Prison: Tracking Process Interactions to Contain Malware

*Benjamin Caillat, Bob Gilbert,
Richard Kemmerer, *Christopher Kruegel, and *Giovanni Vigna

University of California, Santa Barbara
{*benjamin, rgilbert, kemm, chris, vigna*}*@cs.ucsb.edu*
and
*Lastline, Inc.
www.lastline.com

***Abstract*—Modern operating systems provide a number of different mechanisms that allow processes to interact. These interactions can generally be divided into two classes: inter-process communication techniques, which a process supports to provide services to its clients, and injection methods, which allow a process to inject code or data directly into another process' address space. Operating systems support these mechanisms to enable better performance and to provide simple and elegant software development APIs that promote cooperation between processes.**

Unfortunately, process interaction channels introduce problems at the end-host that are related to malware *containment* and the *attribution* of malicious actions. In particular, host-based security systems rely on process isolation to detect and contain malware. However, interaction mechanisms allow malware to manipulate a trusted process to carry out malicious actions on its behalf. In this case, existing security products will typically either ignore the actions or mistakenly attribute them to the trusted process. For example, a host-based security tool might be configured to deny untrusted processes from accessing the network, but malware could circumvent this policy by abusing a (trusted) web browser to get access to the Internet. In short, an effective host-based security solution must monitor and take into account interactions between processes.

In this paper, we present PRISON, a system that tracks process interactions and prevents malware from leveraging benign programs to fulfill its malicious intent. To this end, an operating system kernel extension monitors the various system services that enable processes to interact, and the system analyzes the calls to determine whether or not the interaction should be allowed. PRISON can be deployed as an online system for tracking and containing malicious process interactions to effectively mitigate the threat of malware. The system can also be used as a dynamic analysis tool to aid an analyst in understanding a malware sample's effect on its environment.

I. INTRODUCTION

Contemporary operating systems are designed to offer a flexible platform that supports a diverse set of applications. As such, performance and ease of software development are primary design considerations. These design goals influence an important system trade-off between two diametrically opposed features: process isolation and inter-process communication. Process isolation ensures that a process cannot corrupt the data or code of another process. Inter-process communication provides a channel over which data, code, and commands can be exchanged. If a system isolates processes weakly, then communication is straightforward, for example, by allowing processes to directly read from and write to each other's memory. In contrast, strongly isolated processes may only communicate in ways that are more complicated and performance-intensive, for example, by using remote procedure calls (RPCs).

Completely isolated processes are secure against external tampering, but they are not very useful in and of themselves. Inter-process communication features provide a mechanism by which processes can share data and, perhaps, influence each other's execution in a controlled manner. Thus, process isolation suffers as inter-process communication becomes more flexible and easy to use. Due to the design goals mentioned above, modern systems tend to favor diverse and flexible inter-process communication mechanisms over strong process isolation. Unfortunately, an overly permissive security model combines with weak process isolation to provide an environment in which malware can abuse its ability to freely interact with other processes.

For instance, consider a personal firewall that limits outgoing network traffic to a few trusted applications, such as Internet Explorer for web traffic and Outlook for email. If a malicious process directly attempts to make an Internet connection, the firewall can successfully block the attempt, as it represents a violation of the policy. However, the malicious process can send inputs to the Internet Explorer application in order to drive the browser to execute the request on behalf of the malware. For example, the malware might utilize a COM interface that Internet Explorer exposes or, more generically, it may directly write code into the browser's address space and execute it. In this case, the firewall sees a network request coming from the trusted Internet Explorer process, so it allows the traffic to pass through unabated. This is an example of the *confused deputy problem* [5], and it is characteristic of the access control systems that contemporary operating systems use.

In this paper, we propose PRISON, a system that monitors process interactions to detect and mitigate malicious inter-process communication and code or data injection. In particular, we developed a host-based component that monitors the system calls that enable process interactions. PRISON can detect when a system call is used by one process to interact with another. When such an interaction is found, the system can automatically invoke one of several responses to mitigate the impact of malicious code: block the communication, alert the user, or update the privileges of the target process. Our focus in this paper is on demonstrating the practical impact on end-host security when inter-process communication is not properly taken into account. In addition, we describe the variety and complexity of different inter-process communication channels on a real-world operating system such as Microsoft Windows, and we discuss the challenges that need to be overcome to monitor these channels. Finally, we show how PRISON can be used as the basis for a containment system that prevents malware from abusing trusted processes.

Tracking process interactions in a comprehensive manner is difficult, because operating systems provide a diverse set of communication interfaces using protocols that are often undocumented. As evidence of this difficulty, we evaluated a number of host-based security products, and we showed that these systems fail to capture all process interactions (these experiments are described in Section IV-C). This failure implies that malware is able to effectively "launder" its malicious actions through trusted processes to avoid detection and containment.

We implemented our system as a Windows XP kernel extension. PRISON hooks a variety of kernel system services (system calls) to monitor a diverse set of interaction channels. Our experiments demonstrate that PRISON is effective at blocking unauthorized process interactions over all the mechanisms that we are aware of. Furthermore, we show that our efficient technique offered finer granularity and better coverage than commercial tools. We also verified that the lack of process isolation is a security problem on more recent versions of Windows (in particular, Windows 7), and we ported a subset of PRISON as a proof-of-concept to demonstrate that our proposed approach remains valid.

