Provenance-based Integrity Protection for Windows

Wai Kit Sze and R. Sekar
Stony Brook University
Stony Brook, NY, USA
{wsze,sekar}@cs.stonybrook.edu

ABSTRACT

Existing malware defenses are primarily reactive in nature, with defenses effective only on malware that has previously been observed. Unfortunately, we are witnessing a generation of stealthy, highly targeted exploits and malware that these defenses are unprepared for. Thwarting such malware requires new defenses that are, by design, secure against unknown malware. In this paper, we present SPIF, an approach that defends against malware by tracking code and data origin, and ensuring that any process that is influenced by code or data from untrusted sources will be prevented from modifying important system resources, and interacting with benign processes. SPIF is designed for Windows, the most widely deployed desktop OS, and the primary platform targeted by malware. SPIF is compatible with all recent Windows versions (Windows XP to Windows 10), and supports a wide range of feature rich, unmodified applications, including all popular browsers, office software and media players. SPIF imposes minimal performance overheads while being able to stop a variety of malware attacks, including Stuxnet and the recently reported Sandworm malware. An open-source implementation of our system is available.

1. INTRODUCTION

The scale and sophistication of malware continues to grow exponentially. The reactive approach embodied in malware scanners and security patches is no match for today's stealthy, targeted attacks. Recognizing this fact, researchers as well as software vendors have been developing *proactive techniques* that can protect against previously unseen exploits and/or malware attacks. These techniques can be classified into three main categories: *sandboxing, privilege separation,* and *information flow control.*

Sandboxing techniques [13, 35, 45, 49] mediate all security-relevant operations performed by applications, permitting only those deemed "safe" by a sandboxing policy. The scope of damage that can result from a malicious (or compromised) application is hence limited by this policy. On UNIX, applications frequently targeted by attacks are typically protected using SELinux [24] or AppArmor policies [45]. Microsoft Office used a sandbox for its *protected view* [27]. This sandbox ensures that a compromised process cannot overwrite system or user files or registry entries.

Privilege separation techniques [36] refine the sandboxing approach to support applications requiring significant access to realize their functionality. The application is decomposed into a small, trustworthy component that retains significant access, and a second larger (and less-trusted) component whose access is limited to that of communicating with the first component in order to request security-sensitive operations. While sandboxes can confine malicious as well as frequently targeted benign applications (e.g., browsers), privilege separation is applied only to the latter class. Chromium browser [38], Acrobat Reader and Internet Explorer are some of the prominent applications that employ privilege separation, more popularly known as the *broker architecture.* Both applications sandbox their renderers, which are complex and are exposed to untrusted content. As a result, vulnerabilities in the renderer (or more generally, a *worker*) process won't allow an attacker to obtain all privileges of the user running the application.

Information flow control (IFC) techniques [2, 50, 19, 21, 42, 25, 44, 43] maintain labels on files and processes to keep track of the flow of sensitive and/or untrusted information in the system. Classical integrity policies such as the Biba policy [2] enforce both no-read-down (i.e., integrity-critical applications cannot read untrusted data) and no-write-up (i.e., untrusted applications cannot create or overwrite high-integrity files) policies. In contrast, *Windows Integrity Mechanism* (WIM) [28] enforces just the no-write-up policy. Indeed, WIM is primarily deployed as a sandboxing mechanism: progressively more restrictive policies are enforced on lower integrity processes, while high-integrity processes are unconfined. In contrast, the strength of integrity protection in IFC stems from policy enforcement on high-integrity processes, which prevents them from compromised by consuming untrusted data or code.

1.1 Challenges

Application of these three approaches for malware defense poses several technical as well as practical challenges.

Policy development. Policy affects both usability and functionality of applications. Restrictive policies can block more attacks, but they also tend to break applications. Moreover, policy development requires not only a good understanding of applications, but also the OS semantics. A recent Adobe Reader XI vulnerability [14] exploits the semantics of *junctions* on NTFS, where the broker process failed to sanitize paths and ended up allowing workers to create files at arbitrary locations.

Application and OS compatibility. To run successfully with a policy and its enforcement framework, applications need to be re-architected, or at a minimum, be made aware of the confined environment. Most IFC approaches require non-trivial changes to applications as well as the OS. There have been efforts to automate some of the steps (e.g., automating privilege separation [4]) or to minimize application changes for IFC (e.g., PPI [42] and PIP [44]), but in practice, most

†This work was supported in part by grants from NSF (CNS-0831298 and CNS-1319137).

ACSAC '15, December 07 - 11, 2015, Los Angeles, CA, USA

ISBN 978-1-4503-3682-6/15/12...$15.00
DOI: http://dx.doi.org/10.1145/2818000.2818011

techniques end up requiring substantial effort in rewriting or porting applications or the OS.

Sandbox escape attacks. Given the large effort needed to (a) develop policies and (b) modify applications to preserve compatibility, it is no wonder that in practice, confinement techniques are narrowly targeted at a small set of highly exposed applications. This naturally leads attackers to target sandbox escape attacks: if the attacker can deposit a file containing malicious code somewhere on the system, and trick the user into running this file, then this code is likely to execute without confinement (because confinement is being applied to a small, predefined set of applications). Alternatively, the attacker may deposit a malicious data file, and lure the user to open it with a benign application that isn't sandboxed. In either case, the attacker is in control of an unconfined process that is free to carry out its malicious acts.

As a result of these factors, existing defenses only shut out the obvious avenues, while leaving the door open for attacks based on evasion (e.g., Stuxnet [10]), policy/enforcement vulnerabilities (e.g., sandbox escape attacks on Adobe Reader [11], IE [20] and Chrome [7]), or social engineering. Stuxnet [10] is a prime example here: one of its attacks lures users to plug in a malicious USB drive into their computers. The drive then exploits a link vulnerability in Windows Explorer, which causes it to resolve a crafted *lnk* file to load and execute attacker-controlled code in a DLL.

1.2 Approach overview and key features

We present a new approach and system called SPIF, which stands for Secure Provenance-based Integrity Fortification, to achieve OS-wide integrity protection on Microsoft Windows. Unlike previous approaches, SPIF:

- *Requires no manual effort for policy development.*
- *Requires no application or OS modifications,* being able to support all major versions of Windows since Windows XP, and feature-rich, unmodified applications such as MS Office, IE, Chrome, Firefox, Skype, Photoshop, and VLC.
- *Confines all applications,* thereby taking away the motivation for sandbox escape attacks.

SPIF defends against unknown malware attacks targeting integrity[1], including stealthy malware such as Stuxnet and Sandworm [46].

SPIF uses information-flow tracking to track provenance. We define provenance as the origin ("where") of a piece of information. The classical notion of data provenance includes a history of transformations ("how") performed on the data. Although the availability of such information could lead to more sophisticated integrity policies, for simplicity and performance, we don't currently capture this "how" information. Indeed, our implementation classifies origins into just two categories: *benign* and *untrusted*. Effectively, provenance in our implementation corresponds to just 1-bit of data.

