Sādhanā (2020)45:141
https://doi.org/10.1007/s12046-020-01381-7

PAM: process authentication mechanism for protecting system services against malicious code attacks

MUTHUMANICKAM KRISHNAN[1,*] and LAVARASAN EGAMBARAM[2]

[1]Department of Information Technology, Kongunadu College of Engineering and Technology, Tiruchirappalli, Tamilnadu 621 215, India
[2]Department of Computer Science and Engineering, Pondicherry Engineering College, Puducherry 605 014, India
e-mail: kmuthoo@yahoo.com; eilavarasan@pec.edu

MS received 29 December 2018; revised 12 April 2020; accepted 29 April 2020

Abstract. The kernel of the modern operating system fails to ensure the authenticity of a running process while servicing a system call. Verifying the origin and integrity of a system call is an important security issue in terms of ensuring the proper functioning of an end-system. The conventional process identification parameters such as process identifier, process names and the executable flow exercised by the operating system are not reliable. As a result, a stealthy malware may mimic other processes to carry out many computer crimes, thus compromising the end-system. In this paper, we present a novel idea in which system call invocations made by a malicious application are verified during runtime in Windows operating system. To ensure the authenticity of a process while servicing a system call, we propose a behavior-based mechanism, namely, the process authentication mechanism (PAM), for combating malicious code attacks that verifies the identity of each suspected process before being serviced by the kernel. The simulation and performance evaluation results confirm that our mechanism can effectively block all malicious samples that directly invoke system services in the kernel mode. PAM incurs no more than two percent overhead and helps to strengthen the overall system security.

Keywords. Malicious code attack; process authentication; security; SSDT hook; system call; Windows.

1. Introduction

Computer security has long been required to protect either an important end-system or an entire network of hosts. Today, the stealthy malicious software is one of the most dangerous and challenging security threats. In reality, it is difficult to design a system which is guaranteed to be secure or to remain secure over time. In the modern online world almost 92% [1] of personal computers operate with Microsoft Windows® Environment. This homogeneous property enables the remote attacker to hack into a computer easily. Such unauthorized visits also allow the remote attacker to inject malicious code into the software to gain super-user control of the victim computer. Attackers inject their code into the software and force it to execute the injected code abnormally. During the code injection phase, the attackers need to execute privileged events, by calling a system service, more importantly, a native Application Programming Interface (API)s. Without distinction, the kernel provides services to both the malicious processes of an application and legitimate processes. Though many vendors, import different security features into their product like as security functionality to protect a system completely, this may not be enough to trust the system. The typical Operating System (OS) kernel often fails to include stronger restrictions on the executable program. There are two useful common malware defense techniques which protect system resources against malicious code attacks. They are system call monitoring (SCM) technique and the mandatory Access Control List (ACL) mechanism. The former technique is widely used to detect compromised applications and analyze them to minimize the harm that they can cause [2–6].

This approach relies on setting up policies that impersonate a legitimate application system call and then suspend or terminate the execution if the application pretends to be legitimate. Though the SCM technique alone cannot completely protect an end-system, it can be used as an additional technique to strengthen the detection capabilities of the Intrusion Detection System. The latter approach which implements a Mandatory Access Control (MAC) mechanism requires a malware's unique signature to define a Security Policy Specification (SPS). However, a user can restrict access privileges to various system resources. The existing MAC security solutions such as APPArmor [7] and grsecurity [8] that rely on an authorization mechanism

*For correspondence
Published online: 02 June 2020

allow a user to enforce strong security policies. These MAC solutions are implemented in Linux OS to supervise access rights to different system resources by applying SPS. AppLocker [9], a solution provided by Microsoft to protect an enterprise by identifying an application either belongs to whitelist or blocklist. However, it supports only computer associated policies and falied to detect a user who runs an application with privileged policies. Nevertheless, our approach (PAM) relies on authenticating each suspicious process during runtime, though the user runs the application with highest privilege. In the past, an online anomaly-based detection technique [10] has been proposed to identify a suspected malicious execution path of an application which relies on measuring similarities between execution paths. However, few malicious executables are likely to behave like legitimate applications which are very hard to distinguish.

In our prior work [11], we developed a novel algorithm namely, Concealed Process and Services Discovery Algorithm (CoPDA) for ascertaining the hidden entries of a malicious application by comparing two different lists of entries. In this paper, we authenticate all suspicious processes disvocered by the CoPDA algorithm. In this paper, we present a novel secure scheme with the intention of improving system security. Our work authenticates suspicious processes during runtime to detect malicious system call invocations from abusing and accessing kernel system services. We have implemented PAM in Windows environment. We downloaded 2000 malware samples [12–14] and analyzed their technical details to help us design an effective anti-malicious system. In addition to that, we explored the issues of low usability and incompatibility of existing anti-rootkit detection tools. We have observed the following. First, compatibility is an issue because they fail to distinguish the originality of benign and malicious system call invocation which increases the false positive rate. Second, poor usability is a problem because existing anti-rootkit detection tools act as standalone applications and thus can be victims of a malicious attack code. The results of this investigation allowed us to develop a novel approach that prevents malicious code attacks from making the system call invocation directly. Our experimental results on the function of PAM indicate that it can prevent trusted kernel services from being misused by malicious applications. The contribution of our work is as follows.

i. We present PAM, an authentication mechanism which permits only legitimate system calls to get kernel service; thus preventing misuse of system resources. This is achieved by cryptographically verifying the identities of suspicious processes of an application during runtime.
ii. We implemented PAM on Windows OS to prevent malicious code attacks without the need of malware unique signature in advance.
iii. The simulation results confirm that the proposed PAM scheme causes minimal computational overhead and strengthens overall system security. In addition, PAM offers good usability and good compatibility.

The remaining part of this paper is outlined as follows. Section 2 presents the existing works. Section 3 describes the security model and elaborates the proposed authentication security mechanism. Section 4 describes our implementation set-up, evaluation results and also discusses features such as security, compatibility, usability and limitations. Finally, we present conclusions and future work in section 5.