To summarize, we make the following contributions:

- We perform a comprehensive analysis of the mechanisms on the Windows platform that allow one process to interact with another. This analysis drives our evaluation of a variety of commercial malware defense systems, and we demonstrate that each system could be evaded by a number of attacks that are made possible by weak process isolation.
- We propose a novel approach to malware containment by preventing a malicious process from interacting with another process to perform actions on its behalf. We describe the design and implementation of PRISON, a system that extends the Microsoft Windows XP kernel to implement our proposed approach. Moreover, we verified that Windows 7 faces similar security problems, and we ported parts of PRISON to this platform to demonstrate the validity of our system design.
- We discuss our experimental results, which show that our system is able to efficiently defend against attacks where process interactions are attempted in order to carry out malicious actions. Furthermore, we validate that our analysis is comprehensive by comparing the process interactions we identify to those described by an oracle that does whole-system data tracking.

II. SYSTEM OVERVIEW

Our solution to address the problem of malicious process interaction is PRISON, a host-based system that operates as a kernel extension to the Windows XP platform. In order to get a detailed understanding of how inter-process communication is implemented in Windows, we performed a deep analysis of the communication mechanisms that are available in the system. PRISON monitors all known kernel interfaces to detect when processes interact. The system can block the communication attempt if the source violates a security policy. Alternatively, PRISON can alert the user, asking for permission before allowing a specific interaction.

In the current implementation, PRISON only considers processes that cannot tamper with the kernel (e.g., by loading a driver). We therefore only consider malware running under a non-administrator user account. The system could be extended to processes with administrator permissions by adding a supplementary mechanism that would protect the kernel (and PRISON) from modification. We leave this for future work, but we note that such protection mechanisms have been implemented in the past [15], [20].

Nevertheless, Microsoft is aggressively moving towards an unprivileged user account model [12]. Furthermore, recent studies suggest that malware is increasingly adapting to such an environment, and it runs properly without administrator access [2].

Modes of Operation. PRISON has two modes of operation: *logging mode* and *filtering mode*. In logging mode, PRISON maintains a real-time *interaction log* of all interactions between processes. The system also generates an *interaction graph* that depicts these interactions. For each process, the log includes an entry that enumerates the interactions that involve the process as a target (e.g., an RPC client request or a write into the process' memory space). The log also contains related information, such as the list of LPC ports or named memory sections that the process has opened. The interaction graph offers a visual representation that concisely depicts a set of interactions. Because viewing an interaction graph of the whole system quickly becomes unmanageable, PRISON can focus the graph on specific processes that are of interest.

PRISON can be queried as a service by other malware detection systems that may use the interaction information for their real-time analysis. Our system makes this feature available through an input and output control (IOCTL) interface. Furthermore, PRISON can be used as a dynamic malware analysis tool. To this end, an analyst can configure the system to write its interaction log and graph to disk so that they can be used for offline analysis. The interaction log could also be included in the reports of malware analysis sandboxes, such as Anubis [7] or CWSandbox [21].

In filtering mode, PRISON interposes on each process interaction attempt, and it makes a policy-based decision that determines whether or not this interaction is allowed. If the interaction is permitted under the policy, then the communication request passes unabated to the system service that is responsible for fulfilling the request. However, if the interaction attempt indicates a policy violation, then it is blocked. In this case, the source process is returned an error code that indicates that the target is unavailable and the communication fails.

Interaction Filtering Policies. We have identified three *interaction filtering policies* for use in PRISON's filtering mode. The first policy is a whitelisting approach in which a list of system processes (e.g., `Explorer.exe` and `Csrss.exe`) and trusted applications are granted access to the inter-process communication facilities that they require, and all other interactions are blocked. A straightforward way to identify applications for such a whitelist is by process name, similar to how the "path rule" operates in the Windows Software Restriction Policies mechanism [13]. However, if a stronger notion of identity is required, we could leverage existing work in the area of code identity, including the use of a TPM [16] or a dynamic code identity primitive [4].

An alternative filtering policy is a *process tainting* approach. This policy requires an additional kernel module that augments the Windows access control mechanism with a set of high-level system access rights, such as *file system*, *registry*, and *network*. These rights are granted to a process according to its needs when it is created, and the rights are enforced when the process attempts the corresponding access. Whenever a source communicates with a target, the latter obtains the *intersection* of the two process' respective high-level rights. For example, assume that the Notepad text editor and Internet Explorer browser are granted {*file system*, *registry*} and {*file system*, *registry*, *network*} high-level access rights, respectively. If Notepad attempts to communicate with Internet Explorer (perhaps due to an attack against the text editor), then the browser is left with {*file system*, *registry*} rights, and, thus, would be unable to use the network, since it has been "tainted" by Notepad. Similarly, if any malicious process (that does not have the rights to access the *network*) attempts to induce Internet Explorer (IE) to perform a request on its behalf, the IE process will be stripped of its permission to make network connections. Thus, connection "laundering" attempts are thwarted.

The third policy is a user-guided filtering approach. Under this policy, the user is notified (e.g., via a dialog box) when a suspicious interaction is attempted. The user can then make a decision whether to allow the interaction or not. This is similar to the way that traditional antivirus alerts operate in the presence of detected malware. This policy is the most conservative of the three, as it offloads the filtering decisions to the user. Thus, the policy might be best suited to power users who demand full control of their systems.

The policy-based filtering mode makes PRISON a powerful tool for detecting and mitigating the threat of malicious process interactions. This is important because malware authors have a variety of options at their disposal for leveraging a benign process to perform actions on their behalf. We describe the design and implementation of PRISON next.

