

Detecting Infection Onset With Behavior-Based Policies

Kui Xu Danfeng (Daphne) Yao
Department of Computer Science
Virginia Tech
Email: {xmenxk, danfeng}@cs.vt.edu

Qiang Ma Alexander Crowell
Department of Computer Science
Rutgers University
Email: {qma, acrowell}@cs.rutgers.edu

Abstract—A major vector of computer infection is through exploiting vulnerable software or design flaws in networked applications such as the browser. Malicious code can be fetched and executed on a victim’s machine without the user’s permission, as in drive-by download (DBD) attacks. In this paper, we describe a new tool called *DeWare* (standing for Detection of Malware) for detecting the onset of infection delivered through vulnerable applications. *DeWare* enforces the dependencies between user actions and system events, such as file-system access and process execution. Our tool can be used to provide real time protection of a personal computer, as well as for diagnosing and evaluating untrusted websites for forensic purposes. Our solution demonstrates a usable host-based framework for controlling and enforcing the access of system resources.

We perform extensive experimental evaluation, including a user study with 21 participants, thousands of legitimate websites (for testing false alarms), 84 malicious websites in the wild, as well as lab reproduced exploits. Our results show that *DeWare* is able to correctly distinguish legitimate download events from unauthorized system events with a low false positive rate ($< 1\%$).

I. Introduction

Malicious software (malware) downloaded from the Internet has been the leading infection vector [5]. Malware may be delivered stealthily through a networked application such as a browser, a peer-to-peer file sharing client, or a chat application. Web browser is the most common vehicle for a host to contract malware. 10% of the websites were found to contain drive-by-download exploits [21]. Drive-by-download (DBD) attacks exploit software or design vulnerabilities in a browser or its external components, and stealthily fetch executables from remote malware-hosting server without any user permission. Botnets often use DBD as the initial infection vector, e.g., Torpig [33]. Other networked applications besides the browser may be vulnerable to drive-by-download attacks. For example, a proof-of-concept Quicktime-based drive-by-download attack has been demonstrated in Second Life [22]. Unauthorized web download and process creation are among the symptoms of malicious bots studied by Morales *et al.* in [18].

Conventional signature-based techniques may not be effective against zero-day exploits or code with sophisticated obfuscation. In comparison, host-based detection approaches

are much more feasible against drive-by download attacks and the onset of infection in general. In this paper, we present such a host-based monitoring framework that enforces policies to control how processes are created and file systems are accessed. We provide a tool named *DeWare* standing for detection of malware, that guards a personal computer by detecting signs of malware infection, specifically at the onset of infection. Our solution can be used to detect any type of drive-by downloads, including the browser-based exploits.

DeWare is a host-based security tool for detecting the onset of malware infection through the novel use of rules that enforce the correct dependency characteristics in operating system. The dependency in our context refers to the relation between user behaviors and corresponding system events. *DeWare* is capable of performing host-wide monitoring beyond the browser. Its detection is based on observing stealthy download-and-execution pattern, which is a behavior many active malware exhibits at its onset, including the recent Hydraq malware [3].

Among the challenges that we address in this paper, one is how to distinguish drive-by downloads from legitimate downloads caused by human users. Our solution is to strategically monitor file-system events and analyze them with observed user actions. We aim to identify the dependency between system events and user activities. Our knowledge about user behaviors is further refined with additional application-specific information. Another issue is how to efficiently handle the *voluminous* application-triggered benign downloads, which are not directly caused by user actions. We refer to these downloads as *indirectly* caused by the user’s action. Our solution is designing an access control component and policies within the operating system that enforce and regulate behaviors of applications in terms of accessing system resources. These mechanisms enable us to largely reduce our file-system monitoring scope, reduce false alarms, and provide comprehensive surveillance across the host.

Our contributions We summarize our technical contributions as follows.

- We present our design and implementation of a tool called *DeWare* that detects the onset of infection including drive-by downloads. *DeWare* identifies the infection onset by detecting download-and-execute patterns that most stealthy malware exhibits. Our work uniquely

incorporates user-behavior characteristics in generating and evaluating policies used for detecting attacks. Our behavior-based monitoring approach is general, and is useful beyond the specific drive-by download problem studied.

- We define rules for identifying dependency between the system events and the user actions that initiate them. Finding causal relations among multiple sets of events typically requires sophisticated statistical inference techniques [1], [8]. In operating system, however, system events are artificially created in response to user actions. We demonstrate that simple-yet-effective rules can be defined for enforcing the correct system characteristics and securing a host. Our approach can be generalized to other paradigms of security such as enforcing dependencies among user inputs and outbound traffic.
- We perform extensive experiments including a user study with 21 participants to evaluate DeWare’s ability to detect DBD exploits both in a lab setting and 84 malicious websites in the wild; the ability to accurately identify legitimate user downloads with low false positives (<1%). DeWare generates zero false alarm when being automatically tested on 2000 legitimate websites.