2. Present work

Microsoft Windows treats the kernel of the operating system as a black box; hence designing a monitoring framework become a challenging task. This is one of the reasons why only limited work has been proposed in the Windows environment to strengthen kernel security, specifically to prevent system services against malicious code attacks. The method proposed by Nguyen *et al* [15] captures malicious code attacks by hooking the SSDT in the kernel mode. Therefore, any malicious code which does not follow the predefined execution flow will be detected as an unauthentic code. Their idea was implemented without modifying the kernel of the OS. However, we have found some limitations. First, invocation of a system service request with incorrect dispatch ID results into system crash. Second, implementation difficulty was neglected in which it failed to prove the system's security strength. Third, guessing attack can easily compromise their solution.

Malware is the commonly used tool by a remote attacker to compromise the victim system in a network. As signature-based detection systems rely on policies and try to correlate possible patterns to detect intrusions, they cannot detect novel malware attacks. Weiqin *et al* [16] developed a compiler level prototype tool namely, Auto Shadow to generate the shadow process of a malware related to the original malware. This was accomplished by generating a behavior-based sequence graph using the system calls invoked by the original malware. The solution detects malicious behavior by matching the system call with existing malicious behavior. This technique is more robust, but it is difficult to detect known attacks. Attackers can successfully inject malicious code into a privileged process which takes full control of the system, Battistoni *et al* [17] proposed a method namely, WHIPS for detecting malicious behavior of a system call invocation made by software applications on Windows. As Windows kernel act as a black-box, WHIPS is implemented as a kernel driver and does not require kernel recompilation. WHIPS relies on policies for the detection of critical system calls, however it

is difficult to determine and set exact policies for unknown attacks.

Josse *et al* [18] proposed a secure framework for rootkit detection and monitor few critical components of the OS. A manual analysis framework was devised to automatically extract information about a rootkit and its interaction with OS executables. However, the system proposed by Jose et al. required human intervention to analyze and take a decision to exactly determine the presence of a rootkit migbot. A botnet is a group of zombies called as bots which is purposefully used to represent compromised end-computers. As all bots in a botnet behave similarly, Chung *et al* [19] proposed a behavior based approach to detect a specific gamebot such as migbot which is possibly rogue software program disseminated using a deceptive software marketing method. This approach relies on monitoring and grouping the behaviors of the users. If all the users in a group mimic same, then it indicates the presence of a bot. However, cryptographic based execution of bot like malicious program makes its detection more difficult.

Today, developing a protection system against malicious code attacks which can terminate an Antivirus software is a challenging problem. Hsu *et al* [20] proposed a mechanism to protect Antivirus software against Antivirus terminator programs. This approach worked at the kernel level by hooking the SSDT data structure to intercept a set of specific APIs and analyze their parameters to filter out harmful API calls that could terminate Antivirus software. Their approach has taken only 18 Antivirus terminator program samples for testing and also relied on the known behavior of the malware. In addition, it requires an exact behavior of Antivirus terminators to protect Antivirus software. As a virtual machine acts as a fitting testing environment for understanding a malware's behavior, researchers can install and try new applications without worrying about the type of the malware they have tested. Shan *et al* [21] proposed a behavior-based approach in which the malware affected the OS level information which was grouped into clusters. A suspicious cluster is determined when it exhibits malware behavior. Though their approach produces a low false-positive rate, collecting the scattered memory objects together is a time consuming and difficult task. Kinebuchi *et al* [22] used paging techniques and memory key features to inspect codes and data of a potentially compromised operating system to detect the presence of a rootkit. This approach does not require the support of hardware virtualization, but scanning code and data segments of different pages in memory is not a comprehensive step.

Baliga *et al* [23] developed an anomaly-based rootkit detection tool to infer invariants on kernel data structures of the target computer. It checks whether the data structure of the target's kernel satisfies the inferred invariants. The tool can detect only 23 different rootkits with low runtime overhead, but persistent modifications of kernel data can restrict the process of inferring accurate invariant information. Li *et al* [24] presented a compiler based approach to prevent rootkits from overwriting the kernel control data with arbitrary points. It can prevent rootkit attacks by transforming kernel control data into indexes of jump tables in which only legitimate jump targets are allowed in the kernel's control flow graph. Since rootkits can inject jump instruction in any table, scanning them is a tedious process. Almohri *et al* [25] presented a framework for validating the identities of processes of user-level applications in kernel-level during runtime. Through this approach prevents unauthorized access to system resources, few critical issues are: (i) As this approach requires modification in the kernel, it is not possible to implement it on the Windows platform. (ii) It can be able to identify applications only that run as stand-alone processes. A similar work was proposed by Sun *et al* [26] for preventing malicious code attacks that hooks native API calls in the kernel-mode. This approach was implemented in Windows environment and captures the malicious code attacks by authenticating a system call during run-time. The suggested process authentication approach ensures system level security against malicious code attacks in the kernel mode. Similar to our PAM, this approach was also implemented as a kernel-level plug-in by hooking SSDT data structure. However, we observed some issues. First, the proposed security solution looks simple and keeps all the credential information in plain format in stack. Second, there was no sufficient proof given to prove an existential forgery attack. Third, protecting system services that call higher level APIs was omitted.

Rajagopalan *et al* [27] presented a system call monitoring approach based on policy based authenticated system call. An authenticated system call is ensured by generating MAC from policy arguments, current execution state, system call number, and regular arguments of the call, call site and a cryptographic key. Whenever an authenticated system call is invoked, the kernel compared the computed MAC value with generated MAC value during runtime. Though this sandboxing approach minimized the damage caused by the compromised applications, it required the use of static analysis to generate policies. It failed to prevent stealthy malicious code attacks and precise setting of policies to combat unknown malware. Also, the use of different arguments with different length might not be the right choice to generate secret information. Many other existing anti-rootkit detection tools or API hook detection tools which rely on identifying and locating hooked identities used to compare the system call return address suffered from high false positive rate and system call reordering issues.

In this paper, we present a novel idea which exposes negligible overhead compared to Sun *et al*'s solution [26] and also overcomes the above listed drawbacks. PAM integrates process creation and monitoring in the user-mode with kernel authentication verification, thus PAM not only capable of identifying the hidden footprints of a malicious

software, but also authenticate its system call invocations in the kernel mode before they cause damage to the end-system.

