

Hook	Attempts	TP	FP	TN	FN	Accuracy
file_open	600	300	0	300	0	1
inode_rename	300	150	0	150	0	1
inode_link	300	150	0	150	0	1
inode_unlink	300	150	0	150	0	1
Active Tripwire	300	0	0	300	0	1

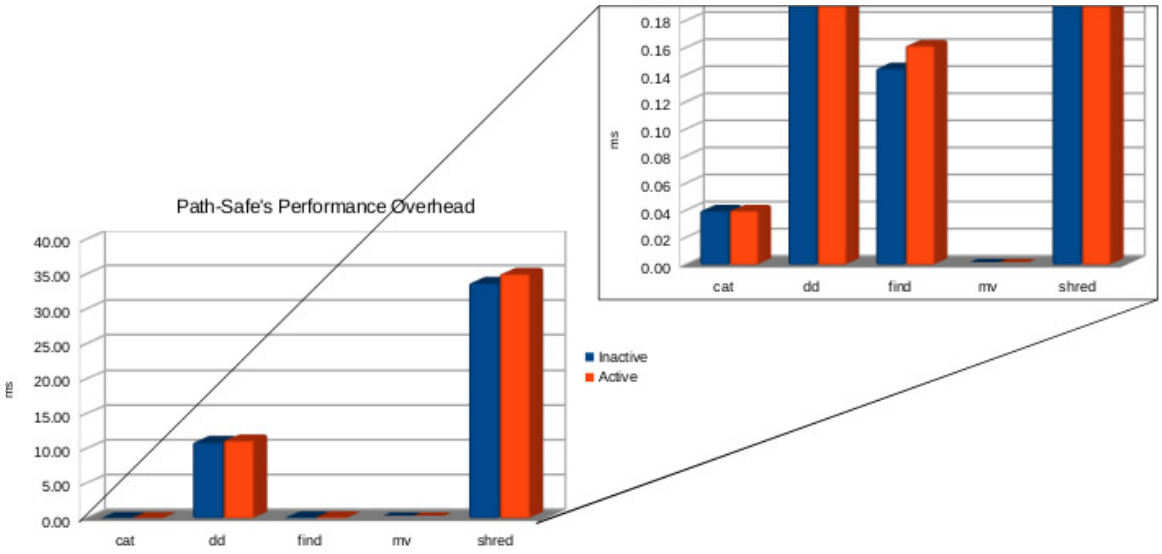

Figure 6.1: We did not observe any significant increase in performance overhead when *Path-Safe* was operational. This is in part due to our threat reduction analysis.

True Negative (TN) Unauthorized Application is denied access to Protected Directory

False Negative (FN) Authorized Application is denied access to Protected Directory

We calculate the accuracy of each LSM hook using the as follows formula:

$$\frac{TP + TN}{TP + TN + FP + FN} \tag{6.1}$$

We run a data generation script to stimulate all of the LSM hooks that *Path-Safe* monitors, generating a total of 1800 access requests to our protected directories. Half of the access attempts were run on symlinks that referenced the protected directories, and 300 attempts were executed with the tripwire interface active. To ensure we had full code coverage, we ran benign applications against both protected and unprotected directories. We can see from Table 6.1 that we achieved 100% accuracy for each attempted access.

6.2 Overhead Analysis

To evaluate the overhead associated with *Path-Safe* , we compare the running time of some benign applications with and without *Path-Safe* running. We also instrumented the *Path-Safe* code to measure the time it takes to search the policy database for kernel object requests and the time to validate a security token vended by our system. What we can see from Figure 6.1 is that applications such as dd or shred, which spend most of their processing time reading or writing to files, had a modest 2-4% increase in system clock time when operating on file sizes of 409.6 Mb. The system time is the time a process spends in kernel mode and is where our system would potentially introduce some processing overhead. Other applications, such as the *find* and *ls*, have different behavior. They spend a significant amount of their system time opening and closing files, which happens to represent the worst-case scenario for *Path-Safe* . We tested the *find* command on the root filesystem of our virtual disk, which contains 6179 directories and 51288 files, and it only experienced a nominal system time increase of around 10-12%. For long-running applications that spend a proportionally small amount of time opening, moving, or deleting files, the system overhead would be negligible and only realized at the applications initialization.

6.3 Performance against Real-world Malware

For this experiment we tested *Path-Safe* against the following three recent malware variants: Awfulshred Wipper, RansomExx, and XorDDos. Given the dangerous nature of the live malware samples, we utilize separate Qemu virtual environments for each malware sample, isolating the samples from our host operating system and network. This also ensures that each malware sample is tested in a pristine environment not damaged by previous malware testing.