Figure 1 summarizes some of the key terms defined in previous works [42, 44] that we reuse in this paper. Below, we summarize the key features of our approach.

1.2.1 Reliable provenance tracking system

Existing provenance tracking systems either develop brand new OSes [50, 9] or instrument OSes [21, 42, 25] to label every subject (process) and object (file) in the system. Developing such a system-wide tracking mechanism can be error-prone and involve substantial engineering challenges. This problem is particularly serious in the context of Windows because its source code is unavailable. SPIF therefore realizes provenance tracking using an existing security mechanism, namely, multi-user protection and discretionary access control (DAC). Unlike Android, which uses a different userid for each app, our design creates one new userid for each existing user. While Android's goal is to isolate different apps, we use DAC to protect benign processes/files from untrusted code/data. (We discuss the alternative of using Windows integrity labels in Section 4.5.)

Term	Explanation
malicious	intentionally violate policy, evade enforcement
untrusted	possibly malicious
benign code	non-malicious but potentially vulnerabilities
benign process	process whose code and inputs are benign, i.e., non-malicious

Figure 1: Key terminology

Files coming from untrusted sources are owned by a "*low-integrity*" user, a new user from the OS perspective.

File download is the most common way to introduce new files. SPIF utilizes the Windows Security Zones [26] information filled by most browsers and email readers to identify file integrity. As these files are used in the system, any subjects and objects derived from these untrusted files will also be labeled as low-integrity.

1.2.2 Robust policy enforcement

Experience with various containment mechanisms such as sandboxie [39], Bufferzone [5] and Dell Protected Workspace [8], as well as the numerous real-world sandbox escape attacks [11, 20, 7], have demonstrated the challenges of building effective new containment mechanisms for malicious code [37]. We therefore resolved to rely on (a) simple policies, and (b) time-tested security mechanisms for sandboxing untrusted code. Specifically, SPIF relies on multi-user protection mechanism for policy enforcement. By relying on a mature protection mechanism that was designed into the OS right from the beginning, and has withstood decades of efforts to find and exploit vulnerabilities, SPIF side-steps the challenges of securely confining malicious code.

To protect overall system integrity, it is necessary to sandbox benign processes as well: otherwise, they may get compromised by reading untrusted data, which may contain exploits. SPIF therefore enforces a policy on benign processes as well. Among other restrictions, this policy prevents benign processes from reading untrusted data. Note that, since benign processes have no incentive to actively subvert or escape defenses, it is unnecessary for this enforcement mechanism to be resilient against adversaries.

1.2.3 Application and OS transparency

Today's OSes do not distinguish users from processes running on behalf of users. Every operation performed by a process owned by user R is considered to be endorsed by user R. Compromising a single user process can therefore compromise all other processes and files owned by that user, and possibly, the entire OS. As a result, the system can be considered secure only if no application is vulnerable or is malicious. This is virtually impossible to ensure.

SPIF embraces the fact that applications will have vulnerabilities, and shifts the responsibility of system integrity

[1]Although SPIF does not focus on confidentiality, note that most malware needs to embed itself into the system in such a manner that it would be invoked automatically. This step requires compromising system integrity, and will be caught by SPIF.

protection to an OS-wide mechanism. Hence SPIF can treat applications as blackboxes, requiring no changes. It can support feature-rich unmodified applications such as Photoshop, Microsoft Office, Adobe Reader, Windows Media Player, Internet Explorer, and Firefox.

1.2.4 Usable policy

One of the design goals of SPIF is to preserve normal desktop user experience. Unprotected systems impose no constraints on interactions between subjects (processes) and objects (files). While this allows maximum compatibility with existing software, malware can exploit this trust to compromise system integrity. Preventing such compromise requires placing some restrictions on the interactions. Simply blocking such interactions can lead to application failures, and hence impact user experience. SPIF comes pre-configured with policies targeted at preserving user experience.

1.2.5 Implementation on Windows

We have implemented SPIF on Windows, supporting XP, 7, 8.1, and 10. Implementing such a system-wide provenance tracking system on closed-source OSes is challenging. We share our experiences and lessons on implementing SPIF on Windows.

Research efforts in developing security defenses have been centered on Unix systems. Prototypes are developed and evaluated on open-source platforms like Linux or BSD to illustrate feasibility and effectiveness. While these open-source platforms simplify prototype development, they do not mirror closed-source OSes like Windows. First, these closed-source OSes are far more popular among end-users. They attract not only application developers, but also malware writers. Second, there is only a limited exposition on the internals of closed-source OSes. Very few researchers are aware of how the mechanisms provided in these OSes can be utilized to build systems that are secure, scalable, and compatible with large applications. For this reason, we believe the design and experience presented in this paper is valuable. To be helpful to a broad audience, we describe SPIF in terms of concepts and features of Unix. We hope this will enable more of the systems community that is rooted in Unix to develop solutions for commercial OSes, where far more vulnerabilities are being exploited in the wild.

2. THREAT MODEL

We assume users of the system are benign. Any benign application invoked by a user will therefore be non-malicious. If a user is untrusted, SPIF can simply treat the user as a low-integrity user and every subject created by that user is of low-integrity.

SPIF assumes that any files received from unknown or untrusted sources will be labeled as low-integrity. This can be achieved by exclusion: Only files from trusted sources like OS distributors, trustworthy developers and vendors are labeled as high-integrity. All files from unverifiable origins (including network and external drives) are labeled as untrusted. As described later, SPIF's labeling of incoming files has been seamlessly coupled with Windows Security Zones, which has been adopted by all recent browsers and email clients. An administrator or a privileged process can upgrade these labels, e.g., after a signature or cryptographic hash verification. We may also permit a benign process to downgrade labels.

SPIF focuses on defending attacks that compromise the system-integrity, i.e., performing unauthorized modifications to the system (such as malware installing itself for auto-starting) or environment that enables the malware to subvert other applications or the OS. Although SPIF can be configured to protect confidentiality of user files, this requires confidentiality policies to be explicitly specified, and hence we did not explore it further in this paper. It should be noted that files containing secrets useful to gain privileges are already protected from reads by normal users. This policy could be further tightened for untrusted subjects.

We assume that benign programs rely on system libraries (i.e., `ntdll.dll` and `kernel32.dll`) to invoke system APIs. SPIF intercepts API calls from the libraries to prevent high-integrity processes from accidentally consuming low-integrity objects. We *do not* make any such assumptions about untrusted code or low-integrity processes, but do assume that OS permission mechanisms are secure. Thus, attacks on the OS kernel are out of scope for this paper.

3. PROVENANCE-BASED SANDBOXING

SPIF relies on DAC for secure tracking of provenance of processes and objects (Section 3.1). Moreover, it *sandboxes all processes,* using provenance to determine whether to use low-integrity sandbox (Section 3.2) or high-integrity sandbox (Section 3.3). A high-integrity subject may choose to run in a low-integrity sandbox so that it can process low-integrity files (Section 3.4). Finally, policy choices to preserve user experience are discussed in Section 3.5.