3. Model and overview

A process can be identified using different tokens such as Process identifier (Pid), Process handles, Process name, etc. On the other hand, process authentication for a process has to confirm its identity to the OS during validation. The process authentication technique is very rarely discussed. Today, almost all types of OS include features to make access decisions. However, two important issues arise. First, the OS cannot classify a system service call as either legitimate or malicious. Second, such a security mechanism looks simple which would be bypassed by advanced malware attacks. Our technique differs from these solutions in such a way that each file path which is claimed by a process needs to be checked that it is legitimate rather than trusting what was achieved earlier. With stealthy malicious code attack techniques, human's timely reaction to intrusion detection is not possible. Also, Microsoft Windows is the most widely used OS, and attacking a considerable number of systems in a local area network by compromising a single system is possible. Therefore, we concentrate on enhancing the security strength of the kernel to improve overall system security.

The ultimate goal our PAM is to authenticate all suspicious system call invocations during runtime to protect system resources. Microsoft does not permit researchers to modify kernel components; thus we implement PAM as a kernel mode driver to guide all suspected system call invocations into the security monitor. Therefore, PAM does not require modification in the kernel code.

3.1 *Security model*

(i) *The design goals of PAM* – the goal of PAM design is to guarantee that the kernel of the OS appropriately authenticates each system service call raised by an application during runtime and ensures that malicious code cannot pretend to be a legitimate process.
(ii) *Malware Attack Type* – advanced stealthy malicious code on the victim computer may run without user intervention as a user-level process. A remote attacker can inject malicious code into software and force it to abnormally execute the injected code. During the code injection phase, the malware attempts to perform a certain operations in the user-space. The injected malicious code creates duplicate processes that is necessary for an execution in the user-space. Then, the malicious code may pretend to be a legitimate process by spoofing the process names. Hence, we cannot use process names as unique information for differentiating running processes. There are many different requirements for generating inimitable secret data for process authentication problems. Some of them are common and can be found in other credential schemes, but a few are uncommon and new.

3.2 *Proposed design*

This section presents our PAM method that seeks to disallow malicious system call invocation OS by verifying its originality. The objective of PAM is to authenticate each suspected process system calls, whereas all legitimate process will get kernel service directly which indirectly improves the overall performance of our PAM. PAM retains kernel issued credentials and knowledge of each suspected process in the user-mode. This information is queried by the kernel during the authentication phase to determine whether the system call can be allowed or not. The PAM acts as a sandbox that can prevent malware from violating the kernel's predefined security policies. Our PAM comprises of four components: the security monitor (Mtr), the Preservation Function (PF), the Credential Information Generation Function (CIGF) and the Runtime Authenticator. Figure 1 shows the design components of our PAM and its detail functions are as follows.

- First, each process manipulation by an application is monitored by the Mtr module and checked by the CoPDA algorithm. If found suspected, it goes through the PF and CIGF functions which are customized in ntdll.dll.
- During processing PF and CIGF, critical information will be backed up. The suspected process then allowed entering into the kernel mode for the first time.
- The runtime authenticator validates each suspected process and authenticate its originality. The runtime authenticator does not permit malicious processes that attempt in demanding kernel services.
- Processes which succeed authentication stage can acquire the system service.

In the following sections, we explain how each of the components work in details.

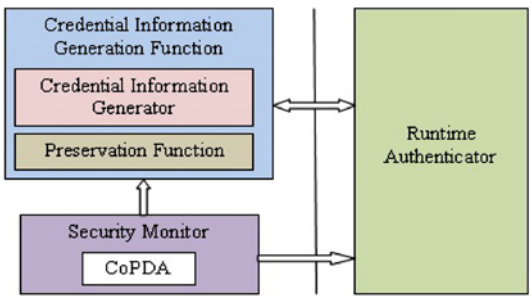

Figure 1. Design Comonents of PAM.

3.3 *Security monitor*

The Mtr component is responsible for monitoring the process manipulation on a system-wide basis. To prevent the installation and execution of malicious programs, we control process creation in the user-mode on a system-wide basis by hooking the NtCreateSection() function which cannot be easily bypassed by any process to launch a new process. Whenever a new process is created in the user-mode, it is monitored by the Mtr and is tested by the CoPDA algorithm. Intercepting processes and verifying each and every incoming system service request during runtime is a tedious and time consuming task. Therefore, the CoPDA algorithm allows each legitimate system service call invocation to be serviced as normal and classifies the remainder as suspicious.

When a computer is compromised, both inside and outside attacker will habitually try to hide different information such as malicious processes, data files, memory access or network connections, etc. Advanced stealthy malware includes functions to hide its footprints from antimalware detection software and stay undetected for longer time to perform their malicious operations.

A hidden process and service will typically not come into view with the user-space process list, namely, Visibleview (V) which is visible to end-users. However, it will have footprints on the kernel-space (lower level) namely, Globalview (G). Thus, the list G reflects the truth information about all system call invocations which can be extracted using lower level kernel API function i.e., G is subject to kernel's perspective. And, the list V represents currently running entities from the end user's perspective which is subject to alteration by stealthy malware.

We can discover hidden entities of malicious applications by comparing the currently running processes and services against the same information obtained from the kernel. If an entity is in G but not in V, it is concluded as a suspicious entity. The main steps of Globalview algorithm and Visibleview algorithm are shown in figures 2 and 3, respectively.

After loading and initializing the necessary information, the Mtr suspends the main thread of each suspected process. Next, the Mtr hooks each suspected process by inserting guidehook.dll into it by allocating necessary space using VirtualAllocEx function. Finally, the Mtr initializes the hooked dll by calling CreateRemoteThread and resumes the suspended thread. Therefore, whenever the hooked code is called, execution transfers to our detoured code to execute it indirectly after completion the control is transferred back to allow the original function for completion.

Operations of guidehook.dll: first, the memory is scanned to find the address of ntdll.dll which contains stubs of kernel API functions. In Windows, the ntdll.dll is the first module to be loaded, i.e., the first LDR_MODULE entry in InInitializationOrderModuleList. Since, the register EAX = PEB → Ldr. InInitializationOrderModuleList.FLink, then

Algorithm 1. Pseudo code for Globalview Algorithm

```
1.  Procedure Globalview Algorithm
2.  begin
3.  T←ø;
4.  G←process_creation;
5.  n←new_process;
6.  G←G ∪ {n};
7.    for each process in G do
8.          pid←handle.UnniqueProcessID
9.        //Enumerate handles and services of all running Processes
10.       if((pid ∈ CSRSS.exe) || (handle ∈ services.exe)) then
11.         T←T ∪ {current_ProcessID};
12.       endif
13.   endfor
14.  end
```

Figure 2. Algorithm for Global View of kernel-space.