6.3.1 Awfulshred Wipper

```
# ./cfca38c408c95e45cdf797723dc5cdb0d6dadb1b8338a5fda6808ce9a04e6486
[   70.922364] cfca38c408c95e4[204]: segfault at 0 ip 0000000008074ee8
   sp 00000000ffc5357c error 4 in
   cfca38c408c95e45cdf797723dc5cdb0d6dadb1b8338a5fda6808ce9a04e6486
   [8048000+3a000]
[   70.930461] Code: 00 00 83 c4 10 89 d8 5b 5e 5f c3 89 1c 24 e8 bf 00
   00 00 83 c4 10 01 c3 89 d8 5b 5e 5f c3 90 90 90 90 8b 44 24 04 8b 54
    24 08 <0f> b6 08 3a 0a 74 16 eb 18 8d b4 26 00 01
Segmentation fault
# ls /root/protected-data/ps/cat1.out -l
-rw-------    1 root     root            41 Jun  8 23:49 /root/protected
   -data/ps/cat1.out
# rm /root/protected-data/ps/cat1.out
# ls /root/protected-data/ps/cat1.out -l
-rw-------    1 root     root            41 Jun  8 23:49 /root/protected
   -data/ps/cat1.out
```

Listing 6.1: The awfulshred malware fails to connect to its remote server and cannot achieve creating its C2 channel when *Path-Safe* is operational. We have protected the protected-data directory to prevent access to commands other than *ls* to access it.

The Awfulshred malware is a wiper Bash script used by Linux variants of Industroyer2 to destroy the entire contents of the system it is attacking using the shred or dd command[15]. It also tries to disable the HTTP and SSH services of the host being attacked to expedite the speed at which the host is rendered inoperable[15]. We downloaded a live sample of the Awfulshred that was submitted to the site https://virusshare.com/ on 2022-05-05 22:48:46 UTC with the following Sha256 hash bcdf0bd8142a4828c6

1e775686c9892d89893ed0f5093bdc70bde3e48d04ab99.

Running the Awfulshred malware is a straightforward malware sample to run. The obfuscated Bash script needed either the command *shred* or *dd* to be installed in the guest operating system used to test the Awfulshred. During the execution of the script, the Awfulshred sample attempts to establish communication with a remote server and attempts to detect the environment it was running in. When the script fails to communicate with the remote server, it attempts to delete itself to avoid detection from the system. Fortunately, after some analysis of the script, we were able to isolate the use of the *shred* or *dd* commands and run them as intended by the malware.

The malware attempts to remove files from the following directories: /boot, /home. /var/log before attempting to destroy the drives via their devfs special files. To counter this attack, we installed policies that protected each directory and only allowed ls, *find*, and *md5sum* to make sure that we could evaluate our files after executing the malware. Fortunately, as soon as the Awfulshed failed to access any of the directories inside these directories, it experienced a segmentation fault. We can see the results of this experiment in Listing 6.1. Our system successfully prevented Awfulshred or benign applications (e.g. *shred* and *rm*) from removing any of the files in our protected directory.

6.3.2 RansomExx

```
# cat /root/protected-data/ps/cat1.out
real  0m 0.04s
user  0m 0.00s
sys 0m 0.04s
# ./196eb5bfd52d4a538d4d0a801808298faadec1fc9aeb07c231add0161b416807.elf
    /root/protected-data/ps
# cat /root/protected-data/ps/cat1.out
```

```
real  0m 0.04s
user  0m 0.00s
sys 0m 0.04s
#
```

Listing 6.2: When our security system is running, we see that the RansomExx malware cannot operate in the protected directory.

We downloaded a live sample of the RansomExx that was submitted to the site https://bazaar.abuse.ch on 2021-03-21 01:45 UTC with the following Sha256 hash 196eb5bfd52d4a538d4d0a801808298faadec1fc9aeb07c231add0161b416807. The RansomExx Elf binaries we downloaded were the only malware binary that was not statically compiled. This required a slight modification and addition of some to satisfy the dynamic loader to get the malware to execute. Namely, it required libpthread support. The ransomware sample expects to receive a list of directory paths to encrypt as input, as seen in [20]. Upon execution, the ransomware iterates through the specified directories, attempting to encrypt all files in the directory that is great than 40 bytes[20]. When run against directories protected with *Path-Safe* , the binary did not encrypt anything and exited silently. When we ran the binary against an unprotected directory, we saw attempted file opens, then a quick segfault.