III. SYSTEM IMPLEMENTATION

In this section, we describe how PRISON is implemented to detect and mitigate malicious process interactions. To interpose on all communication attempts, PRISON must intercept the function calls that implement the interaction techniques. PRISON accomplishes this by hooking the relevant system services in the Windows XP kernel. A few techniques can be used to hook system services [6]. We opted for a standard approach that is used by many security products that target the Windows XP platform ; we patch the *System Service Dispatch Table* (SSDT). This table is used by the operating system to dispatch system call requests to the corresponding system services.

We opted to implement our prototype system on the Windows XP platform, for several reasons: First, more recent kernels integrate some protections that would make the integration of Prison more difficult. Second, it remains a very popular OS with a large (kernel-mode) software development community. Lots of companies are still reluctant to upgrade from Windows XP [8], [9]. Furthermore, we had access to the Windows Research Kernel (WRK). This provided us with some of the Windows XP kernel source code, which aided our analysis and development efforts. Finally, we decided to evaluate the completness of PRISON with a process interaction Oracle based on the project Anubis [7]. The versatility of Anubis considerably eased this development. Unfortunately, this project is only compatible with the 32-bit version of Windows XP. We discuss our efforts to port PRISON to Windows 7 in Section III-C.

A. The Interaction Filtering Policy

PRISON utilizes a whitelisting interaction filtering policy to allow a set of system processes and trusted applications to interact with other processes. The policy limits these trusted processes to only use the mechanisms that are necessary for them to operate. The whitelist is essentially a file that lists the full path and name of the trusted process and a set of interaction techniques that the process is known to utilize during normal operation. We could automatically generate a whitelist by defining a training period in which we track all normal interactions that occur between trusted processes on the host and then add them to the whitelist. This is similar to the policy generation mechanisms that exist in other systems, for example, [14]. This filtering policy provides a simple mechanism by which PRISON can prevent malware from interacting with other processes on the host.

B. Monitoring Relevant System Services

We performed a detailed analysis of the techniques that enable processes to interact on the Windows platform. Overviews of many of these techniques are readily available [10]. Fundamentally, each interaction method is implemented by one or more system services. In this section, we describe the services that PRISON must hook to successfully monitor all of these interaction techniques.

Interposing on most of the interaction mechanisms is a straightforward process of identifying and hooking a single corresponding system service. Unfortunately, handling other techniques is more difficult. In particular, some interaction channels require a deep understanding of undocumented structures and interfaces to map relevant objects to the corresponding source and target processes. It is important to capture this mapping, because PRISON requires the identity of the communicating processes for its analysis. In the following, we describe our engineering challenges and our methods of solving them.

Windows Messages. Windows exposes a set of functions that can be used to send Windows messages, including `SendMessage`, `PostMessage`, and `PostThreadMessage`. These functions resolve to calls to a matching set of system services: `NtUserMessageCall`, `NtUserPostMessage`, and `NtUserPostThreadMessage`, respectively. To intercept the exchange of Windows messages, PRISON hooks these three system services. For each service, PRISON analyzes the message type parameter to detect the type of interaction (e.g., a `WM_COPY` message indicates a clipboard copy operation). The target process is determined from another parameter that specifies a handle to the window that the message is meant for.

Another way in which malware can abuse Windows messages is to attach to the input processing mechanism of another thread. This technique is often used by keyloggers to capture the keystrokes of the user. A user-mode process can call `AttachThreadInput` to accomplish this task. PRISON hooks the corresponding system service, `NtUserAttachThreadInput`, and identifies the target process as described above.

A particular problem arises when attempting to hook user interface (USER) or Graphics Device Interface (GDI) system services, like the ones described above. While most NT services are handled by the kernel itself, USER and GDI services are implemented by the kernel-mode part of the Windows subsystem (in an extension called `Win32k.sys`). For this reason, USER and GDI functions are exported by a secondary shadow SSDT. This additional SSDT is only mapped into the calling process when one of the corresponding services is first requested. This is problematic in cases where PRISON attempts to hook the USER services in the context of a process that has not mapped the shadow SSDT. PRISON solves this as follows: First, it attaches to a process that is known to have already mapped the shadow SSDT (by calling `KeStackAttachProcess`). Second, PRISON locates the physical address of the table. Third, it reverts to the original process context and uses a memory descriptor list (MDL) to map the table into that context. Finally, PRISON can safely patch the shadow SSDT.

Dynamic Data Exchange. The Win32 API provides an interface to the Dynamic Data Exchange (DDE) message passing protocol through functions such as `DdeConnect` and `DdeGetData`. Fundamentally, DDE relies on Windows messages to implement the protocol. In particular, a set of messages, delimited by `WM_DDE_FIRST` and `WM_DDE_LAST`, are sent from the source to the target. Thus, monitoring DDE communication reduces to the handling of Windows messages, as we described above.

Shared Memory. Shared memory functionality in Windows is implemented over the file mapping mechanism. In particular, the provider of a shared memory region will first call `CreateFileMapping`, specifying a name for the shared memory region and a special value, `INVALID_HANDLE_VALUE`, in place of the file handle. Next, the provider calls `MapViewOfFile` to actually map the shared memory into the process' virtual address space. Consumers typically request access to the shared memory by calling `OpenFileMapping`.