3.1 Secure provenance tracking

SPIF tracks subject- and object-provenance by re-purposing multi-user support. For each real user R on the system, SPIF creates a low-integrity user R_U to represent untrusted subjects executing on behalf of R. Subjects owned by R are deemed benign, so all untrusted subjects must be owned by R_U. Objects such as files, registry contents, pipes and various IPC objects are untrusted if they are owned by (or are writable by) R_U; otherwise they are considered benign. Objects are labeled as low-integrity automatically when created by low-integrity processes.

3.2 Sandboxing low-integrity subjects

A security mechanism should mediate all possible attack paths. Developing such enforcement mechanisms can be tricky [3], especially when we are seeking a system-wide enforcement solution against stealthy malware. Developers of such malware are experts at finding vulnerabilities in either the sandbox design or the policy, and exploiting them. For this reason, we build our sandbox for low-integrity subjects over time-tested DAC mechanisms. The following policies are enforced by the sandbox on subjects of R_U:

- *Read permission:* By default, R_U is permitted to read every object (file, registry, pipe, etc.) readable by R. This policy can be made more restrictive to achieve some confidentiality objectives, but we have not pursued this avenue currently.
- *Write-permission:* By default, R_U subjects are *not* permitted to write objects that are writable by R. However, SPIF provides a utility library that can instead perform *shadowing* [23] of a file. Shadowing causes the original file F to be copied to a new location where R_U maintains its shadowed files. Henceforth, all accesses by R_U-subjects to access F are transparently redirected to this shadow file.

  By avoiding permission denials, shadowing enables more applications to successfully execute. But this may not

always be desirable, so we describe in Section 3.5 how to decide between denial and shadowing.

- *Object creation:* New object creation is permitted if R has permission to create the same object. R_U owns these new object and high-integrity processes will not be permitted to read them. If R creates an object whose name collides with a low-integrity object, the low-integrity object will be transparently shadowed.
- *Operations on R's subjects:* R_U-subjects are not allowed to interact with R-subjects. These include creating remote threads in or sending messages to R's processes, or communicating with R's processes using shared memory.
- *Other operations.* R_U-subjects are given the same rights as those of R for the following operations: listing directories, executing files, querying registry, renaming low-integrity files inside user directories, and so on. Operations that modify high-integrity file attributes are automatically denied.

All these rights, except that of shadowing, are granted to R_U-subjects by appropriately configuring permissions on objects. On Windows, object permissions are specified using ACLs, which can encode arbitrary number of principals. Moreover, there are separate permissions for object creation versus writing, and permissions can be inherited, e.g., from a directory to files in the directory. These features give SPIF the flexibility needed to implement the above policies.

File shadowing is implemented using a utility library that is loaded by default by low-integrity subjects. All shadow files are created within a specific directory created for this purpose. R_U is given full access permissions for this directory.

3.3 Sandboxing high-integrity subjects

Windows Integrity Mechanism (WIM) enforces no-write-up policy to protect higher-integrity processes from being attacked by lower-integrity processes. However, WIM does not enforce no-read-down. A higher-integrity process can read lower-integrity files and hence get compromised. This is well illustrated by the *Task Scheduler XML Privilege Escalation attack* [17] in Stuxnet, where a user-writable task-file is maliciously modified to allow the execution of arbitrary-commands with system privileges. Hence, it is important to protect benign processes from consuming untrusted objects accidentally.

While policy enforcement against low-integrity processes has to be very secure, policies on high-integrity subjects can be enforced in a more cooperative setting. High-integrity subjects do not have malicious intentions and hence they can be trusted *not* to actively circumvent enforcement mechanisms[2].

In this cooperative setting, it is easy to provide protection—SPIF uses a utility library that operates by intercepting calls to DLLs used for making security-sensitive operations, and changing their behavior so as to prevent attempts by a high-integrity process to open low-integrity objects. In contrast, a non-bypassable approach will have to be implemented in the kernel, and moreover, will need to cope with the fact that the system call API in Windows is not well-documented.

[2]Although benign applications may contain vulnerabilities, exploiting a vulnerability requires providing a malicious input. Recall our assumption that inputs will be conservatively tagged, i.e., any input that isn't from an explicitly trusted source will be marked as untrusted. Since a high-integrity process won't be permitted to read untrusted input, it follows that it won't ever be compromised, and hence won't actively subvert policy enforcement.

Similar to performing file shadowing transparently for low-integrity processes, SPIF intercepts low-level Windows APIs, checks if an object about to be consumed is untrusted, and if so, the API calls returns a failure immediately.

3.4 Transitioning between integrity-levels

Users may wish to use benign applications to process untrusted files. Normally, benign applications will execute within the high-integrity sandbox, and hence won't be able to read untrusted files. To avoid this, they need to preemptively downgrade themselves and run within the low-integrity sandbox. The (policy) decision as to whether to downgrade this way is discussed in Section 3.5.

For a high-integrity process to run a low-integrity program, it needs to change its userid from R to R_U. On Unix, this is performed using `setuid`, but Windows only supports an impersonation mechanism that temporarily changes security identifiers (SIDs) of processes. This is insecure for confining untrusted processes as they can re-acquire privileges. The secure alternative is to change the SID using a system library function `CreateProcessAsUser` to spawn new processes with a specific SID. SPIF uses a Windows utility `RunAs` to perform this transition. `RunAs` behaves like a setuid-wrapper that runs programs as a different user. It also maps the desktop of R_U to the current desktop of R so that the transition to user R_U is seamless.

In the context of information flow based systems, SPIF adopts the early downgrading model, which allows a process to downgrade itself just before executing a program image. When compared to the strict Biba [2] policy, early downgrading is strictly more usable [43]. While dynamic downgrading [12, 42] is more general, it requires changes to the OS [42, 43], whereas early downgrading does not.

3.5 Policies

In the design described above, there were two instances where a policy choice needed to be made: (a) whether to deny a write request, or to apply shadowing, and (b) whether to execute a benign application at low-integrity. Below we describe how these choices are automated in SPIF.

Deny Vs Shadow. Shadowing converts write-denials into successful operations, but this is not always desirable. For instance, if a user attempts to overwrite a benign file H with untrusted data L, it would be preferable to inform the user that the operation failed, instead of creating a shadow. Otherwise, the user will be confused when she opens the file subsequently using a benign application: she finds that it does not have the content of L, and wonders why her data was lost[3].

For this reason, SPIF applies shadowing only to files that users are largely unaware of. This choice is similar to previous systems such as PIP [44] where shadowing is primarily applied to preference files. Specifically, SPIF applies shadowing to files in `%USER PROFILE%\AppData`, `HKEY_CURRENT_USER` and files in all hidden directories.

Sandbox selection for benign applications. If a benign application expects to consume untrusted inputs, then it should be run as a low-integrity process. Otherwise it should be run as a high-integrity process. Thus, to determine the sandbox that should be used, we need to know in advance whether a benign application will open a low-integrity file.

[3]The data is not actually lost: if she used an untrusted application to open the file, then she would see L.