The rest of the paper is organized as follows. We give our models in Section II. An overview of our design of DeWare is in Section III. Our *DeWare* implementation in Windows operating system is presented in Section IV. Experimental evaluation is described in Section V. Related work is compared in Section VI, and conclusions are given in Section VII.

II. Models And Definitions

We assume that the browser and its components are not secure and may have software vulnerabilities. The operating system is assumed trustworthy and secure, and thus the kernel-level monitoring of system events and user inputs yields trusted information. The integrity of file systems defined in our model refers to the enforcement of user-intended or user-authorized file-system activities; the detection and prevention of malware-initiated tampering.

Our work targets *application-level malware*. We assume that malware makes persistent changes to the target host’s file system or attempts to execute its code by creating a new process, which are commonly observed behaviors at the onset of infection. Similar assumptions were also used in Strider HoneyMonkeys [37].

We do not consider social engineering attacks in this paper. For example, a user may be tricked to click on Web links or links in email messages that result in the download of malicious executables. (The recent Hydraq malware starts with a personalized spam message with an embedded link to attacker’s website [3].)

In our model, we define two types of events: *user action* and *system event*. User actions include keyboard inputs and mouse clicks. System events are any kernel-level transactions such as file system events, network events (outbound and

incoming), process events. In this paper, for detecting drive-by download we focus on two system events, namely file creation and process creation.

DeWare runs at the background. Whenever Alice starts an application such as the browser, all her actions and corresponding system events are monitored. A normal system event should be correlated to Alice’s inputs. When the browser is compromised by a maliciously manipulated website, the stealthy events such as executable downloading and execution are detected.

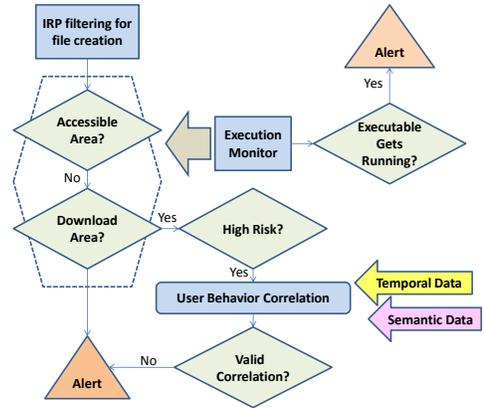


Fig. 1. Schematic drawing of the work flow for detecting the infection onset on a host based on analyzing user-behaviors and file system/process properties.

III. DeWare Design and Its Components

One technical challenge is the fact that many applications (such as the browser) automatically fetch and create files that make persistent changes to the file systems. Our solution is to enforce fine-grained access policies on applications to control and confine their access to the file systems. A temporary file automatically downloaded by an application is considered legitimate *only if* it appears in specified areas and is not executed. Violations of the policies are likely caused by malware infection and are reported.

Users’ intention is embodied by their mouse and keyboard inputs, information of which such as timestamp, corresponding PIDs, and content can be logged at the kernel level or at the application level. Both approaches have different security assumptions and yield user-input information with different granularity. There are pros and cons associated with both methods, which are discussed more in Section III-C. Given the observed user activities and system events, we correlate the two data streams, according to rules on their attributes, e.g., empirically-defined time intervals and process IDs. In our security model, the user inputs obtained are trustworthy, i.e., they are not forged by malware. This assumption can be eliminated if hardware-based attestation (e.g., with trusted platform module) is enabled as described in [32].

DeWare includes three main components: *file-system access monitor*, *input recorder*, and *execution monitor*. The work flow

in DeWare is shown in Figure 1.

- A file-system access monitor is for controlling the access of processes to write to the file system. The policies are defined and enforced through intercepting file-creation related system calls.
- An input recorder is for collecting and interpreting user behavior, in particular user actions regarding downloading files via the browser. We investigate the recording of user actions at both kernel-level and browser-level.
- An execution monitor is for inspecting a portion of the file system for illegal process creation. This component runs with pre-defined policies; it together with the file-system access monitor ensures unauthorized download-and-execute cases are detected at real-time.

A. File-System Monitor: Confining Applications' Downloads

Content automatically downloaded by applications is benign, provided that the application is not comprised. Because our security model does not assume the trustworthiness of applications, it is necessary to examine *all* downloads on the host. However, capturing all file-creation events related to all processes generates an overwhelmingly large number of records. For example, in a study that we performed, a user indirectly triggers 482 file creations in Temporary Internet Files folder and 47 in Cookies directory within 30 minutes. Files created in those folders are usually benign.

We design and implement a framework that allows us to specify policies to limit applications' access to portion of the file system. The access control framework is to reduce white noise due to application triggered downloads. We define the *accessible area* for an application in the file systems.

Definition 3.1: Accessible area of an application is a set of pre-defined directories on the file system where the application is allowed to write to files. The application is not allowed to create or write to files outside its accessible area.

For example, Temporary Internet Files folder is modifiable by Internet Explorer, whereas system folders are not. The directories to include in the accessible area can be learned based on the software's behaviors, which significantly reduces the area to monitor for file creation, and the number of events to record. With this file-system monitor, malware is confined to the accessible area. Moving executables out of the accessible area is also forbidden.