PRISON hooks both the `NtCreateSection` and `NtOpenSection` system services to analyze creation and access requests to shared memory, respectively. Determining the identity of the target (producer) process is not entirely straightforward since this information is not available when the consumer calls `NtOpenSection`. We devised a novel solution to this problem as follows: First, when the producer invokes `NtCreateSection`, PRISON logs the name of the region and the process ID of the producer in an *identity cache*. Later, when the consumer requests access to the shared memory, PRISON uses the identity cache to match the name of the region to the producer's ID to identify the target.

Another subtlety is that the consumer may sometimes invoke `CreateFileMapping` (and, thus, `NtCreateSection`) to access the shared memory region. In this case, PRISON distinguishes between create and open operations by intercepting the call and first attempting to open the section itself. If the attempt succeeds, then PRISON interprets the call as an open request by a consumer.

Named Pipe and Mailslot Communication. The operational semantics of named pipe and mailslot communication are nearly identical. Thus, we will only discuss the monitoring of the former here. A server process initializes a named pipe by calling the `CreateNamedPipe` API, providing a unique name to identify the endpoint (e.g., `\\.\Pipe\SomeNamedPipe`). The client subsequently connects to the server by specifying the pipe name in a call to `CreateFile` or `CallNamedPipe`. If successful, the client receives a handle that represents the connection, which it uses to send and receive data over the pipe (by calling `WriteFile` and `ReadFile`, respectively).

It is straightforward to identify the client process of a named pipe communication attempt, because the call is made in the client's context. Determining the identity of the target (i.e., the server) is more challenging, for two reasons. First, the client sends and receives data over the pipe using the generic `WriteFile` and `ReadFile` APIs. These functions take a file handle as a parameter. The Windows Object Manager uses this handle to identify the client side of the pipe connection, but the handle offers PRISON no information that identifies the server. Second, the data that is sent over the pipe represents a custom protocol between the client and server, so no identifying information can be gleaned from the messages themselves. Our solution is similar to how we handle the identification of the endpoints that communicate over shared memory. In particular, PRISON hooks `NtCreateNamedPipe` to capture the name of the pipe that the server creates along with the process ID of the server. This information is stored in PRISON's identity cache. Later, a client invokes the `NtCreateFile` system service to establish a connection with the server. By hooking this service, PRISON can acquire the pipe name from a parameter and then use the identity cache to match the name to its corresponding server ID.

Local Procedure Call. The Local Procedure Call (LPC) mechanism is designed to be used only by system processes. However, its API is exposed by `Ntdll.dll`, so it is possible for malware to leverage the protocol for inter-process communication. A server creates an LPC port by invoking the `NtCreatePort` or `NtCreateWaitablePort` system service, supplying a name for the port. A client later connects to the port by calling `NtConnectPort` or `NtSecureConnectPort`. Once a connection is established, both client and server communicate by using handles to their respective communication port objects.

PRISON analyzes LPC communication by applying the same strategy that is used for named pipes and mailslots. In particular, PRISON stores (in the identity cache) a mapping between the name of the LPC connection port and the corresponding process ID of the server that created it. When a client connects to the port (e.g., by calling `NtConnectPort`), PRISON extracts the port name and matches it to the process ID that it previously stored to identify the server.

Remote Procedure Call. Remote Procedure Call (RPC) is a flexible inter-process communication mechanism that operates over Winsock (i.e., the TCP/IP protocol stack), named pipes, or LPC. Since PRISON is a host-based system, it ignores RPCs that use TCP/IP network communication. The other two cases are handled by the internal mechanisms themselves, as discussed in the corresponding sections above. PRISON is able to distinguish RPC communications that operate over LPCs from other LPCs due to the naming convention that the RPC framework uses when creating its ports. This is beneficial for logging purposes, but it is not strictly necessary for analysis. Unfortunately, no such convention seems to exist for RPCs that operate over named pipes.

Component Object Model. The Component Object Model (COM) is built upon a communication stack that includes RPC operating over the LPC protocol (for local COM components) or TCP/IP (for remote components). Correctly handling communication over COM is particularly challenging, for the following two reasons. The first difficulty is that detecting the target process that hosts one or more COM objects does not offer enough granularity for PRISON to decide whether or not to block the interaction. The reason is that one process can support multiple COM objects, and we need to distinguish between them. For example, the Background Intelligent Transfer Service (BITS) exposes a COM interface through which a client may transfer files over the Internet. BITS is typically hosted in a shared service process (a Service Host process, `Svchost.exe`) along with a number of other components. In this case, identifying the target process does not allow PRISON to meaningfully analyze the interaction endpoint. Instead, PRISON must identify the particular COM object itself. The second issue is that local COM components dynamically create LPC communication ports with names that do not infer the identity of the endpoint. This makes it exceedingly difficult for PRISON to determine which COM component a particular request is destined for.

We invested a substantial engineering effort to distinguish the COM server endpoint in a given communication attempt. This included intercepting requests to the Endpoint Mapper component to obtain the names of dynamic LPC ports and reverse engineering the payloads of various COM-related

LPC messages. In particular, we needed to decode custom marshaled object references to obtain the identities of the requested interface and endpoint. With this information, PRISON is able to effectively analyze COM interactions. Unfortunately, due to space constraints, we cannot provide all of the details of our implementation here.