While there is no general way to make this prediction, there are important use cases where it is indeed possible to do so. In particular, users most often run applications by double-clicking on a data file, say F. Windows Explorer will spawn a child process to run the designated handler program for this file. This child process will inherit the high-integrity label from Windows Explorer. However, it is clear that the application is being invoked to open F. Thus, if F is a high-integrity file, then the handler program (usually a benign application), should be executed as a high-integrity process. If F is a low-integrity file, then the only sensible choice is to run the handler as a low-integrity file, or else the application won't execute successfully.

Note that if the handler is a low-integrity application, then there is no choice except to run it within the low-integrity sandbox. Thus, this form of *user intent inference* [44] is necessary only for benign applications.

4. SPIF SYSTEM

4.1 Initial file labeling using security zones

An important requirement for enforcing policies is to label new files according to their provenance. Some files may arrive via means such as external storage media. In such a case, we expect the files to be labeled as untrusted (unless the authenticity and/or integrity of files could be verified using signatures or other means). However, we have not implemented any automated mechanisms to ensure this, given that almost all files arrive via the Internet. To enable tracking of the origin of such files, Windows provides a mechanism called Security Zones. Most web browsers and email clients such as Internet Explorer, Chrome, Firefox, MS Outlook, and Thunderbird assign security zones when downloading files. The origins-to-security zones mapping can be customized. Windows provides a convenient user-interface for users to configure what domains belong to what security zones. Microsoft also provides additional tools for enterprises to manage this configuration across multiple machines with ease.

Windows has used security zone to track provenance, but in an ad-hoc manner. When users run an executable that comes from the Internet, they are prompted to confirm that they really intend to run the executable. Unfortunately, users tire of these prompts, and tend to grant permission without any careful consideration. While some applications such as Office make use of the zone labels to run themselves in protected view, other applications ignore these labels and hence may be compromised by malicious input files. Finally, zone labels can be changed by applications, providing another way for malware to sneak in without being noticed.

SPIF makes the use of security zone information mandatory. SPIF considers files from `URLZONE_INTERNET` and `URLZONE_UNTRUSTED` as low-integrity. Applications *must* run as low-integrity in order to consume these files. Moreover, since SPIF's integrity labels on files cannot be modified, attacks similar to those removing file zone labels are not possible.

4.2 Relabeling

SPIF automatically labels files downloaded from the Internet based on its origin. However, it is possible that high-integrity files are simply hosted on untrusted servers. As long as their integrity can be verified (e.g., using checksum), SPIF would allow users to relabel a low-integrity file as high-integrity. Changing file integrity level requires copying the file from shadow storage to its normal location, while the file ownership is changed from R_U to R. We rely on a trusted application for this purpose, and this program is exempted from the information flow policy. Of course, such an application can be abused: (a) low-integrity programs may attempt to use it, or (b) users may be persuaded, through social engineering, to use this application to modify the label on malware. The first avenue is blocked because low-integrity applications are not permitted to execute this program. The second avenue can be blocked by setting mandatory policies based on file content, e.g., upgrading files only after signature or checksum verification.

4.3 Windows API hooking

Utility libraries used by low- as well as high-integrity processes operate by hooking on Windows APIs. The hooking mechanisms used are bypassable, but the libraries themselves possess the exact same privileges that the process already has. Thus, there is no reason for any process to evade hooking.

Hooking methodology. One way to hook on Windows APIs is to modify DLLs statically. However, Windows protects DLLs from tampering using digital signatures, so we cannot modify them. Instead, SPIF relies on a dynamic binary instrumentation tool *Detours* [30]. Detours works by rewriting in-memory function entry-points with *jumps* to specified wrappers. SPIF builds these wrappers around low-level APIs in `ntdll.dll` to modify API behaviors.

To initiate API-hooking, SPIF injects a SPIF-DLL into every process. Upon injection, the `DLLMain` routine of SPIF-DLL will be invoked, which, in turn, invokes Detours.

SPIF relies on two methods to inject the SPIF-DLL into process memory. The first one is based on `AppInit_DLLs` [29], which is a registry entry used by `user32.dll`. Whenever `user32.dll` is loaded into a process, the DLL paths specified in the registry `AppInit_DLLs` will also be loaded.

A second method is used for a few console-based applications (e.g., the SPEC benchmark) that don't load `user32.dll`. It relies on the ability of processes to create a child process in suspended state (by setting the flag `CREATE_SUSPENDED`). The parent then writes the path of the SPIF-DLL into the memory of the child process, and creates a remote thread to run `LoadLibraryA` with this path as argument. After this step, the parent releases the child from suspension.

We rely on the first method to bootstrap the API interception process. Once the SPIF-DLL has been loaded into a process, the library can ensure that all its descendants are systematically intercepted by making use of the second method. Although our approach may miss some processes started at the early booting stage, most processes (such as the login and Windows Explorer) are intercepted.

API interception. SPIF intercepts mainly the low-level functions in `kernel32.dll` and `ntdll.dll`. Higher-level Windows functions such as `CreateFile(A/W)`[4] rely on a few low-level functions such as `NtCreateFile`, `NtSetInformationFile` and `NtQueryAttributes`. By intercepting these low-level functions, all of the higher-level APIs can be handled. Our experience shows that changes to these lower level functions are very rare[5]. Moreover, some applications such as `cygwin` don't use higher-level Windows APIs, but still rely

[4]Calls ending with "A" are for ASCII arguments, "W" are for wide character string arguments.

[5]We did see new functions in Windows 8.1 that SPIF needed to handle.

API Type	APIs
File	NtCreateFile, NtOpenFile, NtSetInformationFile, NtQueryAttributes, NtQueryAttributesFile, NtQueryDirectoryFile,...
Process	CreateProcess(A/W)
Registry	NtCreateKey, NtOpenKey,NtSetValueKey, NtQueryKey, NtQueryValueKey,...

Figure 2: API functions intercepted by SPIF

on the low-level APIs. By hooking at the lower-level API, SPIF can handle such applications as well.

Figure 2 shows a list of API functions that SPIF intercepts. Note that we intercept a few higher-level functions as they provide more context that enables better policy choices. For example, SPIF intercepts `CreateProcess(A/W)` to check if a high-integrity executable is being passed a low-integrity file argument, and if so, create a low-integrity process.

4.4 Handling Registry

To provide a consistent user-experience when benign applications are used to process high- as well as low-integrity files, shadowing is applied on the registry as well. User settings from a high-integrity application can be read when using that application as a low-integrity process. SPIF handles registry shadowing as follows: if a low-integrity process tries to read a registry, it is first checked from R_U's registry. Only if such a registry-entry does not exist, SPIF reads from the R's registry. Registry writes by low-integrity processes are always directed to R_U's registry.

4.5 Alternative choices for enforcement

SPIF could be designed to use WIM labels instead of userids for provenance tracking and policy enforcement. WIM enforces a no-write-up policy that not only prevents a low-integrity process from writing to high-integrity files, but also to processes. Although WIM does not enforce no-read-down, we can achieve it in a co-operative manner using an utility library, the same way how SPIF achieves it now.