Algorithm 2. Pseudo code for Visibleview Algorithm

```
1.  Procedure Visibleview Algorithm
2.  begin
3.  S←ø;
4.  V←snapshot(system);
5.  p←size(ProcessEntry32);
6.  for each process_handle in p do
7.        for each process in V
8.            S←S ∪ {ProcessID};
9.        endfor
10. endfor
11. end
```

Figure 3. Algorithm for Global View of kernel-space.

[EAX+0] ← List entry's FLink and [EAX+4] ← List entry's BLink. As a result, we can obtain the base address value of ntdll.dll at [EAX+8]. Then, the ntdll.dll is hooked by inserting the PF and CIGF into it by using the WriteProcessMemory function. Then, we find the entry point of each native API by inspecting ntdll.dll and replacing the *sysenter* command with a jump PF. Therefore, our detoured code will be executed first whenever a suspected process requests a system service.

Finally, a read-only page is created in the memory by setting PAGE_READONLY protection flag of VirtualProtect function where the address of a process authentication function is stored at M. It is easy to understand the description about complex words is given in table 1. In addition, the diagrammatic representation of this section is depicted in figures 4(a) and (b).

3.4 *Preservation function*

The Preservation Function is one of the kernel helper processes which reside in the user-mode. To avoid data

Table 1. Description of notation and complex word.

Notation/word	Description
Mtr	Security Monitor
VirtualProtect	It changes the access protection of a process
VirtualAllocEx	This function initializes and allocates the necessary memory
WriteProcessMemory	This function is used to write data to an area of memory in a specified process
LDR	It is a pointer to a PEB_LDR_DATA structure which actually contains information about the modules to beloaded in the process
EAX	Register
ESP	Stack pointer register
FLink & Blink	Forward Link and Backward Link
PEB	Process Environment Block

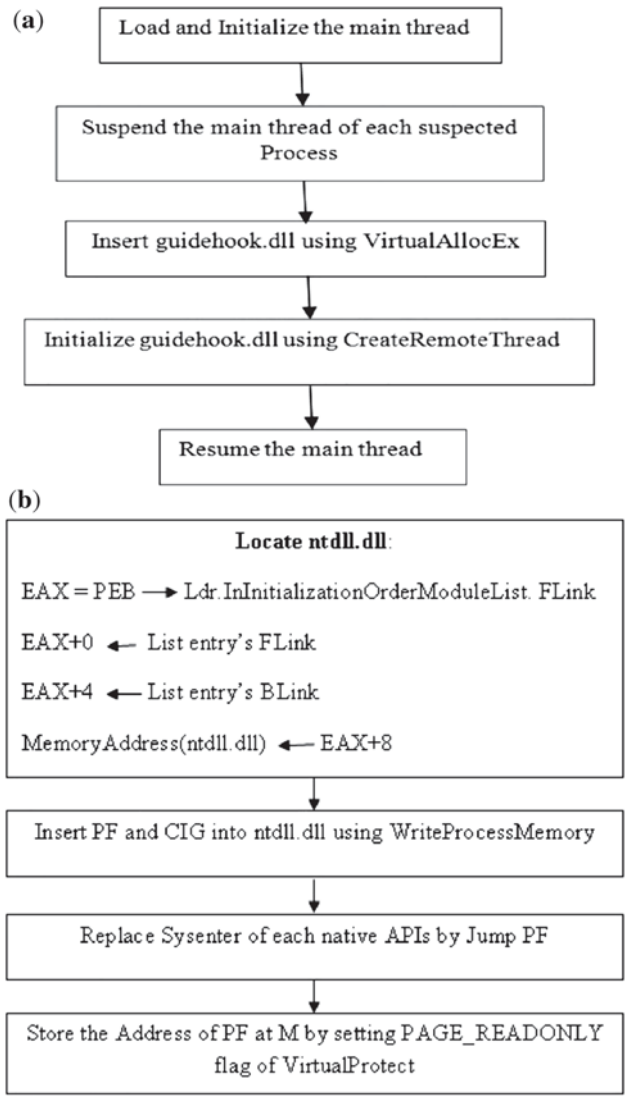

Figure 4. **(a)** Diagrammatic representation of Mtr. **(b)** Diagrammatic representation of guidehook.dll.

integrity problems while processing PF and Credential Information Generation Function, it is necessary to make a copy of their register values in advance. All these values are kept in the stack which will be restored later when the Credential Information Generation Function task is over. The EIP instruction pointer value is restored when the Runtime Authenticator queries the Credential Information Generation Function. To allow the Credential Information Generation Function to perform correctly, we backup the stack pointer ESP at memory address, E. If any malicious process/application tries to bypass this phase, it will fail to succeed in the authentication stage.

3.5 *Credential information generation function*

Hash based message authentication Code (HMAC) is the secure Message authentication code which works based on a hash function. The CIGF is another kernel helper process which performs its task only once for each suspected process of an application during its installation. The credentials are maintained by both, the CIGF and the kernel. Though there are several useful techniques are available to generate credential information, the simple approach is to make use of a pseudo random number generator which works under the control of the kernel. The kernel memory protects the *write* operation be default. Therefore, disabling the write operation to the memory area where kernel helper processes are stored can protect them against modifications by unauthorized parties. In addition, such kernel protected memory area can be accessed only by highest privilege components like, kernel and its helper components.

After backing up all necessary register values, the Credential Information Generator (CIG) which is a component of the CIGF is responsible for generating a 23 bit Strong Random Nonce (srn) using a pseudo-random number generator and register the (pid, p.srn) pair with the Runtime Authenticator. In order to protect the credential values, the CIG is controlled and maintained by the kernel; no other process can read or write into CIGF. Finally, the CIGF

computes the HMAC (h) using the combination of nine digit pid and 23-bit srn and stores the (pid, h) pair in the Credential Information List (CIL). The CIG also sets a short time frame, t, for the h value to expire to avoid replay attack. The credential information is no longer valid, if the application is removed or reinstalled or modified. The length of the srn is actually 23 bits which we can extend to 55 bits in x64 architecture to generate stronger credential information.