Code and Data Injection. There are a number of ways for malware to inject code or data into a target process. These methods result in invocations of the following system services: `NtWriteVirtualMemory`, `NtMapViewOfSection`, `NtCreateRemoteThread`, `NtSetContextThread`, and `NtQueueAPCThread`. PRISON hooks all of these services and analyzes them to obtain the source and target processes of the interaction attempt in a straightforward manner.

Windows Hook Injection. A process can install a Windows hook by calling the `SetWindowsHookEx` API. PRISON interposes on the corresponding system service, `NtUserSetWindowsHookEx`. The service takes a thread ID as a parameter, which designates the thread that the hook targets. PRISON uses the thread ID to determine which process the hook is meant to target. However, it is possible for the source to pass a value of zero as the thread ID when invoking the system service. In this case, the hook is to be installed in all threads that are running within the active desktop. PRISON handles this situation by simply blocking the hook request.

C. PRISON *on Recent Windows Systems*

The problems concerning malicious process interactions still exist on more recent versions of the Windows operating system. This is largely because a primary design goal of these systems is to maintain backward compatibility with legacy software, which has prevented a substantial change to the system architecture. To demonstrate that PRISON also operates on more recent versions of Windows, we ported our system to the (32-bit) Windows 7 platform. This was accomplished using the same SSDT hooking approach that we previously used for XP. In doing so, we discovered that a number of kernel system services have been augmented with newer versions. For example, a new system call, `NtCreateThreadEx`, supersedes the `NtCreateRemoteThread` service, and a family of functions, `NtAlpcXxx`, implements the new ALPC mechanism. It would be straightforward to extend PRISON's monitoring capabilities to account for these services as well, but we did not find much value in reengineering our system to simply account for all of these new functions.

The 64-bit versions of Windows operating systems introduce the Kernel Patch Protection (KPP) feature to prevent SSDT hooking (and other modification of kernel structures) on these platforms [11]. Since we leverage SSDT hooking to intercept kernel functions, KPP renders our current implementation inoperable on this architecture. We could leverage existing work on KPP bypassing techniques [17]–[19] to implement our system on 64-bit Windows as well. However, this is not an ideal solution. Rather, we note that if Microsoft were interested in adopting our system, the company would do so by implementing PRISON in the kernel itself (instead of as an extension). This could be accomplished by inlining our checks into the relevant system services.

IV. EVALUATION

In this section, we discuss the results of our evaluation of PRISON in five areas. First, we demonstrate that our system can monitor process interactions while maintaining compatibility with benign applications. Second, we show that PRISON can be used to effectively contain malware that attempts to interact with trusted processes. Third, we describe how our system compares to host-based security products. Fourth, we show that our system comprehensively monitors interactions that occur between processes on the host. Finally, we demonstrate that PRISON introduces only a negligible performance overhead.

A. *Effect on Application Compatibility*

As we described in Section III, PRISON is implemented as a kernel extension to the Windows XP operating system. Since our system operates at a low level in the host, it is given unfettered access to all executing processes. We need to ensure that PRISON does not adversely affect the standard operation of these applications while our system interposes on all process interactions. We devised three experiments to evaluate this.

First, we executed a ten-step representative workflow: (1) launch Internet Explorer (IE), (2) navigate to the Google search engine, (3) search for "windows filetype:pdf", (4) click the first search result to open a PDF document in the Adobe Reader IE plug-in, (5) copy text from the document onto the clipboard, (6) start Notepad, (7) paste the text into the text editor, (8) enter a number into Notepad and copy it onto the clipboard, (9) start Calculator, (10) paste the number into Calculator. This workflow represents a typical user's interaction with the system and provides a good measure to determine PRISON's effect on normal system operation. Our system was able to effectively monitor all process interactions without a compatibility break in any applications in the workflow.

Second, we wanted to ensure that PRISON does not adversely affect the operation of popular applications with varying functionality. To this end, we gathered a set of 16 programs, including a debugger (OllyDbg), IDE (Visual Studio), text editor (Wordpad), set of media players (GOM Player, RealPlayer, and VLC), browser (Chrome), game (Mario Forever), toolbars (uTorrentBar and Google Toolbar), and various system utilities (Partition Master, ARO 2011, 7Zip, Wireshark, TrueCrypt, and Process Explorer). We manually ran each application in order to exercise its core functionality. For example, we wrote a simple program in Visual Studio and then debugged it in OllyDbg, and we (legally) downloaded a video file with uTorrentBar and viewed it in RealPlayer. All of the applications ran without issue under PRISON.

Finally, we wanted to determine PRISON's effect on benign programs that do not have explicit entries in the interaction whitelist. For this purpose, we created a test suite of 560 executables taken from the Windows directory. We ran each program for five seconds and logged the resulting interaction attempts. In total, we identified 10,903 distinct interaction events. Of these interactions, 80 were blocked by PRISON. We analyzed the cause of the blocked interactions and found that 73 were due to memory read operations and 7 were DDE communication attempts. For example, 26 read requests were made by `blastcln.exe`, a tool that checks for Blaster and Nachi infections in memory. Furthermore, the Windows Presentation Foundation Host process and XPS Viewer attempted to communicate with Internet Explorer over DDE. Since the number of blocked interactions was quite small (0.7% of the total), it would be easy to update the interaction whitelist to accommodate these interactions. Alternatively, we could augment the whitelist with the user-guided filtering approach (discussed in Section II) to handle these few interactions on a case-by-case basis.