With userids, SPIF gets more flexibility and functionality by using DAC permissions to limit the access of untrusted processes. For instance, files that can be read by low-integrity applications can be fine-tuned using the DAC mechanism. Moreover, SPIF can be easily generalized to support the notion of groups of untrusted applications, each group running with a different userid, and with a different set of restrictions on the files they can read or write. Achieving this kind of flexibility would be difficult if WIM labels were used instead of userids. On the positive side, WIM can provide better protection on desktop/window system related attacks. The transition to lower-integrity is also automatic when a process executes a lower-integrity image, whereas this functionality is currently implemented in our utility library. For added protection, one could combine the two mechanisms — this is a topic of ongoing research.

4.6 Limitations

Our WinAPI interception relies on the `AppInit_DLLs` mechanism, which does not kick in until the first GUI program runs. Furthermore, libraries loaded during the process initialization stage are not intercepted. This means that if a library used by a benign application is somehow replaced by a low-integrity version, a malicious library could be silently loaded into a high-integrity process. Our current defense relies on the inability of untrusted applications to replace a high-integrity file, but subtle attacks may be possible where an application loads a DLL from the current directory *if* the DLL is present, but if the DLL is not found, it starts normally. A better solution is to develop a kernel driver to enforce a no-read-down policy on file loads.

Our prototype does not consider IPC that takes place through COM and Windows messages. COM supports ACL, so it may be easy to handle. Windows messages cannot be protected using userids because any process with a handle to the desktop can send message to any other process on the desktop. This is known as a shatter attack. As a result, an untrusted process can send Windows messages to a benign process. There are two ways to solve the problem: The first method is to apply job control in Windows to prevent untrusted processes from accessing handles of benign processes. By setting the `JOB_OBJECT_ULIMIT_HANDLES` restriction, a process cannot access handles outside of the job. The other method is to run untrusted processes as low WIM integrity processes. WIM already prevents lower integrity processes from sending messages to higher integrity processes.

Our prototype does not support untrusted software whose installation phase needs administrative privileges. If we enforce the no-read-down policy, the installation won't proceed. If we waive it, then malicious software will run without any confinement, and can damage system integrity. Techniques for secure software installation [41] can be applied to solve this problem, but will need to be implemented for Windows.

5. EXPERIMENTAL EVALUATION

In this section, we first discuss the implementation complexity of SPIF, and then proceed to evaluate its performance, functionality, and security.

5.1 Implementation complexity

SPIF consists of 4000 lines of C++ and 1500 lines of header. This small size is a testament to the design choices made in our design. A small code size usually translates to a higher level of assurance about safety and security.

5.2 Performance

All performance evaluation results were obtained on Windows 8.1. (Performance does not vary much across different versions of Windows.) Figure 3 shows that on the CPU-intensive SPEC2006 benchmark, SPIF has negligible overhead. This is to be expected, as the overhead of SPIF will be proportional to the number of intercepted Windows API calls, and SPEC benchmarks make very few of these.

We also evaluated SPIF with Postmark [18], a file I/O intensive benchmark. To better evaluate the system for Windows environment, we tuned the parameters to model

	Base (s)	Benign (%)	Untrusted (%)
401.bzip2	1785.9	-0.33%	0.26%
429.mcf	716.4	-1.69%	-0.96%
433.milc	3314.1	1.15%	-0.53%
445.gobmk	1094.9	0.26%	-0.08%
450.soplex	1108.0	0.58%	2.34%
456.hmmer	2386.2	0.02%	0.13%
458.sjeng	1442.5	-0.25%	0.20%
470.lbm	1203.0	-1.51%	-0.32%
471.omnetpp	750.9	0.96%	1.83%
482.sphinx3	2653.6	-2.55%	-3.45%
Mean Overhead		-0.336%	-0.059%

Figure 3: SPEC2006 ref benchmark

File Size	500B to 5KB			5KB to 300KB			300KB to 3MB		
Operations	Base	Benign	Untrusted	Base	Benign	Untrusted	Base	Benign	Untrusted
Files Created per Second	351.14	-5.02%	-10.45%	68.00	-2.79%	-2.02%	8.00	-1.25%	-1.56%
File Read per Second	350.14	-5.18%	-10.59%	67.64	-3.02%	-2.34%	7.60	-3.95%	-1.97%
File Appended per Second	344.79	-5.19%	-10.58%	67.64	-3.02%	-2.61%	8.00	-2.50%	-2.34%
File Deleted per Second	350.21	-5.17%	-10.57%	67.86	-3.03%	-2.00%	8.00	-1.25%	-2.34%
Total Transaction Time (s)	285.36	6.53%	12.38%	367.29	3.05%	4.58%	308.67	1.27%	-0.62%

Figure 4: Postmark overhead for high and low integrity processes in SPIF

files on a Windows 8.1 system. There were 193475 files on the system. The average file size is 299907 bytes, and the median is a much smaller 5632 bytes. We selected 3 size ranges based on this information: small (500 bytes to 5KB), medium (5KB to 300KB), and large (300KB to 3MB) bytes. Each test creates, reads, writes and deletes files repeatedly for about 5 minutes. We ran the tests multiple times and the average is presented in Figure 4. There are three columns for each file size, showing (a) the base runtime obtained on a system that does not have SPIF, (b) the overhead when the benchmark is run as a high-integrity process, and (c) the overhead when it is run as a low-integrity process. As expected, the system shows higher overhead for small files. This is because there are more frequent file creation and deletion operations that are intercepted by SPIF. For larger files, relatively more time is spent on reads and writes, which are not intercepted by SPIF.

We also benchmarked SPIF with Firefox. Specifically, the time required to load webpages. We used a standard test suite [31] to perform page load tests. We fetched Alexa top 1000 pages locally to eliminate network variances. Figure 5 shows the correlation between unprotected page load time with protected benign Firefox and untrusted Firefox. The overheads for benign Firefox and untrusted Firefox are 3.32% and 3.62% respectively.

5.3 Functionality evaluation

Figure 6 shows a list of unmodified applications that can run successfully at high- and low-integrity in SPIF. We used them to perform basic tasks. These applications span a wide range of categories: document readers, editors, web browsers, email clients, media players, media editors, maps, and communication software.

World-writable files. Some applications intentionally leave some directories and files as writable by everyone. As such, low-integrity processes can also write to these locations. SPIF prevents low-integrity processes from writing into these locations by revoking write permissions from low-integrity users. This is achieved by explicitly denying writes in ACLs.