3.6 *Runtime authenticator*

The Runtime Authenticator (RA) is the kernel-mode component which is the heart of our design. Its goal is to authenticate each suspected process at the kernel mode during runtime before it is serviced. When a system service request enters into the kernel for the first time, the RA checks the crosscheck list (S) which only maintains the processes that have been successfully authenticated previously. If any request has not been authenticated, the RA queries the CGF by sending the query (Pid) to retrieve its hmac value. In response, the CGF replies with the h value of the corresponding Pid which is retrieved from CIL. If the returned h value is null or the delay associated with received h exceeds t value, the authentication check is disallowed. Otherwise, the RA recomputes h'←hmac(Pid, p.srn) and compares the values of h and h'. If they match, then the authentication check is successful. Otherwise p is treated as malicious. Finally, information about serviced system call entries are removed from the list S and the same is reflected in the CIL. The sequence diagram of the proposed PAM is illustrated in figure 5.

3.7 *Process authentication protocol*

Let p represent a new user process with a unique pid and p.srn represent the copy of p's secret information. We write hmac-req (p.pid) for sending p's secret credential information retrieval request to RA and a secure hash code generation function, HMAC. Figure 6 illustrates the sequence diagram of the process authentication protocol.

1. For each suspected process, p, the CGF performs the following operations:
   a. Generates a cryptographically Strong Random Nonce (srn) with a time frame, t and forwards (p.pid,p.srn) to RA. The time frame will expire some (short) time afterward or if there is no response from p.
   b. Computes h←HMAC(p.pid, p.srn) and stores (p.Pid, h) in CIL.
2. When p enters the kernel for the first time, RA performs the following operations:
   a. RA confirms whether p.pid $\in$ S. If so, it will be serviced by retrieving its pid and dispatch identifier.
   b. RA recalculates h'←HMAC(p.Pid, p.srn).
   c. If there is such a value found in the CIL, then p is reported as malicious.
   d. If a delay in receiving the HMAC exceeds t, then the authentication request will not be processed.
   e. RA compares h with h'. If it matches, the authentication request succeeds. Otherwise, p is reported as malicious.
3. When p completes, all its corresponding entries in both S and CIL are about to be deleted.

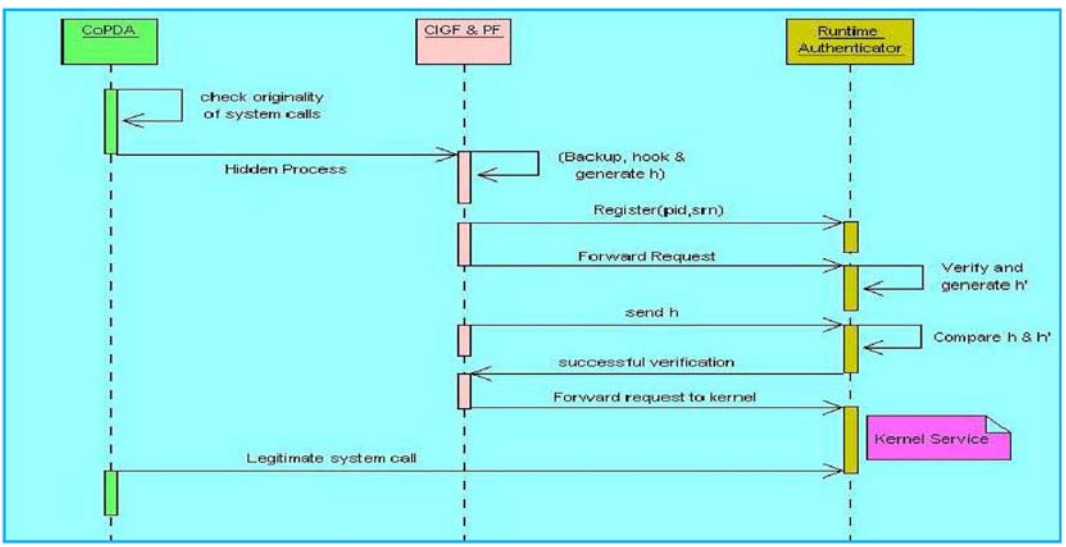

Figure 5. Sequence diagram of PAM.

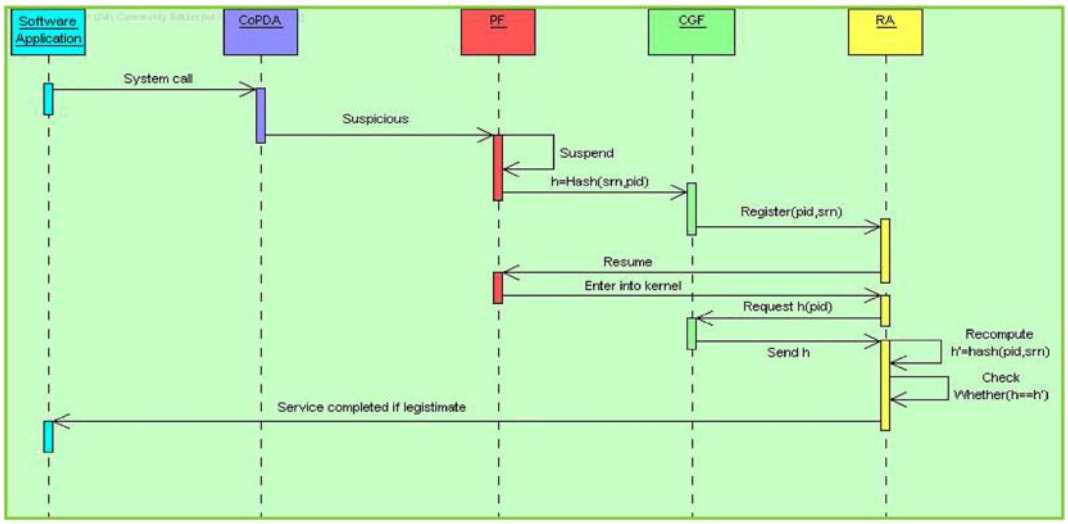

Figure 6. Sequence diagram of Process Authentication Protocol.

4. Implementation and evaluation

To assess the effectiveness of PAM, we designed a general framework for Windows 7 OS and performed a series of tests on Windows XP OS and Windows 7. This is because WOW64 (Windows 32-bit on Windows 64-bit) intercepts all OS system calls made by a 32-bit application. We selected the Windows operating system for three reasons. First, Windows is the most frequently used OS and the malware creators ensure that their creations will work in all types of OS from Windows XP to mobile OS. Second, the system calls and API functions of x32 bit applications will work on x64 bit OS without requiring additional settings. Third, Windows XP OS has been a much greater attractive target for many malware than other OSs.

