View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Computer Science Technical Reports @Virginia Tech

User-Behavior Based Detection of Infection Onset*

Kui Xu
Department of Computer Science
Virginia Tech
xmenxk@cs.vt.edu

Danfeng (Daphne) Yao
Department of Computer Science
Virginia Tech
danfeng@cs.vt.edu

Qiang Ma
Department of Computer Science
Rutgers University
qma@cs.rutgers.edu

Alexander Crowell
Department of Computer Science
Rutgers University
acrowell@eden.rutgers.edu

Abstract A major vector of computer infection is through exploiting 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* for detecting the onset of infection delivered through vulnerable applications. DeWare explores and enforces *causal relationships* between computer-related human behaviors and system properties, 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.

Besides the concrete DBD detection solution, we also formally define causal relationships between user actions and system events on a host. Identifying and enforcing correct causal relationships have important applications in realizing advanced and secure operating systems. We perform extensive experimental evaluation, including a user study with 21 participants, thousands of legitimate websites (for testing false alarms), as well as 84 malicious websites in the wild. Our results show that DeWare is able to correctly distinguish legitimate download events from unauthorized system events with a low false positive rate ($< 1\%$).

Keywords: Malware, drive-by-download, user inputs, integrity, file system.

*This work has been supported in part by NSF grant CAREER CNS-0953638 and CNS-0831186.

1 Introduction

Malicious software (malware) stealthily downloaded from the Internet has been the leading infection vector, accounting for 53% of all incidents in 2008 [4]. Malware may be delivered stealthily through a networked application such as a browser, a peer-to-peer file sharing client, or a chat application. For example, spyware may be bundled with files shared through P2P networks. 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 [23]. 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 the user permission. Botnets often use DBD as the initial infection vector, e.g., Torpig [36]. 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 [24].

Conventional signature-based techniques may not be effective against zero-day exploits or code with sophisticate 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 demonstrate the feasibility and quality of such a host-based monitoring framework. We aim to provide a tool 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.

We present *DeWare* – a host-based security tool for

detecting the onset of malware infection at real time. Deware is application independent, thus it is capable of performing host-wide monitoring beyond the browser. DeWare's 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 [2]. Specifically, DeWare collects and analyzes file-system events and process properties (namely system calls related to file creation and process creation) in the kernel space.

There are several nontrivial technical challenges that we address in order to realize our goal. One is how to distinguish drive-by downloads from legitimate downloads. *Our key observation is that legitimate file creation is mostly triggered by the user.* Our solution is to strategically monitor file-system events and correlate them with observed user actions. We aim to identify the *causal relationship* between system events (e.g., file or process creation) and user activities. We collect information regarding the user's mouse actions at the kernel level. Our knowledge about user behaviors may be further refined with additional application-specific information.

We note that the problem of finding the causal relationship between two or more sets of *natural* events is difficult in general [9] – the correlation of two observed data sets may not indicate a causal relationship between them, as there may be other factors. In the context of operating system, however, system events are *artificially* created in response to user actions and are not natural events. Thus, identifying causal relationship between user actions and system events on a computer can be achieved, because of the obvious underlying connections between them.

Another technical issue we address is how to efficiently handle the *voluminous* application-triggered benign downloads, which are *not* directed caused by user actions. We refer to these downloads as *indirectly* caused by the user's action. For example, when a user types `www.cnn.com` in the address bar, the browser fetches the main page `index.html` as well as any objects embedded in that page, including JavaScript files, images and flashes. It automatically creates temporary files *without* explicit user requests.

Our approach is two-fold: **i)** to minimize false alarms due to application-triggered downloads, and **ii)** to prevent malware from hiding its executables in these temporary storage places. Realizing both goals are necessary for the integrity of the host. We are able to realize both requirements through two separate mechanisms as follows. We design a policy-based monitoring framework that controls applications' access to specified folders, *accessible area*. For example, as a result the browser process cannot create files in certain system directories at run time, but can download to temporary folders at will. In addition, to realize **ii)** we design an execution monitor that reports and halts any new process creation in accessible areas, as that area is typically for storing temporary read-only files (extensively validated by our experiments in Section 5). As a result, malware is unable to start any executables downloaded in the accessible area. These mechanisms enable us to largely reduce our scope for file-system monitoring scope, reduce false alarms, and provide comprehensive surveillance across the host.