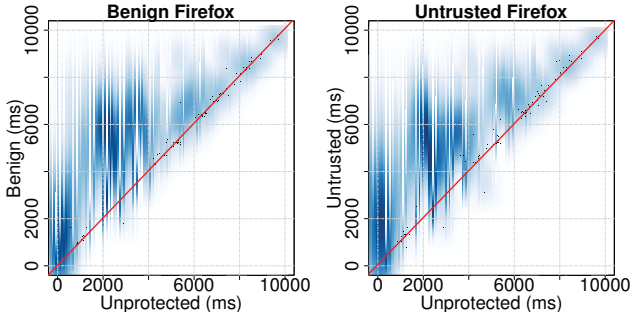

Figure 5: Firefox page load time correlation

Readers	Adobe Reader, MuPDF
Document Processor	MS Office, OpenOffice, Kingsoft Office, Notepad 2, Notepad++, CppCheck, gVim, AklelPad, IniTranslator, KompoZer
Internet	Internet Explorer, Firefox, Chrome, Calavera UpLoader, CCProxy, Skype, Tor + Tor Browser, Thunderbird
Media	Photoshop CC, Picasa, GIMP, WinAmp, Total Video Player, VLC, Picasa, Light Alloy, Windows Media Player, SMPlayer, QuickTime
Other	Virtual Magnifying Class, Database Browser, Google Earth, Celestia

Figure 6: Sample applications supported by SPIF

Some system files are writable by all users, yet they are protected by signatures. SPIF currently does not consider digital signatures as integrity label, and hence we grant benign processes exceptions to read these "untrusted" files. A better approach is to incorporate signatures into integrity label so that no exception has to be granted.

Apart from files, there are also other world-writable resources such as named pipes and devices for system-wide services. SPIF grants exceptions for these resources as none of them can be controlled by low-integrity processes and hence do not carry low-integrity information.

Reading both high and low integrity files. Applications that only read, but not modify files can always start as low-integrity, so that they can consume both high and low integrity files.

Editing both high and low integrity files. SPIF does not allow a process to edit files of different integrity simultaneously as this can compromise the high-integrity files. However, SPIF allows files to be edited in different processes—edit high-integrity files in high-integrity processes, and edit low-integrity files in low-integrity processes. As these processes are running as different users, different instances of the same application can run simultaneously in SPIF.

When it is the users' intention to open low-integrity files, SPIF opens the files with low-integrity processes. However, when users do not expect opening the low-integrity files, such openings would be denied. SPIF considers user-actions such as double-clicking on the files, selecting files from a file-dialog box, or explicitly typing the file names as indications of their intents. When intent is inferred in this manner, SPIF runs the applications as low-integrity. SPIF currently captures such intents via user interaction with the Windows Explorer: when users double-clicked to open a file, Windows Explorer will execute the handler programs with the file-path as an argument. When the file-path corresponds to a low-integrity file, SPIF considers this as a user-consent for starting a program as a low-integrity process.

Low-integrity processes writing high-integrity files. Applications like OpenOffice maintain runtime information in user profile directories. Applications expect these files to be

CVE/OSVDB-ID	Application	Attack Vector
2014-0568	Adobe Reader	Code
CVE-2010-2568	Windows Explorer (Stuxnet)	Data (lnk)
2014-4114/113140	Windows (Sandworm)	Data (ppsx)
104141	Calavera UpLoader	Preference (dat)
100619	Total Video Player	Preference (ini)
2013-6874/100346	Light Alloy	Data (m3u)
2013-3934	Kingsoft Office Writer	Data (wps)
102205	CCProxy	Preference (ini)
2013-4694/94740	WinAmp	Preference (ini)
2014-2013/102340	MuPDF	Data (xps)

Figure 7: Exploits defended by SPIF

both readable and writable— otherwise they will simply fail to start and crash. Having these files as high-integrity would prevent low-integrity processes from being usable. Letting these files become low-integrity would break availability of high-integrity processes.

SPIF shadows accesses to these files inside user-profile directories, hence high- and low-integrity processes can both run without significant usability issues. One problem is that profiles for high and low integrity sessions are isolated. There is no safe way to automatically merge the shadowed files together.

5.4 Security evaluation

We evaluated the security of SPIF against malware from Exploit-DB [34] on Windows XP, 7 and 8.1. We selected all local exploits targeting Windows platform, mostly released between January and October of 2014. Since these exploits work on specific versions of software, we only included malware that "worked" on our testbed, and their results were easy to verify. Figure 7 summarizes the CVE/OSVDB-ID, vulnerable applications, and the attack vectors. We classify attacks into three types: data input attacks, preference/configuration file attacks, and code attacks.

Note that by design, SPIF protects high-integrity processes against all these attacks. Since high-integrity processes cannot open low-integrity files, only low-integrity applications can input any of the malware-related files. In other words, *attackers can only compromise low-integrity processes.* Moreover, there is no mechanism for low-integrity processes to "escalate their privilege" to become high-integrity processes. Note that since low-integrity processes can only modify files within the shadow directory, they cannot affect any user or system files. For this reason, SPIF *stopped all of the attacks shown in Figure 7.*

Both data and preference/configuration file attacks concern inputs to applications. When applications fail to sanitize malicious inputs, attackers can exploit vulnerabilities and take control of the applications. Data input attacks involve day-to-day files like documents (e.g., wps, ppsx, xps). They can be exploited by simply tricking users to open files. On the other hand, attacks using preference/configuration files are typically hidden from users, and are trickier to exploit directly. These exploits are often chained together with code attacks to carry out multi-steps attacks to circumvent sandboxes.

Code attacks correspond to instances where the attacker is already able to execute code but with limited privileges, e.g., inside a restrictive sandbox. For instance, in the Adobe Reader exploit [11], it is assumed that an attacker has already compromised the sandboxed worker process. Although attackers cannot run code outside of the sandbox, they can exploit a vulnerability in the broker process. Specifically, the attack exploited the worker-broker IPC interface — the broker process only enforced policies by resolving the first level NTFS junction. A compromised worker can use a chain of junctions to bypass the sandbox policy and write arbitrary file to the file system with the broker permissions. Since the broker ran with user privilege, attackers could therefore escape the sandbox and modify any user files. SPIF ran both the broker and worker as untrusted processes. As a result, the attack could only create or modify low-integrity files, which means that any subsequent uses of these files were also confined by the low-integrity sandbox.

SPIF stopped Stuxnet [10] by preventing the lnk vulnerability from being triggered. Since the lnk file is of low-integrity, SPIF prevented Windows Explorer from loading it, and hence stopped Windows Explorer from loading any untrusted DLLs.

We also tested the Microsoft Windows OLE Package Manager Code Execution vulnerability, called Sandworm [46]. It was exploited in the wild in October 2014. When users view a malicious PowerPoint file, OLE package manager can be exploited to modify a registry in `HKLM`, which subsequently triggers a payload to run as system-administrator. SPIF ran PowerPoint as low-integrity when it opened the untrusted file. The exploit was stopped as the low-integrity process does not have permissions to modify the system registry.

The most common technique used to exploit the remaining applications was an SEH buffer overflow. The upload preference file `uploadpref.dat` of Calavera UpLoader and `Setting.ini` of Total Video Player were modified so that when the applications ran, the shell-code specified in the files would be executed. Similarly, SEH buffer overflow can also be triggered via data input, e.g., using a multimedia playlist (.m3u) for Light Alloy or a word document (.wps) for Kingsoft Office Writer. Other common techniques include integer overflow (used in `CCProxy.ini` for CCProxy) and stack overflow (triggered when MuPDF parsed a crafted xps file or when WinAmp parsed a directory name with invalid length). In the absence of SPIF, these applications ran with user's privileges, and hence the attackers could abuse user's privileges, e.g., to make the malware run persistently across reboots.