6.3.3 XorDDoS

```
# pwd
/root/malware-samples
# ls -ltra
total 3308
....
```

```
-rw-------    1 root     root        264040 Jun  8 23:48 311
   c93575efd4eeeb9c6674d0ab8de263b72a8fb060d04450daccc78ec095151.zip
-rwx--x--x    1 root     root        562263 Jun  9 00:41 311
   c93575efd4eeeb9c6674d0ab8de263b72a8fb060d04450daccc78ec095151.elf
.....
drwx------    2 root     root          4096 Jul 14 02:24 .
# ./311c93575efd4eeeb9c6674d0ab8de263b72a8fb060d04450daccc78ec095151.elf
# ls -ltra
total 3308
....
-rw-------    1 root     root        264040 Jun  8 23:48 311
   c93575efd4eeeb9c6674d0ab8de263b72a8fb060d04450daccc78ec095151.zip
.....
-rwxr-xr-x    1 root     root           256 Jul 14 02:24 fle.151590
   ce87cccad05440d060bf8a27b362ed8ba0d4766c9beee4dfe57539c113.sh
-rwxr-xr-x    1 root     root        562263 Jul 14 02:24 fle.151590
   ce87cccad05440d060bf8a27b362ed8ba0d4766c9beee4dfe57539c113
drwx------    2 root     root          4096 Jul 14 02:24 .
# cd /root/protected-data/
# ls
nops  ps
# /root/malware-samples/311
   c93575efd4eeeb9c6674d0ab8de263b72a8fb060d04450daccc78ec095151.elf
# ls
nops  ps
#
```

Listing 6.3: The malware XorDDoS extracts a helper script and an executable during

execution. When done in a protected directory, the malware doesn't execute.

We downloaded a live sample of XorDDos that was submitted to the site https://bazaar.abuse.ch on 2023-05-14 18:43:39 UTC with the following Sha256 hash 311c93575efd4eeeb9c6674d0ab8de263b72a8fb060d04450daccc78ec095151. According to [16], the XorDDos trojan will attempt to find a writeable directory in the following list of directories: /bin, /home, /root, /tmp, /usr, and /etc. It then tries to use the curl command to download an ELF file payload from a specified target as the file ygljglkjgfg0. Next, XorDDos attempts to rename the system's *wget* binary to the file named good[16].

After executing our malware sample, we noticed the sample trying to manipulate the iptables rules on our host, followed by a series of tests to find a writable file in the following directories. When it reaches the /root directory, we see the *Path-Safe* denies access to the binary. The binary continues by attempting to create a file in the /tmp directory using the curl command. Our testing environment does not have network connectivity, and causes the malware to attempt to remove itself. When run in an unprotected directory /root/malware-samples, we see that when the binary fails to establish a C2 connection, it extracts its own packed binary and helper script, as seen in Listing 6.3. To test *Path-Safe* against this malware, we ran the malware in both protected and unprotected directories. We can see that our system prevented XorDDos from extracting its payload when run in a protected directory.

Limitations and Potential Solutions

In this chapter, we discuss some of the limitations of the current system and some of the potential solutions to overcome them. Our main assumption in this thesis is that the attacker has not already gained access and control over the kernel. *Path-Safe* can be enhanced to support key management technologies such as SGX, TPM, SEG, and hardware key managers (HKM) to provide some protections against in-kernel threats attacks and out-of-band modification of *Path-Safe* policy database, for example, during an evil maid attack.

The current implementation of *Path-Safe* monitors the inode_link hook to prevent the unauthorized creation of hard links for protected directories. We assume preexisting hard links associated with a file inode do not create backdoor access to a protected directory. This is easily overcome by having *Path-Safe* iterate through every inode hardlink during a resource request. This may provide some extra overhead when searching the policy database for policies that cover each hardlink, but in practice, multiple hardlinks are not common. Still, this might be addressed in future work.

We currently are not using the session tracking to revoke previously open file descriptors on policy changes. Our work focuses on preventing unauthorized access, as laid out in Section 3. We can easily expand our system and keep an active list of open file descriptors covered by a *Path-Safe* policy. This would require developing a file descriptor tracking capability for each session. To minimize memory resources, *Path-Safe* will only track the file descriptors associated with a process's session ID and file descriptors covered by a security policy. This adaption would enable *Path-Safe* to close any open file descriptors affected by

a modification in the policy database or threat environment.