Our contributions We summarize our technical contributions as follows.

- We give new definitions for direct and indirect causal relationships among user actions and system events in the context of host security. Identifying and enforcing causal relationships are useful in detecting the onset of infection. We point out the associated technical challenges and usability requirements. We believe that the integration of human behaviors can greatly improve the design and effectiveness of cyber security solutions.
- We present our design and implementation of a tool called *DeWare* that detects the onset of infection including drive-by downloads, through the efficient and comprehensive system monitoring and data analysis. DeWare is easy to deploy. It identifies the infection onset by detecting download-and-execute patterns that most stealthy malware exhibits.
- 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-triggered downloads with low false positives (<1%). In addition, DeWare triggers no false alarms when being automatically tested on 2000 legitimate websites.

One main advantage of our (basic) DeWare design is its application-independence – there is no need to modify the application, or understand how it works. Our inspection is external to applications and treats them as a black box. This feature differentiates our solution from others (such as download managers of browsers and some academic solutions [17, 29]).

Organization of the paper The rest of the pa-

per is organized as follows. We give our definitions for causal relationships as well as our security model in Section 2. An overview of our design of DeWare is in Section 3. Our *DeWare* implementation in Windows operating system is presented in Section 4. Extensive experimental evaluation is described in Section 5. Related work is compared in Section 6, and conclusions and future work are described in Section 7. Some experimental results and pseudo-code of our analysis algorithm are in the Appendix.

2 Definitions For Causal Relationships and Security Models

Intuitively, DeWare detects the onset of infection by detecting anomalies on a host. Given a system event e, we aim to infer the *provenance* of the event by constructing a causal-relationship map that identifies another event that directly or indirectly triggers e. In our model, we broadly define two types of events: *user action* and *system event*. User actions include keyboard inputs and mouse clicks. There are two approaches to observe user events – one is at the kernel level through the placement of hooks and listeners, and one is at the application level through interfacing with the application that consumes the user actions. For example, at the kernel level the information related to a user's mouse click may include coordinates of the click, ID of the process receiving the inputs. In comparison, semantic information associated with the user actions has to be obtained at the application level, such as the URLs of the links that the user clicks on.

System events include any kernel-level transactions such as file system events, network events (outbound and incoming), process events. Each user event or system event e has a timestamp T_e. In this paper, for detecting drive-by download we focus on two system events, namely file creation and process creation. System events are triggered either by user actions or other system events. We define direct causal relationship and indirect causal relationship as follows and give some examples.

Definition 2.1 *Direct causal relationship is between two events e and e', where $T_e < T_{e'}$ and the event e directly triggers or causes the event e'.*

We refer to the event e as trigger event, and e' as effect event. We denote the direct causal relationship between them by $e \rightarrow e'$. Trigger event can be either a user action or a system event. The effect event is usually a system event. A user may react to a system event, e.g., the email inbox receives a new message, then the user clicks on the inbox to view it. Thus, the effect event can be a user action as well.

Definition 2.2 *Indirect causal relationship is between two events e_i and e_j $(j > i)$ and $T_{e_i} < T_{e_j}$, where there exists an event sequence $e_i, e_{i+1}, e_{i+2}, \ldots, e_j$ such that $e_i \rightarrow e_{i+1}$, $e_{i+1} \rightarrow e_{i+2}$, ..., and $e_{j-1} \rightarrow e_j$.*

That is, event e_j is indirectly triggered by event e_i, which directly triggers e_{i+1}, and e_{i+1} directly causes e_{i+2}, and so on. We denote the indirect causal relationship between events e_i and e_j by $e_i \rightsquigarrow e_j$.

For example, a user's mouse click may directly cause a browser to send some packets or download files from remote sites, or directly cause an application to start (i.e., creating a new process). For an indirect causal relationship between a mouse click and a file creation, consider the following scenario. Images embedded in `index.html` page are automatically fetched by the browser. Thus, the user's click on `index.html` indirectly causes the creation of image files in the temporary folder. Causal relationships should be defined with a specified granularity, as different granularity result in different direct and indirect causal relationships.