4.1 *Implementation*

All experimental tests are conducted on a computer with 2.8 GHz Intel Pentium 4, 4 GB RAM and running Windows XP Professional OS as well as Windows 7. The Microsoft Windows Driver Development Kit [28] is used for developing the kernel driver module of our PAM. Except for the file-copy operation, the behavior of a native API function can be monitored only by intercepting one system call, for example, NtCreateFile(), and NtOpenFile(), etc.

However, behaviors like code-injection into other processes consist of the invocation of multiple system calls such as OpenProcess(), VirtualAllocEx(), WriteProcessMemory(), CreateRemoteThread(), etc. Therefore, we planned for intercepting the first system call itself to prevent the execution of subsequent calls which would disallow other subsequent calls.

4.2 *Evaluation*

We evaluate the effectiveness of PAM from two different aspects: it strengthens in guaranteeing security and the additional computational time after executing PAM . In addition, the overhead of PAM is mathematically verified.

*(a) Security - t*o validate the potential ability of our PAM to limit malicious code attacks that target system call hooking, we selected 100 malware samples from a collection of 2000 malware samples and purposefully chose 75 malware samples based on the attacking techniques employed and execution environment. In addition, we used 25 benign samples obtained from two reputable websites [29, 30]. To manipulate and validate the experimental results, we arranged a client-server model as a test-bed.

The noteworthy servers such as an FTP server, IIS server and IRC chat server included all malicious samples. Another server ran the only web server was in full trust mode and only held benign samples to examine trusted communications. The end-computer installed with Windows XP OS ran different client programs such as FTP client applications, email applications such as Thunderbird, IRC clients, web browsers such as IE and Firefox, newsgroups and eMule which are attractive targets of malware attacks. We defined certain protocols such as FTP, ICMP, IRC, SMTP and eMule as dangerous. We purposefully introduced malicious samples into the host and carried out different actions between the client and server. With this setup, the effectiveness of PAM can be carefully evaluated with and without considering it. We confirmed whether the malicious samples were executed on the host machine by using the log files generated by the tools with and without setting security protection. The same was manually verified in the log files of process, files and registry entries.

The outcome was confirmed by using three popular anti-rootkit detection techniques: GMER [31], IceSword [32], and Redline [33]. All these tools are capable of detecting rootkit malware samples that target SSDT hook attacks. We tested each security tool against all of the samples. For each kind of samples, we computed the total number of false positives and false negatives after launching PAM. We noted a false positive if a security tool incorrectly identified a benign activity as malicious. Figure 7 shows the False Positive Rate (FPR) obtained by testing all the four malicious code detection tools including PAM. Both GMER and IceSword anti-malware security tools incurred FPR about 24%, but in the case of Redline it was 32%, whereas PAM had FPR of 4%. On the other hand, the False Negative Rate (FNR) of GMER and IceSword was 20% and 15% respectively. As Redline's detection capability of SSDT hook attacks depends on the type of API function to be hooked, it achieved FNR by 65%. However, the FNR of PAM was almost zero. The PAM can effectively block any type of malware that targets system call hooks, but none of the existing anti-malware security tools tested by using a kernel-level authentication mechanism to verify the originality of a system call invocation.

(b) Performance overhead: Table 2 shows the performance overhead incurred by PAM was computed by measuring the CPU cycles to be taken for executing additional tasks such as intercepting each system call and API function and authenticating their originality. To evaluate the additional overhead caused by PAM, we setup a test-bed that ran with 2.8 GHz Intel Pentium 4, 4 GB RAM and running Windows XP professional OS with SP3 as well as Windows 7. The performance overhead of PAM is determined by estimating the additional time taken to execute the three important components of PAM.

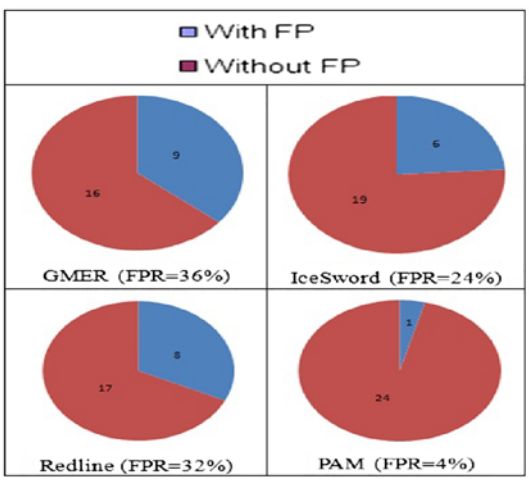

Figure 7. Comparison of FPR with existing anti-malware techniques.

(c) Hypothesis test: We should statistically check if there is a significant deviation in the performance of the kernel/system before and after enabling PAM. We defined the null and alternative hypothesis as follows.

H0: Statistically, there is no significant difference in the kernel performance after enabling PAM.

H1: Statistically, there is some association between before and after enabling PAM.

The given data in Table 3 show the CPU cycles in the two test cases. Hence the CPU cycles in the two tests can be regarded as correlated and therefore, the t-test for paired values was used to confirm the performance deviation between these two cases. Let $d = x_1 - x_2$ and $\bar{d} = \Sigma d / n$, where x_1 and x_2 denote the CPU cycles in the two tests and n is the number of functions tested.

Applying t-test, $t = \frac{\bar{d}\sqrt{n}}{S}$, where $\bar{d}$ the mean of the difference and S is the standard deviation of the difference. From t-table, for γ=n-1=4 degrees of freedom, $t_{0.05}$=2.776. In both cases, i.e., PAM-b and PAM-m, $t_{cal} < t_{0.05}$, hypothesis H_0 is accepted and it is concluded that there is no significant change in the kernel/system performance after enabling our PAM framework. In Microsoft Windows OS, there is no single system call invocation that manipulates a new process of an executable application.

In order to retrieve the name of the file and its execution path, the normalizer function needs to identify the NtOpenFile() function system calls which belong to the process manipulation task. In order to create a process, the OS can run a set of system calls to allocate necessary resources for its completion and start the new process: NtCreateSection() function which is one of the native API functions we used in CoPDA and NtOpenFile() function which is used for estimating the performance of the PAM. This indicates that PAM encounters multiple system call invocations (11,000 NtRead and NtWrite system calls) during its performance estimation process.