For this work, file integrity management is assumed to be provided by other security safeguards, such as Linux's Integrity management architecture(IMA). One of the strengths of leveraging the LSM architecture is that it facilitates a layered security approach where *Path-Safe* can be one of many security systems present in the system. Integrity management is currently outside the scope of this work; however, support for program argument lists would enable an extra layer of protection, especially during a living-of-the-land attack.

Conclusion

This chapter concludes this thesis. In the first section, we discuss some of the future work where we can expand the *Path-Safe* MAC system. We then summarize our work developing a token-based MAC.

8.1 Future Work

There are a few areas we a looking to expand the *Path-Safe* systems capabilities beyond adding more advanced key management support. *Path-Safe* is already effective in embedded and industrial control systems, where special files in devfs and sysfs can expose access to hardware devices and protocol busses (e.g., i2c, smb, and RS-232.) The next step is to incorporate hardware-based triggers to augment the tripwire functionality. This would enable physical access control integration to hosts protected by *Path-Safe* . We can utilize the encrypted security token to development of remote distributed Security managers that can be incorporated into a larger enterprise security model is being developed. *Path-Safe* can be enhanced to support more advanced hardware features such as SGX, TPM, and SEG. They would facilitate building a more robust capability to prevent in-kernel attacks and the out-of-band modification of Path-Safe policies.

One unique feature of utilizing a session-based approach to managing processes is that it can reevaluate access in real time when a policy changes while a process continues to run. One area of expansion would be to develop file descriptor tracking to enable *Path-Safe* the

ability to revoke file descriptor affected by a policy change by its session id. Developing a session ID cache for each policy would allow us to close any already open file descriptors that are out of date. A session ID cache would also allow us to cache access credentials to limit the number of token decryptions required, reducing some of the overhead associated with our system. This would be particularly advantageous for applications such as *find*, which open and close a lot of files.

The current policy schema is designed to be highly interpretive by a user. There is more work to be done to expand the policy description capabilities. One advancement would be the ability to group binaries into groups. This abstraction would simplify policy creation for applications that need to access many applications. The motivation for this advancement is to begin the development of a highly secure package management and software deployment system.

8.2 Summary

To summarize, in this thesis, we have demonstrated that the use of encrypted tokens provides the flexibility to develop next-generation MAC security systems that can adapt to the current threat environment in real time and with a per-process granularity. The ability to modify access on a process basis allows us to develop security systems that are immune to living-off-the-land attacks and potentially bridge physical security sensors with the security system to mitigate some insider threat attacks. This would be particularly advantageous in industrial control settings, where long-running processes can perform critical operations and continuous monitoring without interruption. By modeling and analyzing the malware threat, we reduced the threat space to a handful of kernel object actions. This enabled us to focus monitoring of kernel object access to a handful of actions, avoiding the overhead associated with other MAC security systems, which require every kernel object and action to be labeled. Using kernel object actions exposed by the LSM framework facilities

seamless support for OS upgrades and system call interface changes. This work has shown that session-based MAC systems enable stateful tracking of kernel object access from a process, allowing new techniques for protecting kernel objects and moving us beyond an immutable yes or no access model. Tokenized management of security policies is extensible and can be extended beyond only protecting filesystem access-based threats. Our methodology for designing MAC has the potential to develop exceptional state-of-the-art security protections and resource management systems.

Bibliography

[1] Jinwoo Ahn, Donggyu Park, Chang-Gyu Lee, Donghyun Min, Junghee Lee, Sungyong Park, Qian Chen, and Youngjae Kim. KEY-SSD: Access-control drive to protect files from ransomware attacks. Technical report, Sogang University, Seoul, Republic of Korea, University of Texas at San Antonio, TX USA, Korea University, Seoul, Republic of Korea, 2019

[2] Harun Oz, Ahmet Aris, Albert Levi, and A. Selcuk Uluagac. A survey on ransomware: Evolution, taxonomy, and defense solutions. ACM Comput. Surv. 1, 2021.

[3] Suhyeon Lee, Huy Kang Kim, and Kyounggon Kim. Ransomware protection using the moving target defense perspective. Computers & Electrical Engineering, 78:288–299, 2019.