User download does not need to follow the same requirements. However, the user is not allowed to download files anywhere on the file system, but to a specified *downloadable area*, which is described in Section III-E. With the aforementioned file-system monitor, an attacker may download malware executables to the accessible area of a rogue application, e.g., to the temp folder via a compromised browser. Thus, file-system monitoring alone is not sufficient for detecting drive-by download. We solve the problem by monitoring execution patterns in these download areas, which is presented in the next section.

B. Real-Time Execution Monitoring on Host

Although dormant malware (downloaded but not executed) does not pose immediate threat to the host, it may get executed later on. In our experiments with real-world malware, some malware starts itself after a reboot. In DeWare, execution monitor is to prevent malware stored at accessible area from being run. Execution monitor inspects the accessible areas granted to applications.

For a newly created process, we collect information including process ID, image file name, parent process ID, and timestamp. Once a new process is created, the execution monitor is notified. It verifies that the image file is not in the accessible area of any application. As our security model assumes that the kernel is not compromised, the process information collected is complete and trustworthy. We do not consider the existence of rootkits that hide their presence in the process table. As we aim to detect the onset of the infection on an otherwise clean host, this assumption is valid.

C. Input Recorder

There are several approaches for obtaining user intention on a computer: *i)* directly asking the user (e.g., through a pop-up window) [38], *ii)* analyzing user's transaction history and extracting patterns [25], or *iii)* recording the events entered through keyboard or mouse devices [32]. Each method has pros and cons. For example, pop-up window may be intrusive to user, but is easy to implement. In DeWare, we opt for the third approach.

There are two levels of user-input monitoring: kernel level and application level. Kernel-level input logging records user inputs at their origin, as it is to collect events triggered by activities on external input devices via device drivers. It intercepts information about user inputs – coordinates, timestamp, and content – by placing listeners or hooks in the kernel. The inputs are consumed by the current foreground process, whose ID can be identified. Kernel-level input logging is application-independent, which makes the method general.

One can also record user inputs from the application. This approach requires writing a plug-in specific to the application. It also assumes the trustworthiness of the extension to ensure the data integrity. The obvious advantage of application-level input logging is the ability to *semantically* interpret certain input events, such as the URL that a mouse clicks on. This URL is application-specific data and cannot be easily obtained outside the browser. At the kernel level, a user's mouse click is not associated with this semantic information. In our prototype in Windows, we realize and compare both methods.

D. Dependency Rules

User event such as a left mouse click has specific *semantic* meanings in the application that consumes the event. More specifically in the context of download event, semantic properties of a mouse-click event include the information about the file that the user intends to download, for example, source URL, file name, type, size, destination directory, as well as temporal information (i.e., timestamps of input events), and

properties of the process consuming the input event. Therefore, our inference rules are defined to inspect and compare the file, process, and temporal properties between user actions and system events.

We formalize our dependency rules as follows. Consider a file-creation system event e with attributes $(fname, fpath, ftype, T, pid, pname)$, where $fname$, $fpath$, $ftype$ are the name, path, and type of the file, respectively, T is the corresponding timestamp, pid and $pname$ are the process ID and name of the application that creates the file, and consider an user-download event e^* with attributes $(fname^*, fpath^*, ftype^*, T^*, pid^*, pname^*, intended_dl_URL, actual_dl_URL)$, where $fname^*$, $fpath^*$, and $ftype^*$ are the name, destination path, type of the file obtained through parsing the user event, T^* is the corresponding timestamp, pid^* and $pname^*$ are the process ID and name of the application that consumes the user event. $intended_dl_URL$ is the URL where user is attempting to download a file, while $actual_dl_URL$ is where the application actually makes the download.

We define the dependency between events e and e^* (i.e., file-creation event e is caused by a user mouse-click event e^*) as the satisfaction of the following rules.

- *Rule 1:* File properties of events match: $fname = fname^*$, $fpath = fpath^*$, and $ftype = ftype^*$. That is, the file to be created should match the file that a user intends to download.
- *Rule 2:* Process properties of events match: $pid = pid^*$ and $pname = pname^*$. Both events should be for the same application.
- *Rule 3:* Temporal constraint is satisfied: $0 < T - T^* \leq \tau$. That is, a legitimate file-creation event is required to take place within a short threshold τ after a valid user-input event. Different τ values are experimentally evaluated in our user study in Section V.
- *Rule 4:* User intention is ensured: $intended_dl_URL = actual_dl_URL$. User’s download intention should be correctly reflected by application’s action. For example, a file creation of iTunesSetup.exe to the system can be correlated back to the events of *i)* user clicks to initiate and confirm the download from <http://appldnld.apple.com.edgesuite.net> and *ii)* the browser goes to fetch the executable form <http://appldnld.apple.com.edgesuite.net/.../iTunesSetup.exe>. It means that file should be downloaded from the source URL that user intends.