Although preference files are specific to applications, there exists no permission control to prevent other applications from modifying them. SPIF makes sure that preference files of high-integrity applications cannot be modified by any low-integrity subject. This protects benign processes from being exploited, and hence attackers cannot abuse user privileges. On the other hand, SPIF does not prevent low-integrity instances of the applications from consuming low-integrity preference or data files. While attackers could exploit low-integrity processes, they only had privileges of the low-integrity user. Furthermore, all attackers' actions were tracked and confined by the low-integrity sandbox.

6. RELATED WORK

The first step in most malware attacks is an exploit, typically targeting a memory corruption vulnerability to gain arbitrary execution capability. Widespread deployment of ASLR and DEP have raised the bar, but in the end, attackers always seem to be able to bypass these defenses. Comprehensive memory corruption defenses [48, 33] can stop these

exploits, but they introduce some incompatibilities in large and complex software. Light-weight bounds-checking [15] avoids this problem by trading off off some protection for increased compatibility and performance.

Instead of focusing on the exploit mechanism, most malware defenses target the payload execution phase. The payload may be an exploit payload, or it may refer to installed malware. These defenses can be partitioned into several categories discussed below.

6.1 Sandboxing and Isolation

Various sandboxing techniques [13, 35, 45, 24, 49] have been discussed earlier in the paper. A central challenge here is policy development: how to identify a policy that effectively blocks attacks without unduly degrading functionality. Although some techniques (e.g., model-carrying code [40]) have been devised to ease application-specific policy development, they require some level of trust on the software. If one suspects that it could be truly malicious, then a secure policy will likely preclude all access, thus causing the application to fail.

Full isolation is a more realistic alternative for software that could be malicious. Android apps, by default, are fully isolated from each other, thereby preventing one malicious app from compromising another. This approach is so popular that vendors back-ported the idea to recent desktop OSes (Windows 8 and Mac OS X). Unfortunately, full isolation means that no data can be shared. As a result, an untrusted application cannot be used to view or process user files that may have been created by another application. This difficulty can be solved using the concept of *one-way isolation* [22], which allows untrusted applications to read user files but not overwrite them. The idea of shadowing files was proposed in that work to permit untrusted applications to run safely without experiencing any security failures.

In practice, full isolation proves to be too restrictive, so mobile OSes such as Android permit apps to communicate with each other, or with system applications, using well-defined interfaces. Unfortunately, the moment such interactions take place, security can no longer be guaranteed: if a benign process receives and processes a request or data from an untrusted process, it is entirely up to the benign process to protect itself from damage due to this interaction. It is in this context that information-flow based techniques such as Spif help: by keeping track of the provenance of input, Spif can either prevent a benign process from consuming the input, or downgrade itself into a low-integrity process before consuming it.

6.2 Information flow techniques

These techniques label every object and subject with an integrity (and/or confidentiality) label, and globally track their propagation. The earliest works in this are date back to the 1970s, and rely on centralized IFC, where the labels are global to the system. In contrast, some recent efforts have focused on decentralized IFC (DIFC) [50, 9, 19], which allows any principal to create new labels. This flexibility comes with the responsibility to make nontrivial changes to application and/or OS code. Since backward compatibility with existing code is a high priority for Spif, we have not pursued a DIFC model.

Several recent works [42, 25, 21, 44] focused on making IFC work on contemporary OSes, specifically Linux. Of these PPI [42] specifically targeted the same problem as us, namely, integrity protection for desktop systems against malware and exploits. Unlike Spif, PPI relies on kernel modifications (implemented using LSM hooks) for label propagation as well as policy enforcement. While such an approach provides more flexibility and hence supports a wider range of policies, its downside is that it is difficult to port to other OSes. In contrast, PIP [44] avoids OS changes and is hence most closely related to Spif. Like Spif, PIP also re-purposes multi-user protection for information-flow tracking. But its design, targeted at Unix, necessarily differs from Spif that targets Windows. Spif can take advantage of mechanisms specific in Windows (such as ACLs and WIM) to remove the need of helper process or the need for a separate display server. Moreover, Spif's design provides a greater degree of portability across different OS versions, and a higher-level of application compatibility, having been applied to a much larger range of complex, feature-rich applications. Spif's integration with the security zone in Windows also provides a better end-to-end protection.

6.3 Provenance

Data provenance has become an important consideration in many domains, including scientific computing, law and health care. In these domains, provenance captures not only the origin of date ("where"), but also how it was generated [6]. Securing data provenance [16] is an important concern in many domains. Some recent efforts have incorporated secure provenance tracking into OSes, e.g., Linux [1].

Other works in security have been focused on (security) applications of provenance. Reference [47] associates every network packet with a keystroke event. These keystroke events serve as provenance labels of a packet. This enables the detection of malware-generated network packets that won't have these provenance labels. Reference [32] uses provenance-tracking to correlate malicious network traffic to the application that generated it. Spif combines the idea of provenance and information flow tracking to protect system integrity against unknown malware.

7. CONCLUSION

In this paper, we presented Spif, a comprehensive system for integrity protection on Windows that is based on system-wide provenance tracking. Unlike existing malware defenses, which are reactive in nature, Spif is pro-active, and hence works against unknown and stealthy malware. We described the design of Spif, detailed its security features, and features designed to preserve application usability. Our experimental results show that Spif imposes low performance overheads, almost negligible on many benchmarks. It works on many versions of Windows, and is compatible with a wide range of feature-rich software, including all popular browsers and Office software, media players, and so on. We evaluated it against several malware samples from Exploit Database [34], and showed that it can stop a variety of highly stealthy malware.

We certainly don't claim at this point that our prototype is free of vulnerabilities, or that it can stand up to targeted attacks. But we do believe that any such weaknesses are the result of limited resources expended so far on its implementation, and are not fundamental to its design. Hardening it to withstand targeted, real-world malware attacks will require substantial additional engineering work, but we do believe that Spif represents a promising new direction for principled malware defense. An open-

source implementation of our system is available from http://seclab.cs.stonybrook.edu/download.

8. REFERENCES

[1] Bates, A., Tian, D. J., Butler, K. R., and Moyer, T. Trustworthy Whole-System Provenance for the Linux Kernel. In *USENIX Security* (2015).

[2] Biba, K. J. Integrity Considerations for Secure Computer Systems. In *Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Hanscom Air Force Base, Bedford, Massachusetts* (1977).

[3] Brian Gorenc, J. S. Thinking outside the sandbox - Violating trust boundaries in uncommon ways. In *BlackHat* (2014).

[4] Brumley, D., and Song, D. Privtrans: Automatically Partitioning Programs for Privilege Separation. In *USENIX Security* (2004).

[5] BufferZone Security Ltd. BufferZone, http://bufferzonesecurity.com/.

[6] Buneman, P., Khanna, S., and Tan, W. C. Why and Where: A Characterization of Data Provenance. In *ICDT* (2001).