(d) Comparison of our scheme to related work: PAM is a process authentication mechanism that aims to prevent malicious processes of an application from misusing system resources. The CoPDA algorithm is responsible for discovering suspicious processes of a malicious application and CIGF is used to generate credential information and attaching it into each suspicious process. The Runtime Authenticator is more important and responsible for securely authenticating suspicious processes and validates their legitimate identities before getting kernel services. The notable strength of PAM is that it directly prevents malicious code API hook attacks rather than locating them and resulting in a false positive. We have developed PAM framework as plug-in in Windows, including all its three modules. There are few works have been proposed previously by researchers, but none of them dealt kernel level authentication to prevent malicious code attacks which mainly target SSDT hook technique. The security comparison of our PAM and other related schemes is presented in table 4.

Table 2. Performance overhead of PAM (CPU Cycles) (The columns PAM-b and PAM-m illustrate the CPU cycles acquired by the benign programs and malware programs).

		CPU cycles taken		CPU Overhead	
Function	Native	PAM-b	PAM-m	PAM-b (%)	PAM-m (%)
NtOpenFile	167703	169721	169823	1.2	1.3
NtWriteFile	245201	249993	338546	1.9	38.1
NtCreateFile	334568	338579	348579	1.2	4.2
NtCreateProcess	206556	208945	215326	1.1	4.2
OpenService	6568202	6568423	6679899	<0.1	1.7

Table 3. t-test Computation.

	PAM-b		PAM-m	
Function	d	d^2	d	d^2
NtOpenFile	2018	4072324	2120	4494400
NtWriteFile	4792	22963264	93345	8713289025
NtCreateFile	4011	16088121	14011	196308121
NtCreateProcess	2389	5707321	8770	76912900
OpenService	221	48841	111697	12476219809
	Σd=13431	Σd2=180391761	Σd=229943	Σd2=21467224255
	t_{cal} = 0.912871		t_{cal} = 1.970616	

Table 4. Comparison of our PAM to related work in terms of security.

Security Features	PAM	Hsu *et al* [20]	Sun *et al* [26]	Almohri *et al* [25]
Requirement of kernel recompilation	No	No	No	No
Target Applications	Suspicious Processes	Frangible APIs	All Processes	Stand alone processes
Resist hidden process attack	Yes	Neglected	Neglected	Neglected
Resist high-level API hook attack	Yes	No	No	No
Type of credential information	Kernel-level	Not used	User-level	Kernel-level
Resist existential forgery attack	Yes	No	Yes	Yes
Protection of user-mode components	Protected	Neglected	Neglected	Protected
Implementation Environment	Windows	Windows	Windows	Linux

Hsu *et al* suggested a solution to protect antivirus software against antivirus terminators. The suggested approach targets only few important kernel level APIs and failed to focus other API hook attacks. The work presented by Sun et al. caused 36.7% computational overhead in the system and does not provide strong cryptographic solution against advanced stealthy malware attacks. Almohri *et al* presented a similar work but developed and tested in Linux OS. The suggested scheme focused to attach cryptographic information into all applications and verify the attached information before obtaining kernel service. This mechanism actually increases the computational overhead significantly. PAM avoids this issue through authenticating only suspicious processes of an application and outperforms other realted schemes as presented in table 4.

4.3 *Discussions*

(a) Resist to Existential Forgery Attack

Theorem *A MAC, (pid, srn, t) is considered to be secure against existential forgery attack under a chosen process attack.*

Proof We will consider the contrapositive.

Suppose we have a forger R who produces successful existential forgeries against MAC. We build a MAC forger R' for (pid', srn', t').

Let we present security as the probability of winning a game, where the MAC is considered to be secure if no adversary can win the game. Let us consider the description of R' with the challenger i.e., CFG.

R chooses srn at random and queries CFG which responds with $(y_1,y_2,\ldots y_i)$ for some $(pid_1, pid_2, \ldots pid_q)$ generated randomly.

Set srn pub = $(y_1,y_2,\ldots y_i)$ and sends srn pub to R. For each query, R sends h(pidi) to CFG.

After at most q queries by the adversary, R produces a MAC and a positive candidate srn forgery X form M. If $L(M_i) = h(M)$ for some i ε {1, 2, …q}, then the CFG returns a forgery attempt (M,X) with $M = M_i$ for some i, but $M \neq Mi$.

(b) Security Assurance

The strength of security protection guaranteed by PAM was verified by analyzing the confidentiality of the credentials used on authentication stage and the integrity of PAM components. Without using a strong pseudorandom number generator for generating secret credentials, forging the existing credentials is impossible. Also, a malicious process's arbitrary code may try to replace PAM generated credentials cannot successfully bypass the authentication stage. This is because the arbitrary code which is not generated by PAM does not appear in the record. To prevent another application revealing the secret information generated to perform a challenge-response attack to be launched by a malware, PAM restricts read access. A malware may attempt to steal secret information from PAM components or an application's memory at runtime. This issue is resolved by using the typical process memory segregation feature offered in the OS itself. PAM ensures confidentiality by disallowing other applications that have direct access to the secret credentials except PF and CGF which are kernel helper processes.

PAM components span both the user-space and the kernel- space. The Runtime Authenticator resides in the kernel. As we trust the integrity of kernel resources, this component is trustworthy. However, the integrity of user-space components, the PF and the process CGF need to be confirmed as they may be the ideal targets of malware spoofing and tampering. Only the kernel of the OS can access or modify the code segment of these components.

(c)Usability

Software usability mainly concerns with the assessment of effectiveness and efficiency with which end-users can perform tasks with a software tool. Nowadays, assessing usability is an important element of the software development process. As PAM does not include any configuration settings, it automatically detects and prevents potential malicious code API hook attacks.

(d) Limitations

PAM is capable of classifying interpreted software applications by functioning as a stand-alone process. However, PAM cannot reveal the malicious code that is already injected using NTFS transactions into an authenticated process and runs as a stand-alone process. The strength of PAM lies in the accuracy of classification precision. The trustworthy classification of an application is a challenging and difficult task, and inexactness may permit a malware to acquire the secret credentials of a legitimate process.