Given observed user's mouse-click events and file and process creation events, we aim to infer their causal relationships based on pre-defined rules, and to identify suspicious system events that cannot be attributed to any user action. Our inference approach involves the comparison of attributes of events including timestamps, file types, process IDs, and semantic information. In this work, the causal relationship among events is inferred or estimated, thus is not provably-assured. Cryptographic provable provenance has been realized in the host-based system security settings to ensure keystroke authenticity and traffic generation [34].

Our work belongs to a broader category of research on *human-behavior driven malware detection*. We define that the key research problems in human-behavior driven malware detection include *i)* how to select humans' characteristic behavior features, *ii)* how to prevent malware forgery, and *iii)* how to make security solutions nonintrusive and transparent to users. Anti-forgery techniques have been proposed with the help of tamper-resistant cryptographic hardware (namely TPM) [12, 34]. Researchers have investigated keystroke dynamics [35], Internet chat be-

haviors [10] for anomaly-detection purposes. For *iii)* Nonintrusiveness requires any (host-based) security solution to produce *minimal* disruption and annoyance to users.

Security solutions that rely on constant user interactions (for example, through pop-up windows as in [41]) may distract users and cause usability issues. Besides, feedback supplied by users may not be accurate and reliable. Our detection in DeWare collects and utilizes the *existing* human computer interactions, in particular user inputs to applications, for security detection. The tool does not require user to perform additional tasks, thus is easy to use.

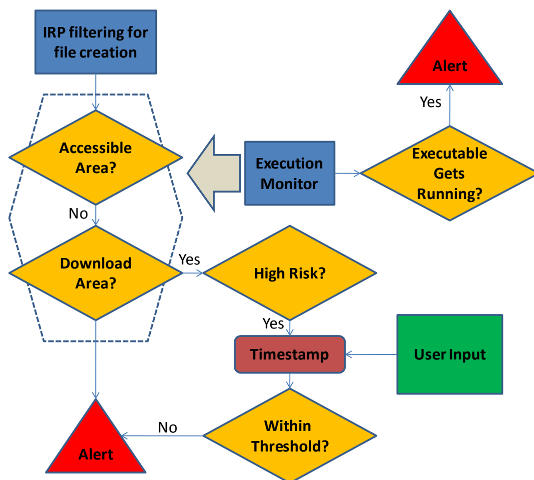

Figure 1: Schematic drawing of the work flow for detecting the infection onset on a host based on analyzing user-behaviors and file system/process properties.

Security and attack models We assume that the browser and its components are not secure and may have software vulnerabilities. The operating system is assumed to be trusted and secure, and thus the kernel-level monitoring of file-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. 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 [40].

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. (Recent Hydraq malware started with a personalized spam message with an embedded link to attacker's website [2].)

3 Overview of DeWare Design

The purpose of DeWare is to detect unauthorized system activities, in particular file and process creations on a host. We consider the system events directly caused by the owner (human user) of the computer legitimate. Malware-triggered download and execution are unauthorized. We realize DeWare by instrumenting specialized sensors (i.e., components mostly at the kernel level for collecting and monitoring system-event data), and inferring at real-time legitimate events directly or indirectly caused by the human user's activities. Thus, file creation or process creation that cannot be attributed to any human actions (namely mouse activities) is likely to be suspicious and caused by malware infection. DeWare is capable of host-wide monitoring that is application independent.

System events that we focus on in this work are the file creation and process creation – either of the two characteristics is often observed at the onset of malware infection. Similar assumption was previously used by others as well [17, 40]. We discuss the limitations of DeWare against more sophisticated malware in Section 5.

Although this intuition of DeWare is simple, technical challenges arise due to 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 area and is not executed. Violations of the policies are likely caused by malware infection and are reported. We describe it in more details in Section 3.1 and 3.2.

In our detection system, users' intention is embodied by their mouse 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 3.3. 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 (described in Section 2), 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 [34].

In our model, the file downloaded by the user through specific mouse operations is legitimate. These files are *directly caused* by user actions, i.e., direct causal relationship.

The work flow in DeWare is shown in Figure 1. Putting these all together, DeWare includes the following components.

- A file-system monitor for intercepting file-creation related system calls;
- A click recorder for collecting user behaviors information;
- An execution monitor for inspecting a portion of file system for illegal process creation.