Additional rules can be introduced to capture and specify contextual properties of the download and file-creation events. We note that temporal-constraint alone does not provide sufficient security protection against infection onset. Consider coincidental malware download as follows.

Definition 3.2: We define *coincidental download* as the download of malware that coincidentally occurs immediately after a user’s mouse click, more specifically the time interval between the download event and its immediately preceding mouse-click event is within threshold τ .

The problem of coincidental download only exists when logging user’s mouse events at the kernel level, which lacks semantic and contextual information of the events. For example, a click on a URL cannot be distinguished from a click on download-dialogue box. This problem is prevented when using application-level input logging as described in Section III-C.

E. Piggybacking Download and its Detection

Piggyback download defined in [31] is where malware is part of a piece of software downloaded with proper user permission, e.g., spyware bundled with compression software. Preventing piggybacking code from being downloaded to the file system is difficult in our model. Thus, our approach is to detect it when the downloaded malware is being executed. The area that execution monitor inspects needs to be expanded beyond the accessible areas of applications. The execution monitor also inspects the area at which user-authorized downloads are stored, and seeks confirmation from the user if files in that area get executed. We define downloadable area in Definition 3.3 below.

Definition 3.3: Downloadable area of a user is a set of pre-defined directories on the file system to which the user downloads files from the Internet.

If piggybacking download in this area creates new processes, the execution monitor prompts the user for additional approval. Note that the user interaction is only needed for files with download-and-execute patterns; *not* for each process creation in the system. Thus, the required user interaction is minimal. As stated in Section II, we assume that the host is *not* infected with malware that is capable of eavesdropping on user mouse events (and downloads immediate after the user clicks the mouse). This assumption is valid, as DeWare aims to detect the onset of infection on a clean host.

IV. Prototype Implementation in Windows

We describe our implementation of DeWare prototype in Windows XP utilizing some existing kernel driver and tools for monitoring system events in Windows. The prototype is easy to deploy and efficient to run.

A. File-System Monitor and Execution Monitor

Our file-system monitor consists of a kernel-space driver and a user-space component. File-creation events are collected at the kernel level and reported to the user space for filtering. The kernel-space driver filters all IRP (I/O request packet) calls issued by operating system to the low level I/O devices. We specify the applications to be monitored and focus on the FILE_CREATE system call. Other file-related system calls such as FILE_OPEN and FILE_OVERWRITE may be recorded as well. For intercepting file-related system calls, our prototype utilizes an existing kernel driver (Minispy) for Windows OS that can monitor all system calls involving opening a handle to a file object, including file creation.

TABLE I
USER INPUTS RECORDED THROUGH OUR FIREFOX EXTENSION.

Download Scenarios	Recorded
Clicking on a link	Yes
Typing a URL into address bar	Yes
Using "Save Target As..." button	Yes
Download initiated by page redirect	Yes
Download from embedded plugin	No

DeWare keeps track of targeted processes as well as their child processes. We organize the directories in both accessible and downloadable areas with the an array data structure for each process. If a file is to be created outside the accessible area of the process, then the event along with its attributes is reported to the user space. This verification has low overhead due to small search array sizes. In the downloadable area of user, high-risk files as defined by [20] in that area are examined in our dependency analysis. Certain low-risk downloads (e.g., TEXT files) can be safely ignored, but we monitor the area for unauthorized execution.

Our execution monitor leverages the security-monitoring features provided by Windows OS (XP and higher), which comes with security settings for monitoring the local host, and among those the AuditPolicy is able to track all the processes. The execution monitor records all the process start and exit events, and applies policies to inspect them for violations.

B. Collect and Correlate User Behavior Information

DeWare implements two independent mechanisms for collecting user inputs: one at kernel level and another within the Firefox browser. The kernel-level logging records user inputs through hooks SetWindowsHookex provided by Windows OS. The choice of proper temporal parameters is discussed and evaluated in Section V. We also develop a Firefox extension (based on tlogger) which is capable of recording users' clicks that correspond to downloading activities. This extension provides semantic properties of user input as shown in Table I. We note that if a user clicks on buttons within a browser plug-in (e.g., Adobe Reader) to download, then the extension is unable to record the event. With the gathered semantic properties, the accuracy of dependency inference can be significantly boosted.

Our extension captures user's activities on a web page and also the information from the common download-dialog window, which asks the user for file-download confirmation. We retrieve the download-event related attributes such as file name, source, and file type. Within the XUL files which specify our additional extension functions, we include JavaScript to listen for user's behavior such as click on the OK button. Once the user confirms her download in the dialog window, our extension records the relevant attributes regarding this download event into a log file.

In our prototype, the dependency analysis is performed offline, after data is collected. We also implement a real-time system based on temporal correlation. Our solution does not create any new privacy vulnerability. DeWare is a stand-alone

host-based solution that does not export the collected data out of the user's computer. Collected data is erased after the analysis and does not need to be stored for a long term.