These results demonstrate that PRISON can be deployed on a host to analyze interactions without adversely affecting the processes that typically run. In particular, our whitelisting filtering policy allows all of the interactions among trusted processes to occur, while still being able to contain malware. We evaluate this latter aspect next.

B. Containing Malicious Process Interactions

Malware often interacts with other processes when exploiting the host that it infiltrates. Frequently, malicious software injects code into trusted processes so that its illicit operations appear to come from the latter. In this way, the malware has a greater chance of going unnoticed by the victim or antivirus software that may be running on the user's behalf. We deployed PRISON in a virtual environment and executed a number of malware instances to determine how well our system can defend against malicious process interactions. We ran each malware sample two times: first, with PRISON configured in logging mode and second, with the system in filtering mode. This way, we could use PRISON to learn more about the samples' operations and then demonstrate the system's ability to contain the malware.

We executed three malware samples that perform code injection into the address space of other running processes. The first malware instance was a Zeus bot that injects code into the Winlogon process as part of its installation. The second sample was a Korgo bot that injects a thread into the Windows shell, `Explorer.exe`. The third sample was a suspicious application called YGB Hack Time that injects a DLL called `Itrack.dll` into most running processes.

PRISON successfully blocked all of the malware samples from interacting with other processes. Zeus was contained because it attempted to inject code into `Winlogon.exe`. The logging mode graph showed us that Zeus successfully communicated with the Winlogon process, and the injected code created a named pipe through which the malware issued commands to its components. By contrast, with filtering mode enabled, PRISON blocked this interaction, thereby preventing Zeus from completing its installation. Korgo was blocked because a child process that it creates attempted to inject code into `Explorer.exe`. YGB was mitigated because it attempted to set a Windows hook into all running processes, and PRISON filters global Windows hook requests. These results show that malware uses a variety of interaction mechanisms, which underscores the importance of our broad coverage. Furthermore, the results demonstrate that PRISON is an effective tool for both dynamic malware analysis and malware containment.

C. Evaluating Security Products

The previous experiment demonstrates that PRISON is able to effectively mitigate the threat of malware that attempts to interact with other processes. A number of commercial host-based security products claim to feature techniques that also accomplish such blocking. We wanted to evaluate these tools to determine the degree to which they are able to filter malicious process interactions, particularly in comparison to PRISON. Furthermore, we will discuss how two prominent academic research systems fail to cope with malicious process interactions.

Attack Tool Suite. In order to test the commercial security products, we developed a suite of attack tools that exercise a number of process interaction techniques. The tools are divided into two classes: inter-process communication tools and injection tools. In particular, the tools that perform inter-process communication leverage the following attacks: One tool sends `WM_SETTEXT` Windows messages to an instance of Notepad to display arbitrary text in its window. Another tool acts as a keylogger that uses the `Attach-ThreadInput` API to capture keystrokes. A third attack tool uses the `WWW_OpenURL` DDE command to force Internet Explorer (IE) to download a binary on behalf of the attacker from a specified website. This tool also utilizes COM to drive IE (using the `Navigate` method of the `IWebBrowser2` interface) or the Background Intelligent Transfer Service (using `IBackgroundCopyManager` interface methods) to accomplish the binary download. Finally, a set of tools create attack scenarios to exercise communication over shared memory, named pipes, mailslots, and RPCs, respectively. In each case, the tools launch a fictitious (trusted) server that consumes messages that are delivered by an attacker that uses the corresponding communication mechanism.

The goal of the injection tool is to inject an attacker's code into a target process and then execute that code. The tool attempts the actual injection in one of three ways: First, the tool can write code into the target by using the `WriteProcessMemory` API. Second, it can map a file that contains the code into the target's address space (using `MapViewOfFile`). Finally, the tool can use the `SetWindowsHookEx` API to inject a DLL into the target. The next step is to force the target to execute the code. The tool uses three mechanisms for this: `CreateRemoteThread`, the debugging API, or an asynchronous procedure call (APC). Note that in the case of injection via `SetWindowsHookEx`, the execution occurs automatically, since the entry point of the attacker's DLL will be called when the module is loaded. Otherwise, in order for the attack to be successful, the tool must leverage at least one technique from both stages (i.e., injection and forced execution).

Commercial Security Products. We evaluated five different commercial security products against our attack tool suite. We used the most recently available versions when we ran our experiments. The products include BitDefender Total Security 2012, Kaspersky Internet Security 2011, McAfee Total Protection 2011, Outpost Internet Security Suite v7.5, and ZoneAlarm Extreme Security 2012. We tested each product in two scenarios. First, we installed the free evaluation version of the products and employed their out-of-the-box configuration. Second, we used the products' configuration options to manually harden them against attacks. While this improved the detection of malicious interactions in some cases, it also frequently led to many unrelated false alarms. Using our attack tool suite, we evaluated both scenarios for each product as well as PRISON.

Summary of Results. We found that BitDefender, Kaspersky, and McAfee were not offering any process interaction protection in their default configuration. After hardening BitDefender and Kaspersky, the detection improved somewhat, but the products still only detected 53% and 41% of the attacks, respectively. McAfee did not provide an option to improve its security posture. Outpost and ZoneAlarm did have security configuration options, but the settings had no effect on our attacks. In particular, the hardened configuration for these products each matched their out-of-the-box detection rates of 71% and 41%, respectively. PRISON was able to detect and block all of the attacks except for RPC communication over the TCP/IP transport (a detection rate of 94%). However, as we discussed in Section III-B, PRISON ignores communication attempts to endpoints over the network, so we do not consider this attack to be a problem. By contrast, each of the aforementioned commercial tools failed to prevent against infections that abuse interactions with trusted processes. That is, these products are designed to prevent malware from leveraging a "download and execute" style of attack, but they failed to do so when such an attack is laundered through a trusted process.