3.1 File-System Monitor: Confining Application-Triggered Downloads

Application-triggered downloads are indirectly triggered by user actions. These downloads are legitimate, provided that the application is not comprised, e.g., the application is not infected with virus or parasitic malware [32]. 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, during a user's 30-minute browsing period in Internet Explorer, a user indirectly triggers 482 file creations in Temporary Internet Files folder and 47 Cookies directory. 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 create files. The application is not allowed to create 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. For Firefox, we specify 18 folders shown in the Appendix. As a result, we significantly reduce the number of directories to monitor for file creation, and the number of events to record. This approach of defining policies to confine processes is similar to what is used in SELinux. One difference is that our detection requires real-time data computation, which is beyond the use of rules. Our solution is also simpler, as it is specific to file-system access and monitoring.

File-system monitor intercepts system calls related to file creations, and probes kernel data structures to gather process information. Timestamps can be obtained from input logger at runtime to perform temporal correlation. With this file-system monitor, malware is confined to the accessible area. Moving executables out of the accessible area is also forbidden.

Download directly triggered by the user 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 more details in Section 3.5.

With the aforementioned file-system monitor, an attacker may still be able 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.

3.2 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 executables stored at accessible area from being run. Execution monitor which gives additional inspection to areas where access is granted to an application.

Information of a newly created process that we collect includes 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.

Our security model assumes the operating system is not compromised, in particular, the kernel is not compromised, the process information observed is complete and trustworthy. Therefore, we do not consider the existence of rootkits that hide their presence in the process table. Because we aim to to detect the onset of the infection on an otherwise clean host, this assumption is valid.

3.3 Intercepting Users' Input Activities

Obtaining user intention can be realized in three different ways: *i)* directly asking the user (e.g., through a pop-up window) [41], *ii)* analyzing user's transaction history and extracting patterns [25], or *iii)* recording the events entered through keyboard or mouse devices [34]. 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. Click recorder intercepts information about user inputs – coordinates, timestamp, and content – by placing listeners or hooks in the kernel. The inputs go to 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 at the application. This approach – referred as application-level input logging – 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, for example, 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 (See also Section 5.2).

3.4 Rule-Based Causal-Relationship Analysis

DeWare infers causal relationships between user's mouse clicks and file-creation events based on temporal-based comparison rules and/or semantic-based comparison rules.

For temporal-based comparison, a legitimate file-creation event is defined as one that takes place within a short threshold τ after a valid user-input event. Different τ values can significantly affect the false positive and false negative rates, which are experimentally evaluated in our user study in Section 5. The two events being compared need to have the same process ID.

The semantics of a mouse click event includes information about the file or URL that the user downloads or clicks, for example, source URL, file name, type, size, and destination directory. Therefore, in semantic-based comparison these pieces of information need to match with the file being created, in addition to the temporal constraint on event timestamps. We describe how the semantics information can be obtained in Section 4.

3.5 Coincidental Download, Piggybacking Download, and Their Detection

Our simple temporal-based rule does not provide sufficient security protection against infection onset, because of coincidental malware download and piggybacking malware download defined 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 3.3.

Piggyback download defined in [33] is where malware is part of a piece of software downloaded with proper user permission, e.g., spyware bundled with compression software. (Stamminger *et al.* described several automated techniques for recognizing spyware [33].)

Preventing piggybacking code from being downloaded 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 2, 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.

4 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. We describe our realization of file-system monitor, execution monitor, click recorder, and causal-relationship analysis respectively.

File-System Monitor Our prototype builds on Minispy to intercept file-related system calls. Minispy is an existing kernel driver for Windows that monitors all system calls involving opening a handle to a file object, including file creation. 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 program images (running process names) to be monitored, so only the IRP calls issued by those programs are collected. In our prototype, we focus on the FILE_CREATE system call. Other file-related system calls such as FILE_OPEN and FILE_OVERWRITE may be recorded as well.

We keep track of targeted processes as well as their child processes. We implement the directories appeared in both accessible and downloadable areas with the array data structure for each process. Given a file-creation event requested by a process, the corresponding array is searched to check for violations. If file creation outside the permitted areas is found, then the user is alerted. This verification has low overhead, because of the small array sizes. In the downloadable area of user, certain low-risk downloads (e.g., TEXT files) can be safely ignored. High-risk files [21] in that area are examined in our causal-relationship analysis.