Limitations DeWare is capable of detecting a wide spectrum of syndromes associated with infection onset. Our detection assumes that malware either makes persistent changes to the disk or creates its own new process. Thus, DeWare cannot detect the infection onset where code or dynamic loadable library (DLL) is injected into the memory of a legitimate process [26], [27], [28]. This type of in-memory injection does not need to touch the hard disk and the malicious code can run in the context of the compromised process without the creation of a new process.

An attacker may try to circumvent our detection by injecting fake inputs, and the purpose is to bypass correlation and to authenticate illegal downloads. This type of injections can be identified with the help of a cryptographic hardware attester, such as TPM [32].

V. Experimental Evaluation

We carry out extensive experiments to evaluate the effectiveness and usability of our solution. We perform a user study with 21 users to collect real-world user download behavior data. We also use DeWare to evaluate a large number of both legitimate and malicious websites for testing its accuracy.

A. Detecting Known DBD Exploits and Real-World Malicious URLs

We test DeWare against real-world websites with drive-by download exploits [16], [17]. The testing system is Windows XP Version 5.1 (Service Pack 2) installed within VMware 7.0. Internet Explorer Version 6.0.2900.2180 is used as the victim application. We take a snapshot of the clean system and then test DeWare with malicious websites. We give each URL one minute to load and launch any attack and revert the system to the initial clean state before each new test. During a two-week period, we successfully detected 84 unique domains with drive-by download exploits. Here are the observations.

- The malicious websites typically download executables and .dll files onto the victim's host, and then try to get the executables running. The entire procedure is surreptitious. In some cases, after reboot the browser is automatically loaded and re-directed to pornography websites.
- There are several popular exploit kits, such as Phoenix exploit kit and Eleonore exploits pack, which are widely used by many malicious websites. They target at multiple software vulnerabilities including Flash, PDF, Java and browser.
- There are websites who track the incoming requests. In that case, the first visit triggers the exploit, while during the second visit no exploit is observed or the web server refuses the connection.
- Some exploits attempt to download executables to directories such as

Documents and Settings\Administrator

\Local Settings\Temp\ to avoid detection, which can be detected by DeWare.

We also produce several known drive-by-download exploits in a lab environment shown in the following. DeWare can successfully detect executables downloaded as a result of a successful exploit.

- Heap Feng Shui attack
- HTML Object Memory Corruption Vulnerability
- Superbuddy exploits through AOL activeX control
- Adobe Flash player remote-code execution
- Microsoft Data Access Component API misuse
- DBD exploiting IE 7 XML library

B. User Study

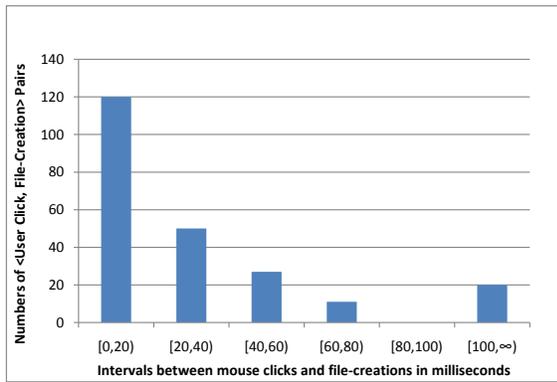
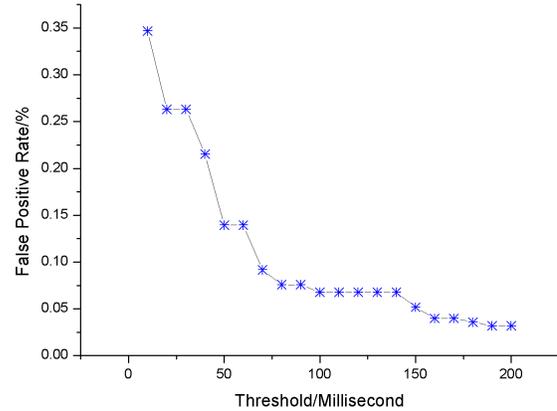


Fig. 2. Histogram on the interval in milliseconds of the user click events and their corresponding file creation events.

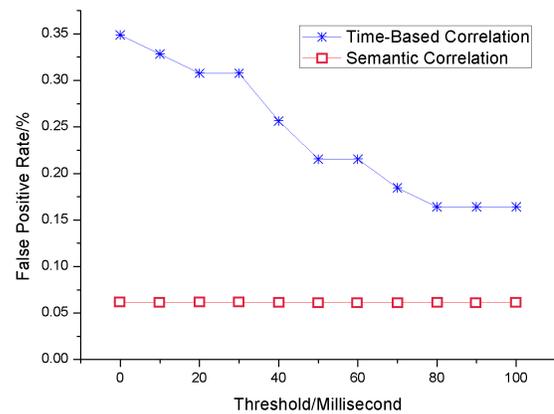
21 users participated in our user study – all of them are graduate or undergraduate students from a university. Each user is asked to surf the web for 30 minutes and download at least 10 files of her choice. In Figure 2, we give the histogram of the observed intervals between a user’s mouse-click event and the corresponding file-creation event. For this analysis, the two types of events are correlated manually by the authors. User’s click events are recorded at the kernel level through a mouse hook to the input device driver, and the timestamps for file-creation events are extracted from the intercepted system calls. The majority of user-triggered download have a short delay within 80 milliseconds.