D. Completeness

A system that claims to monitor all process interactions must ensure that a process is not able to bypass the analysis by making use of an unknown interface that the system did not anticipate (i.e., the system should be complete). Furthermore, such a system must not introduce false positives by claiming that interactions have occurred that, in fact, did not (i.e., the system should be accurate). In order to test these properties with respect to PRISON, we implemented an *interaction oracle* that does whole-system dynamic taint tracking on the Windows XP platform.

Our oracle tracks all data flow through the operating system. It accomplishes this by tagging all data with a taint label that corresponds to the process to which the data belongs. In this way, when one process interacts with another such that the source passes data to the target, the latter will acquire taint labels that are associated with the former. Therefore, the propagation of taint labels through this system demonstrates all instances in which process interactions occur.

By running PRISON on a host that is also analyzed by the interaction oracle, we can compare the results of the two perspectives. Specifically, we can determine if the taint labels that are propagated by the oracle correspond to interactions that PRISON identifies. If so, this gives us assurance that PRISON is an effective tool for monitoring all of the interactions that occur on the host (given the test cases).

Oracle Implementation. We implemented our taint tracking system as an extension to the Anubis malware analysis sandbox [7]. Anubis itself is built upon QEMU, a powerful virtualization environment that performs processor emulation through dynamic binary translation [3]. This gives our oracle complete control over the guest operating system on a per-instruction basis. To perform data tracking, the oracle leverages the taint propagation facilities that Anubis provides [1]. Thus, our focus was on implementing a taint introduction policy that suits our needs.

The interaction oracle introduces taint into the system at a byte-level granularity. The oracle analyzes each write instruction. When a source operand is tainted, the destination operand is tainted with the same label. If not, the oracle determines if the address of the destination operand is within a specially tracked memory region – namely, a stack, a heap, or a data segment. If this is the case, the oracle taints the destination address with the label that corresponds to the current process. The intuition behind this taint introduction policy is that a process will write its data into the stack, heap, and data segment as part of its normal operation (e.g., when initializing local variables or writing to a dynamically allocated buffer). Thus, the oracle taints all values that a process writes (and that do not derive from already tainted data) with the label that corresponds to this process. Furthermore, taint is introduced when certain events occur. For example, when a process initializes its read-only data segment (`.rdata`) or maps a file into its address space, the oracle taints these memory regions appropriately. This taint policy allows the oracle to accurately track all instances in which data flows from one process to another.

Evaluating PRISON Against the Oracle. We deployed PRISON in a virtualized environment with the interaction oracle running. We again performed the experiments with the attack tool suite as we previously described in Section IV-C. We manually compared the perspectives of the two systems in order to evaluate the completeness of our approach.

The oracle's log contains, on a per-process basis, a list of virtual address ranges and the process taint labels that are associated with each region. For each process in the log, we extracted a list of unique taint labels that corresponded to other processes. This list represents the oracle's summary of the processes that interacted with the given process.

We compared the oracle's process interaction list to the output from PRISON's interaction log and attempted to find corresponding entries that explain the interaction. In all cases, we found that the process interaction was accounted for. A few spurious taints were introduced into the analysis, but we manually investigated each of these instances and discovered that they were artifacts of residual taint propagation and should be discounted. For example, in some cases, the oracle reported a few taint labels in a process' address space that corresponded to another process that terminated before the given process began execution. These instances can be safely ignored. Thus, we are confident that PRISON monitors all known process interaction mechanisms.

We also investigated all of the process interactions that were reported in PRISON's interaction log to determine if a corresponding set of taint labels existed in the oracle's log. In all cases, each interaction that was reported by PRISON was also captured by the oracle. This gives us confidence that PRISON does not introduce nonexistent interactions.

E. Performance Impact

PRISON is implemented as an extension to the Windows XP kernel, and it must monitor a number of system services that support process interactions. These kernel functions are executed frequently, so it is important to measure the overhead that our system introduces with respect to these invocations.

To this end, we devised three microbenchmarks to evaluate PRISON's impact on process interaction performance. The first two experiments were designed to simulate common inter-process communication scenarios. In particular, we launched an RPC server that listened on an LPC port and named pipe, respectively. Then, a client connected to each server and invoked 100,000 RPC calls. Recall that COM operates over RPC, so these tests encompass a broad range of inter-process communication mechanisms. In the third experiment, we emulated a direct data injection attack. Specifically, a source process allocated a region of memory in the address space of a target process and then executed 1 million calls to `WriteProcessMemory`. Table I shows the results. The overhead can be attributed to the additional locking and accounting that PRISON performs. This particularly impacts LPC tracking, due to the additional analysis required to identify potential COM endpoints. While the overhead may seem high, we note that these microbenchmarks represent specific interaction operations that comprise a small portion of the host's overall activity; the measurements do not constitute a degradation of the entire system. As such, we believe that the overhead is acceptable.

Table I: Average execution times (in milliseconds).