Execution Monitor Our execution monitor is implemented based on `PsTools` suite, which is an existing collection of command-line utilities to manage processes in Windows. The tool leverages the security-monitoring features provided by Windows OS. Windows OS (XP and higher) comes with security settings for monitoring the local host, among which 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. A sample log entry collected is shown as follows.

```
A new process has been created:
New Process ID: 3444
Image File Name: C:\WINDOWS\system32\cmd.exe
Creator Process ID: 1396
User Name: Administrator
Domain:  XMENXK-50C6105A
Logon ID:  (0x0,0xEB4D)
```

Click Recorder DeWare implements two independent mechanisms for collecting mouse clicks: one at kernel level and another within the Firefox browser. The kernel-level logging records user inputs at the kernel level through hooks `SetWindowsHookex` provided by Windows OS. We also write a Firefox extension (based on `tlogger`) that is capable of recording users' clicks that correspond to downloading activities. This extension – assumed to be trusted – provides semantics of user clicks, so one is able to learn the detailed meaning of what a click is intended to achieve as shown in Table 1. 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 semantic information, the inference accuracy can be improved (shown in Section 5). DeWare can securely function with only the kernel-level click logging.

Causal-Relationship Analysis In our prototype, the causal-relationship analysis is performed off-line, after data is collected. This implementation can be extended to realize real-time analysis. Our detailed analysis algorithm in pseudo-code is shown in Fig-

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

Table 1: User clicks recorded through our Firefox extension.

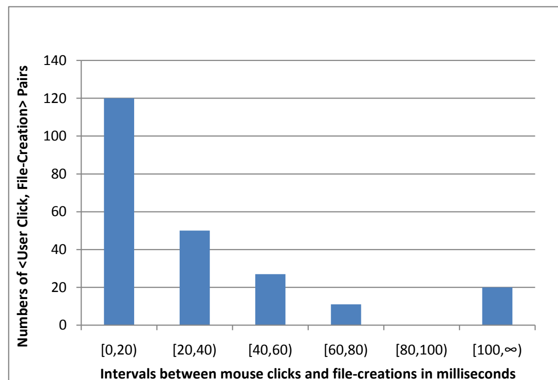

Figure 2: Histogram on the interval in milliseconds of the user click events and their corresponding file creation events.

ure 6 in the Appendix.

Privacy Discussion 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. Only mouse clicks are recorded (user's keyboard inputs are not). 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 is able to retrieve and store library files just in memory and load them directly. It is stealthy since it 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.

5 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 malware-hosting websites for testing its accuracy.

5.1 User Study

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 with Firefox browser for 30 minutes and download at least 10 files of her choice. Firefox 3.0.19 is run on Microsoft Windows XP Professional Service Pack 3.

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.

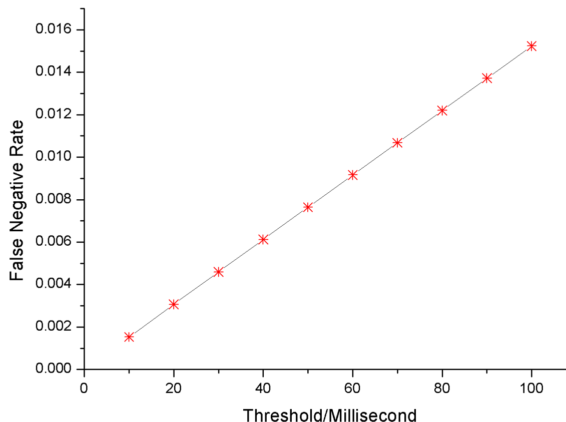

Figure 3: Simulated probability of false negatives caused by piggybacking download vs. allowed threshold between user click event and file creation.

Our false negative analysis shown in Figure 3 is based on simulating coincidental downloads (defined in 3.5). Here, a false negative is where a malware download is mistakenly classified as a legitimate download, due to the close proximity in time of the download event to an observed user mouse click. We aim to estimate the likelihood of having coincidental downloads.