The false positives based on the temporal correlation (i.e., comparing timestamps of events) are reported in Figure 3(a). False positives in our user study may come from two sources. *i)* File creation from user download in the downloadable area of the user that has a high risk extension, but is not within the required threshold, and *ii)* (legitimate) file creation by the browser that is not in the accessible area. During the user study, we have six such violations, e.g., \Program Files\NOS\bin\getPlusPlus_Adobe.exe, which all result from a single installation of *getPlus* (a download manager from NOS). *getplus* is piggy-backing downloaded with Adobe Reader.

The results show that the number of false alarms that temporal-comparison based DeWare generates in our user study is small ($< 1\%$) compared to the total number of file creations. A larger threshold leads to a lower false positive rate, but at the same time it also increases the likelihood of false negatives as we show earlier. Given our results, we recommend a threshold within 80-100 milliseconds. The false positives are further reduced with the application-assisted semantic comparison.



(a)



(b)

Fig. 3. (a) The averaged false positive rate vs. various threshold values based on a total of 25,092 file-creation events including temporary files by the browser. (b) Reduced false positives with semantic-based comparison between user’s mouse-click events and downloaded files.

For eight participants in our user study, we also collected application-level inputs using the Firefox extension described in Section IV. It records users’ mouse-click activities within webpages and on download-dialog windows, and obtains semantic information of the user event such as file name, type, size, source URL, timestamp, and destination directory. Without applying any dependency rules, there are 34 false positives for these eight participants in our dataset. With the temporal constraint (*Rule 3* in Section III-D) and process constraint (*Rule 2*), our analysis generates 16 false positives. The threshold τ is set to 100 milliseconds. With all rules including the comparison with semantic properties of user-input events, we further reduce the false positives from 16

to 6. Figure 3(b) shows that the semantics of user actions helps reduce the false positive rate. These remaining six false positives are due to JavaScript files that are bundled together with a webpage a user downloaded.

False alarm evaluation with legitimate websites. Given the pre-defined accessible area (consisting of 18 folders) of Firefox, we test DeWare with legitimate websites to see if there are any false alarms. A false alarm may be caused by a website *i*) downloading files outside the accessible area, or *ii*) creating new processes from files stored in the accessible area. We automatically evaluate 2000 websites from Alexa.com on August 20th, 2010. We give the browser 30 seconds to load a webpage. We have zero false alarm in our experiment. The result indicates that our pre-defined accessible area is sufficient in containing the files that the browser and common plug-ins create. We demonstrate the feasibility of confining a process' access to file system for reducing DeWare's workload of system-call monitoring. We are able to achieve it without the need of modifying how browser operates.

VI. Related Work

Jaeger and colleagues did pioneering work in operating-system security and security kernel architecture for OS-level control of program behaviors, including regulating downloaded executable content [11] and general-purpose policy enforcement through intercepting inter-process communication in OS [10]. In comparison with existing OS-security work, our work is more specific but not limited to the browser environment. The novelty of our solution is the incorporation of user-behavior characteristics and the leveraging of system-event dependency in evaluating security policies.

Cova, Kruegel, and Vigna proposed a DBD detection solution [4] that abstracted and categorized commonly shared features in the Javascripts that launch drive-by-download attacks. Another DBD-detection solution was proposed based on monitoring browser's inter-module communication patterns [29]. The methods used in [4], [13], [23] are based on feature extraction and classification from malicious code such as existence of redirection and obfuscation. Careful preparation and selection of features make this feature-based detection robust against known threats. [19], [37] both utilize virtual machine to load webpage, take records and perform post-analysis. This class of behavior-based detection is better at catching 0-day attack. Instead of implementing a client-side protection, [19], [23] choose to place their detection on a proxy as a network service, which helps reduce the client side overhead and also increase efficiency. In [7], string buffers are checked for executable instructions, which enables the solution to detect the shellcode before an vulnerability can be exploited. Egele, Kirda, and Kruegel [6] gave general introductions to drive-by exploits and mentioned possible detection and mitigation approaches (without concrete implementations).

The work that is conceptually close to ours is BLADE [15], which is an effective host-based solution independently developed by Lu *et al.* for detecting drive-by downloads, in

particular unconsented-content execution [15]. The authors performed extensive experiments in Windows OS environments. Although sharing the same goal, our technical approaches differ significantly, particularly on how to obtain user behavior information and how file system is monitored. DeWare treats applications as a black box – passively monitors the operating system and does not change to how applications access the existing file systems. BLADE performs redirections on browsers' I/O requests. Our solution is designed for general applications, not specific for the browser; thus we provide monitoring mechanisms in the kernel level, which are external and independent of the application. DeWare functions well even without the optional semantic comparison that involves an application-level component. Our experimental evaluation approach is also different, including a nontrivial user study with 21 participants.