Interaction mechanism	Standard	With PRISON	Overhead
RPC over LPC	1834	4194	129%
RPC over named pipe	3098	4209	36%
`WriteProcessMemory`	6421	8748	36%

V. SECURITY ANALYSIS

In this section, we analyze the security of our proposed approach to monitoring malicious process interactions. The malicious process could attempt to evade the whitelist filtering policy by setting its path and name to match that of a trusted program that is on the whitelist. For this attack to succeed, the malware would need to overwrite a trusted application. Clearly, this is not viable with respect to system processes, since they reside under the privileged `System32` directory. Nonetheless, it may be possible to subvert the policy if the system administrator adds applications in unprivileged paths to the whitelist. In this case, we could use a stronger mechanism to establish process identity, such as a code identity primitive [4], [16].

It may be argued that whitelisting policies in general are difficult to maintain due to an ever changing software ecosystem. However, we learned from our experiments that although there are many interactions within the operating system, the interactions are primarily limited to standard, trusted system processes (such as `Csrss.exe` and `Explorer.exe`) that are well-known and long-lived. Furthermore, we note that global distribution mechanisms already exist (such as Microsoft Update), which our system could leverage to perform periodic policy maintenance.

Malware could also interact with a target process indirectly. For example, a malicious process may place a DLL on the file system and leverage the `AppInit_DLLs` legacy feature to execute it. We solve this problem by only allowing trusted processes to write to the `AppInit_DLLs` registry value. Additionally, the malware could edit files on disk that trusted processes may read for configuration. We do not currently address this attack, but we are considering ways to augment our system with a file system interaction tainting approach. This is left as future work.

VI. Conclusions

In this work, we present PRISON, a system that monitors and blocks all malicious process interactions on the Windows XP platform. Our system is implemented as a host-based kernel extension that interposes on inter-process communication among processes and injection attempts by which one process loads code or data into the address space of another. By dynamically monitoring the interaction behavior among processes on the host, PRISON can protect a user from malware that attempts to leverage a trusted process to carry out its malicious actions. Furthermore, PRISON can output interaction logs and interaction graphs that provide a malware analyst with a concise summary of the interactions that pertain to processes of interest. We have evaluated our approach along a number of axes. The results demonstrate that PRISON is deployable, useful, and efficient. Future work will investigate the handling of indirect mechanisms for interaction (e.g, overwriting a target's configuration file) with a file system interaction tainting technique. Also, we are interested in exploring how new filtering policies may affect the flexibility of our approach.

References

[1] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, Behavior-Based Malware Clustering. In *Symposium on Network and Distributed System Security (NDSS)*, Feb. 2009.

[2] U. Bayer, I. Habibi, D. Balzarotti, E. Kirda, and C. Kruegel. A View on Current Malware Behaviors. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, Apr. 2009.

[3] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference (ATC)*, Apr. 2005.

[4] B. Gilbert, R. Kemmerer, C. Kruegel, and G. Vigna. Dymo: Tracking Dynamic Code Identity. In *Symposium on Recent Advances in Intrusion Detection (RAID)*, Sept. 2011.

[5] N. Hardy. The Confused Deputy (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, Oct. 1988.

[6] G. Hoglund and J. Butler. *Rootkits: Subverting the Windows Kernel*. Addison Wesley Professional, 2005.

[7] International Secure Systems Lab. Anubis: Analyzing Unknown Binaries. http://anubis.iseclab.org/.

[8] John Wenz. One Year Later, Millions of People Are Still Using Windows XP. http://www.popularmechanics.com/technology/a14980/still-using-windows-xp/.

[9] Lance Whitney. Windows XP clings to No. 2 spot as Windows 10 gets closer. http://www.cnet.com/news/windows-xp-use-continues-to-drop-but-still-in-no-2-spot/.

[10] Microsoft Corporation. Interprocess Communications. http://msdn.microsoft.com/en-us/library/windows/desktop/aa365574.aspx.

[11] Microsoft Corporation. Kernel Patch Protection for x64-based Operating Systems. http://technet.microsoft.com/en-us/library/cc759759(WS.10).aspx.

[12] Microsoft Corporation. New UAC Technologies for Windows Vista. http://msdn.microsoft.com/en-us/library/bb756960.aspx.

[13] Microsoft Corporation. Description of the Software Restriction Policies in Windows XP. http://support.microsoft.com/kb/310791, Sept. 2009.

[14] N. Provos. Improving Host Security with System Call Policies. In *USENIX Security Symposium*, Aug. 2003.

[15] R. Riley, X. Jiang, and D. Xu. Guest-Transparent Prevention of Kernel Rootkits with VMM-based Memory Shadowing. In *Symposium on Recent Advances in Intrusion Detection (RAID)*, Sept. 2008.

[16] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *USENIX Security Symposium*, Aug. 2004.

[17] skape and Skywing. Bypassing PatchGuard on Windows x64. *Uninformed*, Jan. 2006.

[18] Skywing. Subverting PatchGuard Version 2. *Uninformed*, Dec. 2006.

[19] Skywing. PatchGuard Reloaded: A Brief Analysis of PatchGuard Version 3. *Uninformed*, Sept. 2007.

[20] J. Wilhelm and T. Chiueh. A Forced Sampled Execution Approach to Kernel Rootkit Identification. In *Symposium on Recent Advances in Intrusion Detection (RAID)*, Sept. 2007.

[21] C. Willems, T. Holz, and F. Freiling. Toward Automated Dynamic Malware Analysis Using CWSandbox. *IEEE Security and Privacy Magazine*, Mar. 2007.