We assume that malware is *equally* likely to be downloaded across the time spectrum considered. In Figure 3, X-axis is the threshhold in milliseconds, and Y-axis is our simulated false negative rate, which is computed as in Equation 1, where n is the number of users, m_i is the number of mouse clicks by user$_i$, $Surfing_time_i$ is user$_i$'s total computer-use time (30 minutes). Given threshold τ and a mouse click at time t, *safe_time* is from t to $t+\tau$, and is of duration τ. Thus, $m_i\tau$ gives the total safe_time for user$_i$.

$$\frac{1}{n}\sum_{i=1}^{n}\frac{m_i\tau}{surfing_time_i} \tag{1}$$

For example, the probability is 0.00154 for a threshold of 10 milliseconds and 0.01524 for 100 milliseconds. The likelihood of having false negatives due to coincidental download is low, and the value increases with increasing threshold, which is expected.

The false-positives based on the basic temporal correlation (i.e., comparing timestamps of events) are reported in Figure 4. False positives in our user study may come from two sources. *i)* File creation from user download in downloadable area of the user that has a high risk extension, but is not within the required threshhold, 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 result from a single installation of `getPlus` (a download manager from `NOS`) that is piggybacking downloaded with Adobe Reader.

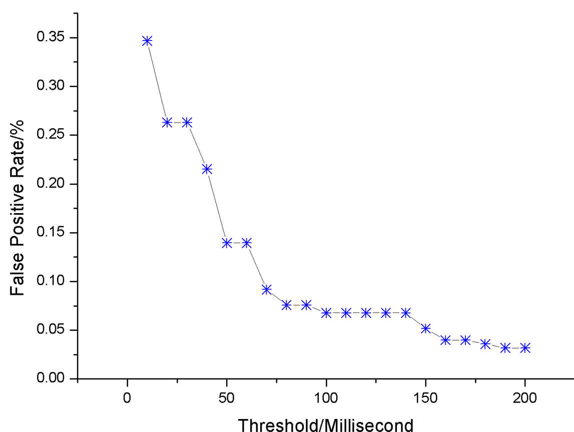

Figure 4: The averaged false positive rate vs. various threshold values based on a total of 25092 file-creation events including temporary files by the browser.

The results show that the number of false alarms that temporal-comparison based DeWare generates in our user study is small 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 in Section 5.2 with the application-assisted semantic comparison.

False alarm evaluation with legitimate websites. Given the pre-defined accessible area (consisting of 18 folders, shown in Appendix) of the Firefox browser, we test DeWare with 2000 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 evaluate the top 1000 and bottom 1000 websites according to the ranking from `Alexa.com` on August 20th, 2010 with Firefox. 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.

5.2 Reducing False Positives With Semantic Correlation

For 8 out of the 21 participants, we also collected application-level inputs using the Firefox extension described in Section 4. It allows us to collects users' mouse-click activities on download-dialogue windows, and obtain semantic information of the user event such as file name, type, size, source URL, timestamp, and destination directory. The timestamp, file name, and directory information are compared with those of the file-creation event for causal-relationship analysis. The mouse clicks that do not trigger download are ignored and not recorded. Figure 5 shows that the semantics of user actions helps reduce the false positive rate. The remaining (six) false positives were due to JavaScript files that came together with a webpage that a user downloaded.

5.3 Detecting Known DBD Exploits and Real-World Malicious URLs

We test DeWare against real-world websites with drive-by download exploits [18, 19]. The ex-

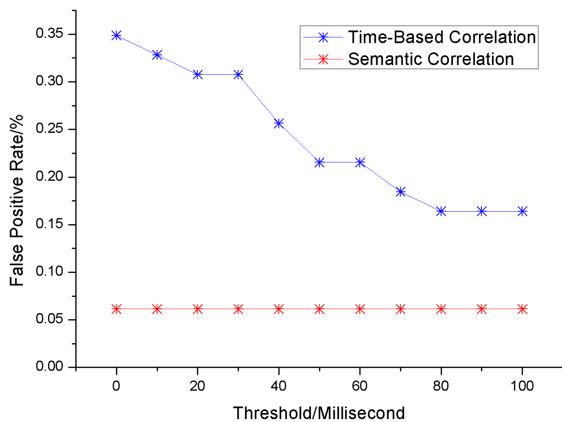

Figure 5: Reduced false positives with semantic-based comparison between user's mouse-click events and downloaded files.