The Alcatraz work [12], [34] also aims at malware defense, but by detecting the kind of malware that infects victim machines through user permitted installation. Botzilla [24], on the other hand, detects malware at a later point after the infection. Their approach is based on the analysis of traffic pattern when a malware initiates contact with its maintainer.

There exist several system integrity solutions that may bear superficial similarity to our work. Tripwire [30] is a cryptographic-based solution that aims to detect tampering in files by comparing the cryptographic hash values of file systems. Baliga, Ifode, and Chen proposed a rootkit containment strategy Paladin [2] that is based on tracking processes' hierarchical relationships and their corresponding file creation, and using policies to restrict processes' privileges to directories and memory access. What sets our work apart is that *i*) we correlate user inputs with file system monitoring; and *ii*) we focus on distinguishing authorized and illegal file creation. It is worth mentioning that in Paladin virtual machine monitor (VMM) is used to enforce the integrity of the detection, namely policy tables. VMM can be applied in our model to relax the assumption of the trust on the operating system.

Many browser security solutions have been proposed, including secure browsers and browser-as-an-OS [9], [36], securing browser extensions [14], and browser mashup security [35], [39]. Our work fundamentally differs from the secure browser line of research, as our solution is completely independent of the browser without any assumption on its or its components' integrity – yielding a more robust detection technique.

VII. Conclusions and Future Work

We illustrated the analysis of user behaviors for protecting the system integrity of a computer, in particular for detecting the onset of malware infection via drive-by download. We described the design, implementation, and use of *DeWare* for host-based security protection against unauthorized system events through enforcing dependency among events within the operating system environments. We realized DeWare through defining and enforcing access-control policies across multiple

domains (i.e., file system, process management, and user inputs) within the operating-system environment. Our user-behavior based policies are new and we demonstrated their use in solving the practical DBD detection problem. DeWare can be easily deployed and used in Windows.

Tracking the origin and provenance of critical system events by enforcing the dependency between them and user actions is a novel and effective approach for managing a secure host. We envision that our system-level access control approach can be generalized beyond file and process events within OS environments. For example, the dependency between user actions and outbound network traffic can be identified and enforced for blocking stealthy malware in the future.

Acknowledgement We would like to thank the first author of [15] for the information on malware websites.