[7] Constantin, L. Researchers hack Internet Explorer 11 and Chrome at Mobile Pwn2Own. http://www.pcworld.com/article/2063560/researchers-hack-internet-explorer-11-and-chrome-at-mobile-pwn2own.html/.

[8] Dell. Dell Data Protection | Protected Workspace. http://www.dell.com/learn/us/en/04/videos~en/documents~data-protection-workspace.aspx.

[9] Efstathopoulos, P., Krohn, M., VanDeBogart, S., Frey, C., Ziegler, D., Kohler, E., Mazières, D., Kaashoek, F., and Morris, R. Labels and Event Processes in the Asbestos Operating System. In *SOSP* (2005).

[10] Falliere, N., Murchu, L., and Chien, E. W32. Stuxnet Dossier. *White paper, Symantec Corp., Security Response* (2011).

[11] Fisher, D. Sandbox Escape Bug in Adobe Reader Disclosed. http://threatpost.com/sandbox-escape-bug-in-adobe-reader-disclosed/109637.

[12] Fraser, T. LOMAC: Low Water-Mark Integrity Protection for COTS Environments. In *S&P* (2000).

[13] Goldberg, I., Wagner, D., Thomas, R., and Brewer, E. A. A Secure Environment for Untrusted Helper Applications (Confining the Wily Hacker). In *USENIX Security* (1996).

[14] Google Security Research. Windows Acrobat Reader 11 Sandbox Escape in MoveFileEx IPC Hook. https://code.google.com/p/google-security-research/issues/detail?id=103.

[15] Hasabnis, N., Misra, A., and Sekar, R. Light-weight bounds checking. In *CGO* (2012).

[16] Hasan, R., Sion, R., and Winslett, M. Introducing Secure Provenance: Problems and Challenges. In *StorageSS* (2007).

[17] jduck. CVE-2010-3338 Windows Escalate Task Scheduler XML Privilege Escalation | Rapid7. http://www.rapid7.com/db/modules/exploit/windows/local/ms10_092_schelevator.

[18] Katcher, J. Postmark: A new file system benchmark. Technical Report TR3022, Network Appliance, 1997.

[19] Krohn, M., Yip, A., Brodsky, M., Cliffer, N., Kaashoek, M. F., Kohler, E., and Morris, R. Information Flow Control for Standard OS Abstractions. In *SOSP* (2007).

[20] Li, H. CVE-2015-0016: Escaping the Internet Explorer Sandbox. http://blog.trendmicro.com/trendlabs-security-intelligence/cve-2015-0016-escaping-the-internet-explorer-sandbox.

[21] Li, N., Mao, Z., and Chen, H. Usable Mandatory Integrity Protection for Operating Systems . In *S&P* (2007).

[22] Liang, Z., Sun, W., Venkatakrishnan, V. N., and Sekar, R. Alcatraz: An Isolated Environment for Experimenting with Untrusted Software. In *TISSEC* (2009).

[23] Liang, Z., Venkatakrishnan, V., and Sekar, R. Isolated program execution: An application transparent approach for executing untrusted programs. In *ACSAC* (2003).

[24] Loscocco, P., and Smalley, S. Meeting Critical Security Objectives with Security-Enhanced Linux. In *Ottawa Linux Symposium* (2001).

[25] Mao, Z., Li, N., Chen, H., and Jiang, X. Combining Discretionary Policy with Mandatory Information Flow in Operating Systems. In *TISSEC* (2011).

[26] Microsoft. URL Security Zones (Windows) - MSDN - Microsoft. https://msdn.microsoft.com/en-us/library/ie/ms537021%28v=vs.85%29.aspx.

[27] Microsoft. What is Protected View? - Office Support. https://support.office.com/en-au/article/What-is-Protected-View-d6f09ac7-e6b9-4495-8e43-2bbcdbcb6653.

[28] Microsoft. What is the Windows Integrity Mechanism? https://msdn.microsoft.com/en-us/library/bb625957.aspx.

[29] Microsoft. Working with the AppInit_DLLs registry value. http://support.microsoft.com/kb/197571.

[30] Microsoft Research. Detours. http://research.microsoft.com/en-us/projects/detours/.

[31] Mozilla. Buildbot/Talos/Tests. https://wiki.mozilla.org/Buildbot/Talos/Tests.

[32] Nadji, Y., Giffin, J., and Traynor, P. Automated Remote Repair for Mobile Malware. In *ACSAC* (2011).

[33] Nagarakatte, S., Zhao, J., Martin, M. M., and Zdancewic, S. SoftBound: SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *PLDI* (2009).

[34] Offensive Security. Exploits Database, http://www.exploit-db.com/.

[35] Provos, N. Improving Host Security with System Call Policies. In *USENIX Security* (2003).

[36] Provos, N., Markus, F., and Peter, H. Preventing Privilege Escalation. In *USENIX Security* (2003).

[37] Rahul Kashyap, R. W. Application Sandboxes: A Pen-Tester's Perspective. http://labs.bromium.com/2013/07/23/application-sandboxes-a-pen-testers-perspective/.

[38] Reis, C., and Gribble, S. D. Isolating Web Programs in Modern Browser Architectures. In *EuroSys* (2009).

[39] Sandboxie Holdings, LLC. Sandboxie, http://www.sandboxie.com/.

[40] Sekar, R., Venkatakrishnan, V., Basu, S., Bhatkar, S., and DuVarney, D. C. Model-Carrying Code: A Practical Approach for Safe Execution of Untrusted Applications. In *SOSP* (2003).

[41] Sun, W., Sekar, R., Liang, Z., and Venkatakrishnan, V. N. Expanding Malware Defense by Securing Software Installations. In *DIMVA* (2008).

[42] Sun, W., Sekar, R., Poothia, G., and Karandikar, T. Practical Proactive Integrity Preservation: A Basis for Malware Defense. In *S&P* (2008).

[43] Sze, W. K., Mital, B., and Sekar, R. Towards More Usable Information Flow Policies for Contemporary Operating Systems. In *SACMAT* (2014).

[44] Sze, W. K., and Sekar, R. A Portable User-Level Approach for System-wide Integrity Protection. In *ACSAC* (2013).

[45] Ubuntu. AppArmor. https://wiki.ubuntu.com/AppArmor/.

[46] Ward, S. iSIGHT discovers zero-day vulnerability CVE-2014-4114 used in Russian cyber-espionage campaign. http://www.isightpartners.com/2014/10/cve-2014-4114/.

[47] Xu, K., Xiong, H., Wu, C., Stefan, D., and Yao, D. Data-Provenance Verification For Secure Hosts. In *TDSC* (2012).

[48] Xu, W., DuVarney, D. C., and Sekar, R. An efficient and backwards-compatible transformation to ensure memory safety of C programs. In *FSE* (2004).

[49] Yee, B., Sehr, D., Dardyk, G., Chen, J. B., Muth, R., Orm, T., Okasaka, S., Narula, N., Fullagar, N., and Inc, G. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *S&P* (2009).

[50] Zeldovich, N., Boyd-Wickizer, S., Kohler, E., and Mazières, D. Making Information Flow Explicit in HiStar. In *OSDI* (2006).