periment is carried out on a laptop with 2.26 GHz dual-core CPU and 2 GB RAM. The testing system is Windows XP Version 5.1 (Build 2600.xpsp_sp2_rtm.040803-2158: Service Pack 2) installed within VMware 7.0. We use Internet Explorer Version 6.0.2900.2180.xpsp_sp2_rtm.040803-2158 as the victim application. We take a snapshot of a clean version of the system and then test DeWare with malicious websites. We give each URL one minute to load and launch the attack. We 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. Observations from the experiment are summarized as follows.

- 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 `\DocumentsandSettings\Administrator\LocalSettings\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 [30];
- HTML Object Memory Corruption Vulnerability [13];
- Superbuddy exploits through AOL activeX control [37];
- Adobe Flash player remote-code execution [8].
- Microsoft Data Access Component API misuse [6];
- DBD exploiting IE 7 XML library [14].

5.4 Evaluation of Off-the-Shelf Security Products Against a DBD Exploit.

We evaluate popular commercial anti-virus software and Internet security products against the IE 7 XML Heap Corruption exploit (VU#493881). The results are shown in Table 3 in the Appendix.

Our comparison shows that signature-based security monitoring is *not* effective in detecting drive-bydownloads, due to the variation in the malicious executables downloaded. For example, 360 Safeguard cannot detect IE 7 exploits with an older version (360v6.0.1) anti-virus driver and a newer definition database, but can do so vice versa. In our experiment, Trend Micro Internet Security Pro is not effective against the DBD attack tested. Although Zonealarm Pro and Microsoft Security Essentials alert the user the DBD threats, the malicious executables are still downloaded.

6 Related Work

Cova, Kruegel, and Vigna proposed a DBD detection solution [3] that abstracted and categorized commonly shared features in the Javascripts that launch drive-by-download attacks. They used machine learning techniques to build characteristics of normal Javascript code, and then in the detection phase the system is to identify malicious Javascript code. This system is built with the goal of detecting zero-day exploits. A DBD-detection solution was proposed based on monitoring browser's inter-module communication patterns [29]. The work utilizes the inter-module communications of browser to

Solutions	Architecture	Analysis	Browser Modified	Traffic Monitored	Security assumption
Obfuscated-JS [15]	Client side	Offline	No	No	N/A
SpyProxy [20]	Client side & Network Service	Online	No	No	Secure virtual machine
Inter-Module [29]	Client side	Online	Yes	No	Secure browser and OS
DBD-JS [3]	Client side	Online	Yes	Yes	Secure browser
Shellcode [7]	Client side	Online	Yes	No	Secure browser
Strider-HM [40]	Client side	Offline	No	Yes	Secure virtual machine
BLADE [17]	Client side	Online	Yes	Yes	Secure OS
DeWare	Client side	Online	No	No	Secure OS

Table 2: Comparison of select drive-by-download solutions.

collect signatures of drive-by-download exploits and performs online detection of known exploits.

The work that is conceptually close to ours is BLADE [17], which is a host-based drive-by-exploit detection framework in Windows. BLADE is an effective solution independently developed by Lu *et al.* for detecting drive-by downloads, in particular unconsented-content execution [17]. The authors performed extensive experiments in Windows OS environments. Although the goal of BLADE is the same as in ours, our technical approaches differ much, particularly on obtaining 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 all of them 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 different from what is reported in [17], as we carried out a nontrivial user study with 21 participants.

In our work, we also present and take the first step to formalize the new concepts of causal relationships and their important application in the context of securing personal computers, which is beyond the specific drive-by download detection problem studied.

We compared related work about DBD detection with ours in Table 2. The method used in [3, 15] is based on feature extraction 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. [20, 40] 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. In [7], string buffers are checked for executable instructions, which enables the solution to detect the shellcode *before* an vulnerability can be exploited. (Some papers did not specify about their security models, thus we infer them in Table 2.)

Egele, Kirda, and Kruegel [6] and Niki [22] gave general introductions to drive-by exploits and mentioned possible detection and mitigation approaches (without concrete implementations), such as emulation-based shellcode analysis technique based on existing work in [7], and analyzing DNS and web server relationships.