5. Conclusions and future work

Though almost all stealthy rootkit malwares have integrated with different techniques, their main intention is to hide their traces and stay undetected for an extensive period of time to perform their illegal activities. Bearing this common character in mind, we presented, designed, implemented, and evaluated a kernel-level malicious code authentication prototype for a Windows environment to prevent malicious code attacks that target SSDT hook in the kernel mode.

We discussed how process authentication mechanisms can effectively isolate and disallow system calls malicious processes from getting system services and thus disable system resources against malicious code attacks. The authentication technique of PAM is portable and can be integrated with other static or dynamic behavior-based system call monitoring tools with customization. Our simulation results prove that PAM does not significantly affect the overall system performance and also produces negligible overhead.

Our PAM design is robust against malicious code API hook attacks and achieved good usability and compatibility. It is capable of defeating malicious code attacks that use the SSDT hook technique, including unknown malware samples. As cyber criminals work to sure that their malicious software creations work equally on systems from Windows XP to Android OS, we will focus in future on porting PAM to Android OS for mobile devices to provide strong authentication to applications.

References

[1] Desktop Operating System Market Share, http://www.netmarketshare.com. Last Accessed June 2017

[2] Bernaschi M, Gabrielli E and Mancini L 2000 Operating system enhancements to prevent the misuse of system calls. In: *Proceedings of the 7th ACM Conference on Computer and Communication Security (CCS'00)*, pp. 174–183

[3] Garfinkel T 2003 Traps and pitfalls: practical problems in system call interposition based security tools. In: *Proceedings of the Network and Distributed Systems Security Symposium (NDSS '03)*, pp. 163–176

[4] Garfinkel T, Pfaff B and Rosenblum M 2004 Ostia: a delegating architecture for secure system call interposition. In: *Proceedings of the Network and Distributed Systems Security Symposium (NDSS '04)*, pp. 187–201

[5] Provos N 2003 Implementing host security with system call policies. In: *Proceedings of the 12th Conference on USENIX Security Symposium (SSYM'03)*, pp. 257–272

[6] Sekar R, Venkatakrishnan V, Basu S, Bhatkar S and Duvarney D 2003 Model-carrying code: a practical approach for safe execution of untrusted applications. In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP'03)*, pp. 15–28

[7] Loscocco P and Smalley S 2001 Integrating flexible support for security policies into the linux operating system. In: *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pp. 29–42

[8] Grsecurity. http://www.grsecurity.net/. Last Accessed on August 2017

[9] AppLocker, https://www.beyondtrust.com/assets/documents/bt/Applockr-Least-Privilege.pdf. Last Accessed on October 2019

[10] Parsa S and Naree S A 2012 A new semantic kernel function for online anomaly detection of software. *ETRI J.* 34(2): 288–291

[11] Muthumanickam K and Ilavarasan E 2015 CoPDA: concealed process and service discovery algorithm to reveal rootkit footprints. *Malaysian J. Comp. Sci.* 28(1): 1–15

[12] Malware Samples. http://www.offensivecomputing.net/. Last Accessed on August 2017

[13] Malware Samples. VX Heavens. http://vx.netlux.org/. Last Accessed on August 2017

[14] Malware Samples. Xfocus Team. http://www.xfocus.net/. Last Accessed on August 2017

[15] Nguyen L Q, Demir T, Rowe J, Hsu F and Levitt K 2007 A framework for diversifying Windows native APIs to tolerate code injection attacks. In: *Proceedings of the 2nd ACM symposium on Information, computer and communications security (ASIACCS '07)*, pp. 392–394

[16] Ma W, Duan P, Liu S and Gu G 2012 Shadow attacks: automatically evading system-call-behaviour based malware detection. *J. Comput. Virol.* 8(1): 1–13

[17] Battistoni R, Gabrielli E and Mancini L V 2004 A host intrusion prevention system for windows operating systems. *Lecture Notes in Computer Science*, 3193: 352–368

[18] Josse S 2007 Rootkit detection from outside the matrix. *J. Comput. Virol.* 3: 113–123

[19] Chung Y, Park C, Kim N, Cho H, Yoon T, Lee H and Lee J H 2013 Game bot detection approach based on behavior analysis and consideration of various play styles. *ETRI J.* 35(6): 1058–1067

[20] Hsu F-H, Wu M-H, Tso C-K and Chen C-W 2012 Antivirus software shield against antivirus terminators. *IEEE Trans. Inf. Forensics Secur.* 7(5): 1439–1447

[21] Shan Z, Wang X and Chiueh T 2013 Malware clearance for secure commitment of OS-level virtual machines. *IEEE Trans. Dependable Secure Comput.* 10(2): 70–83

[22] Kinebuchi Y, Butt S, Ganapathy V, Iftode L and Nakajima T 2013 Monitoring integrity using limited local memory. *IEEE Trans. Inf. Forensics Secur.* 8(7): 1230–1242

[23] Baliga A, Ganapathyand V and Iftode L 2011 Detecting kernel-level rootkits using data structure invariants. *IEEE Trans. Dependable Secure Comput.* 8: 670–684

[24] Li J, Wang Z, Bletsch T, Srinivasan D, Grace M and Jiang X 2011 Comprehensive and efficient protection of kernel control data. *IEEE Trans. Inf. Forensics Secur.* 6: 1404–1417

[25] Almohri H M J, Yao D and Kafura D 2014 Process authentication for high system assurance. *IEEE Trans. Dependable Secure Comput.* 11(2): 168–180

[26] Sun H-M, Wang H, Wang K and Chen C 2011 A native APIs protection mechanism in the kernel mode against malicious code. *IEEE Trans. Comp.* 60(6): 813–823

[27] Rajagopalan M, Hiltunen M A, Jim T and Schlichting R D 2006 System call monitoring using authenticated system calls. *IEEE Trans. Dependable Secure Comput.* 3: 216–229

[28] Windows Driver Dev. Kit, http://www.microsoft.com/download/en/. Last Accessed on August 2017

[29] Benign Samples Collection, http://www.download.com. Last Accessed on August 2017

[30] Benign Samples Collection, http://technet.microsoft.com. Last Accessed on August 2017

[31] GMER V1.0.15.15087, http://www.nonags.com/freeware-gmer-3786.html. Last Accessed on August 2017

[32] IceSword v 1.22, https://icesword.jaleco.com. Last Accessed on August 2017

[33] Redline, http://www.mandiant.com/product/free-software/redline/. Last Accessed on August 2017