[4] Ziya Alper Genc, Gabriele Lenzini, and Peter Y. A. Ryan. No random, no ransom: A key to stop cryptographic ransomware. In Cristiano Giuffrida, Sebastien Bardin, and Gregory Blanc, editors, Detection of Intrusions and Malware, and Vulnerability Assessment, volume 10885, pages 234–255. Springer International Publishing, 2018. Series Title: Lecture Notes in Computer Science

[5] Sailik Sengupta, Ankur Chowdhary, Abdulhakim Sabur, Adel Alshamrani, Dijiang Huang, and Subbarao Kambhampati. A survey of moving target defenses for network security. Number: arXiv:1905.00964

[6] Dorka Palotay. Ransomware as a service. Technical report, Sophos, 2017

[7] Abdulrahman Alzahrani, Ali Alshehri, Hani Alshahrani, Raed Alharthi, Huirong Fu, Anyi Liu, and Ye Zhu. RanDroid: Structural similarity approach for detecting ransomware applications in android platform. In 2018 IEEE International Conference on Electro/Information Technology (EIT), pages 0892–0897. IEEE, 2018

[8] Amin Kharraz, William Robertson, Davide Balzarotti, Leyla Bilge, and Engin Kirda. Cutting the gordian knot: A look under the hood of ransomware attacks. In Magnus Almgren, Vincenzo Gulisano, and Federico Maggi, editors, Detection of Intrusions and Malware, and Vulnerability Assessment, volume 9148, pages 3–24. Springer International Publishing, 2015. Series Title: Lecture Notes in Computer Science.

[9] Emanuele Cozzi, Mariano Graziano, Yanick Fratantonio, and Davide Balzarotti. Understanding linux malware. In 2018 IEEE Symposium on Security and Privacy (SP), pages 161–175, 2018. ISSN: 2375-1207.

[10] Jinghao Jia, YiFei Zhu, Dan Williams, Andrea Arcangeli, Claudio Canella, Hubertus Franke, Tobin Feldman-Fitzthum, Dimitrios Skarlatos, Daniel Gruss, Tianyin Xu. Programmable System Call Security with eBPF. arXiv/2302.10366. 2023.

[11] Serge Hallyn, Phil Kearns. Domain and Type Enforcement for Linux. In 2000 4th Annual Linux Showcase & Conference (ALS 2000), USENIX Association, 2000.

[12] ”What is SELinux?,” www.redhat.com. https://www.redhat.com/en/topics/linux/what-is-selinux (accessed May. 24, 2023).

[13] John Johansen,Steve Beattie. ”About” apparmor.net. https://gitlab.com/apparmor/apparmor/-/wikis/About (accessed May. 24, 2023).

[14] Daniel Kapellmann Zafra, Raymond Leong, Chris Sistrunk, Ken Proska, Corey Hildebrandt, Keith Lunden, Nathan Brubaker. ”INDUS-

TROYER.V2: Old Malware Learns New Tricks" www.mandiant.com. https://www.mandiant.com/resources/blog/industroyer-v2-old-malware-new-tricks (accessed June 10, 2023).

[15] "Industroyer2: Industroyer reloaded", www.welivesecurity.com. https://www.welivesecurity.com/2022/04/12/industroyer2-industroyer-reloaded (accessed June 10, 2023).

[16] "Rise in XorDdos: A deeper look at the stealthy DDoS malware targeting Linux devices", www.microsoft.com. https://www.microsoft.com/en-us/security/blog/2022/05/19/rise-in-xorddos-a-deeper-look-at-the-stealthy-ddos-malware-targeting-linux-devices/ (accessed June 10, 2023).

[17] Naveen. "How to Detect Raising New XORDDOS Linux Trojan", https://www.socinvestigation.com. https://www.socinvestigation.com/how-to-detect-raising-new-xorddos-linux-trojan/ (accessed June 10, 2023).

[18] Claudia Glover. "Ukraine DDoS attacks could mask more sophisticated cyber warfare" techmonitor.ai. https://techmonitor.ai/technology/cybersecurity/ukraine-ddos-attacks (accessed June 10,2023).

[19] "RansomEXX", www.trendmicro.com. https://www.trendmicro.com/vinfo/us/security/news/ransomv spotlight/ransomware-spotlight-ransomexx (accessed June 10, 2023).

[20] "RansomExx Upgrades to Rust", securityintelligence.com. https://securityintelligence.com/posts/ransomexx-upgrades-rust/ (accessed June 10, 2023).

[21] "Linux.Encoder.1", vms.drweb.com. https://vms.drweb.com/virus/?i=7704004 (accessed June 10, 2023).

[22] Aljanabi Mohammad,Ismail Mohd Arfian, Ali Ahmed. (2021). Intrusion Detection Systems, Issues, Challenges, and Needs. International Journal of Computational Intelligence Systems. 14. 10.2991/ijcis.d.210105.001.

[23] Kevin Savage, Peter Coogan, and Hon Lau. The evolution of ransomware. Technical report, Symantec, August 2015.