References

- [1] G. K. Baah, A. Podgurski, and M. J. Harrold. The probabilistic program dependence graph and its application to fault diagnosis. In *Proceedings of the 2008 international symposium on Software testing and analysis, ISSTA '08*, pages 189–200, New York, NY, USA, 2008. ACM.
- [2] A. Baliga, L. Iftode, and X. Chen. Automated containment of rootkits attacks. *Computers & Security*, 27(7-8):323–334, 2008.
- [3] E. Chien. The new generation of targeted attacks, 2010. Keynote in Recent Advances in Intrusion Detection (RAID).
- [4] M. Cova, C. Kruegel, and G. Vigna. Detection and analysis of drive-by-download attacks and malicious javascript code. In *Proceedings of 19th International World Wide Web Conference*, 2010.
- [5] M. Cruz. Most abused infection vector, 2008. Trend Micro. <http://blog.trendmicro.com/most-abused-infection-vector/>.
- [6] M. Egele, E. Kirda, and C. Kruegel. Mitigating drive-by download attacks: Challenges and open problems. In *Proceedings of the Open Research Problems in Network Security(iNetSec)*, pages 52–62, 2009.
- [7] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *Proceedings of the Sixth Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2009.
- [8] D. A. Freedman. *Statistical Models and Causal Inference: A Dialogue with the Social Sciences*. Cambridge University Press, 2010.
- [9] C. Grier, S. Tang, and S. T. King. Secure web browsing with the OP web browser. In *IEEE Symposium on Security and Privacy*, May 2008.
- [10] T. Jaeger, J. Liedtke, and N. Islam. Operating system protection for fine-grained programs. In *Proceedings of the 7th USENIX Security Symposium proceedings*, 1998.
- [11] T. Jaeger, A. D. Rubin, and A. Prakash. Building systems that flexibly control downloaded executable content. In *Proceedings of the 6th USENIX Security Symposium*, July 1996.
- [12] Z. Liang, W. Sun, V. N. Venkatakrishnan, and R. Sekar. Alcatraz: An isolated environment for experimenting with untrusted software. *ACM Trans. Inf. Syst. Secur.*, 12:14:1–14:37, January 2009.
- [13] P. Likarish, E. E. Jung, and I. Jo. Obfuscated malicious javascript detection using classification techniques. In *Proceedings of 4th International Conference on Malicious and Unwanted Software*, 2009.
- [14] M. T. Louw, J. S. Lim, and V. N. Venkatakrishnan. Enhancing web browser security against malware extensions. *Journal in Computer Virology*, 4(3):179–195, 2008.
- [15] L. Lu, V. Yegneswaran, P. Porras, and W. Lee. Blade: An attack-agnostic approach for preventing drive-by malware infections. In *Proceedings of 17th ACM Conference on Computer and Communications Security*, 2010.
- [16] www.malwaredomainlist.com.
- [17] www.malwareurl.com.
- [18] J. A. Morales, E. J. Kartaltepe, S. Xu, and R. S. Sandhu. Symptoms-based detection of bot processes. In I. V. Kottenko and V. A. Skormin, editors, *MMM-ACNS*, volume 6258 of *Lecture Notes in Computer Science*, pages 229–241. Springer, 2010.
- [19] A. Moshchuk, T. Bragin, and D. Deville. SpyProxy: Execution-based detection of malicious web content. In *Proceedings of the 16th USENIX Security Symposium*, 2007.
- [20] Microsoft high-risk extensions. <http://support.microsoft.com/kb/883260>.
- [21] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. The ghost in the browser analysis of web-based malware. In *HotBots '07: Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets*, Berkeley, CA, USA, 2007. USENIX Association.
- [22] Apple QuickTime 7.3 RTSP Response Exploit, CVE-2007-6166, <http://securityevaluators.com/content/case-studies/sl/>.
- [23] K. Rieck, T. Krueger, and A. Dewald. Cujo: efficient detection and prevention of drive-by-download attacks. In *Proceedings of the 26th Annual Computer Security Applications Conference, ACSAC '10*, pages 31–39, New York, NY, USA, 2010. ACM.
- [24] K. Rieck, G. Schwenk, T. Limmer, T. Holz, and P. Laskov. Botzilla: detecting the “phoning home” of malicious software. In *Proceedings of the 2010 ACM Symposium on Applied Computing, SAC '10*, pages 1978–1984, New York, NY, USA, 2010. ACM.
- [25] J. Shirley and D. Evans. The user is not the enemy: Fighting malware by tracking user intentions. In *Proceedings of New Security Paradigms Workshop (NSPW)*, pages 22–25, September 2008.
- [26] skape. Understanding windows shellcode. <http://www.nologin.org/Downloads/Papers/win32-shellcode.pdf>, 2003.
- [27] skape. Metasploit’s meterpreter. <http://www.nologin.org/Downloads/Papers/meterpreter.pdf>, 2004.
- [28] skape and J. Turkulainen. Remote library injection. <http://www.nologin.org/Downloads/Papers/remote-library-injection.pdf>, 2004.
- [29] C. Song, J. Zhuge, X. Han, and Z. Ye. Preventing drive-by download via inter-module communication monitoring. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2010.
- [30] E. H. Spafford and G. Kim. The design and implementation of tripwire: A file system integrity checker. In *2nd ACM Conf. on Computer and Communication Security (CCS)*, 1994.
- [31] A. Stamminger, C. Kruegel, G. Vigna, and E. Kirda. Automated spyware collection and analysis. In P. Samarati, M. Yung, F. Martinelli, and C. A. Ardagna, editors, *ISC*, volume 5735 of *Lecture Notes in Computer Science*, pages 202–217. Springer, 2009.
- [32] D. Stefan, C. Wu, D. Yao, and G. Xu. Cryptographic provenance verification for the integrity of keystrokes and outbound network traffic. In *Proceedings of the 8th International Conference on Applied Cryptography and Network Security (ACNS)*, June 2010.
- [33] B. Stone-Gross, M. Cova, L. Cavallaro, B. Gilbert, M. Szydowski, R. A. Kemmerer, C. Kruegel, and G. Vigna. Your botnet is my botnet: analysis of a botnet takeover. In E. Al-Shaer, S. Jha, and A. D. Keromytis, editors, *ACM Conference on Computer and Communications Security*, pages 635–647. ACM, 2009.
- [34] W. Sun, R. Sekar, Z. Liang, and V. N. Venkatakrishnan. Expanding malware defense by securing software installations. In *Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment, DIMVA '08*, pages 164–185, Berlin, Heidelberg, 2008. Springer-Verlag.
- [35] H. J. Wang, X. Fan, J. Howell, and C. Jackson. Protection and communication abstractions for web browsers in MashupOS. In *ACM Symposium on Operating Systems Principle (SOSP)*, pages 1–16. ACM Press, 2007.
- [36] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The multi-principal OS construction of the Gazelle web browser. In *Proceedings of the 18th Usenix Security Symposium*, August 2009.
- [37] Y.-M. Wang, D. Beck, X. Jiang, R. Roussee, C. Verbowski, S. Chen, and S. King. Automated web patrol with Strider HoneyMonkeys: Finding web sites that exploit browser vulnerabilities. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*, 2006.
- [38] H. Xiong, P. Malhotra, D. Stefan, C. Wu, and D. Yao. User-assisted host-based detection of outbound malware traffic. In *Proceedings of International Conference on Information and Communications Security (ICICS)*, December 2009.
- [39] S. Zarandioon, D. Yao, and V. Ganapathy. OMOS: A framework for secure communication in mashup applications. In *ACSAC'08: Proceedings of the 24th Annual Computer Security Applications Conference*, pages 355–364, December 2008.