There exist several system integrity work that may bear superficial similarity to our work. Tripwire [31] is a cryptographic-based solution that aims to detect tampering in files by comparing the cryptographic hash values of file systems. Tripwire is not designed for detecting illegal file creation, as it does not have a classification mechanism for that purpose. Baliga, Iftode, and Chen proposed a rootkit containment strategy Paladin [1] 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. The part of our work on controlling the access

of processes to file systems has a similar spirit with Paladin. What sets our work apart is that i) we correlation user inputs with file system monitoring; and ii) we focus on distinguishing authorized and illegal file creation, whereas Paladin allows arbitrary 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 [11, 39], securing browser extensions [5, 16], and browser mashup security [38, 42]. 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.

7 Conclusions and Future Work

We describe the importance of inferring the causal relationships among the user actions and system events on a personal computer for security monitoring and forensic purposes. Specifically, the analysis on file and process creation enables the identification of the onset of infection through drive-by download, as demonstrated in this paper. We described the design, implementation, and use of *DeWare* for host-based security protection against unauthorized system events. A main technical challenge of our approach is how to accurately interpreting user intention and to tell apart legitimate user-triggered download, malware download, and benign application-triggered download. Our techniques are based on defining practical policies and the enforcement of policies through data collection and analysis. We performed extensive experiments to evaluate the usability, accuracy, and precision of DeWare, including a user study, tests on thousands of legitimate URLs for evaluating false positives, and tests on known malicious URLs. DeWare is effective against drive-by download exploits that we evaluated with a low ($<$ 1%) false positive rate.

Tracking the origin and provenance of critical system events by inferring and enforcing the causal relationships between them and user actions is a novel and effective approach for managing a secure host. We envision that our approach can be generalized beyond file and process activities. For example, the correlation between user actions and outbound traffic can be systematically studied for cyber security purposes.

8 Acknowledgments

The authors would like to thank the first author in [17] for sharing the sources for obtaining malicious URLs.

References

[1] A. Baliga, L. Iftode, and X. Chen. Automated containment of rootkits attacks. *Computers & Security*, 27(7-8):323–334, 2008.

[2] E. Chien. The new generation of targeted attacks, 2010. Keynote in Recent Advances in Intrusion Detection (RAID).

[3] 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.

[4] M. Cruz. Most abused infection vector, 2008. Trend Micro. `http://blog.trendmicro.com/most-abused-infection-vector/`.

[5] M. Dhawan and V. Ganapathy. Analyzing information flow in javascript-based browser extensions. In *ACSAC*, pages 382–391. IEEE Computer Society, 2009.

[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] Adebe Flash Player remote code execution. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-3456`.

[9] D. A. Freedman. *Statistical Models and Causal Inference: A Dialogue with the Social Sciences*. Cambirdge University Press, 2010.

[10] S. Gianvecchio, M. Xie, Z. Wu, and H. Wang. Measurement and classification of humans and bots in internet chat. In *Proceedings of USENIX Security Symposium*, 2008.

[11] 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.

[12] R. Gummadi, H. Balakrishnan, P. Maniatis, and S. Ratnasamy. Not-a-Bot: Improving service availability in the face of botnet attacks. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NDSI)*, 2009.

[13] HTML Object Memory Corruption Vulnerability. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-0249`.

[14] Vulnerability Note VU#493881. US-CERT. `http://www.kb.cert.org/vuls/id/493881`.

[15] 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.

[16] 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.

[17] 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.

[18] `www.malwaredomainlist.com`.

[19] `www.malwareurl.com`.

[20] A. Moshchuk, T. Bragin, and D. Deville. SpyProxy: Execution-based detection of malicious web content. In *Proceedings of the 16th USENIX Security Symposium*, 2007.

[21] Microsoft high-risk extensions. `http://support.microsoft.com/kb/883260`.

[22] A. Niki. Drive-by download attacks: Effects and detection methods. Master's thesis, Royal Holloway University of London, 2009.

[23] 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.

[24] Apple QuickTime 7.3 RTSP Response Exploit, CVE-2007-6166, `http://securityevaluators.com/content/case-studies/sl/`.

[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] A. Sotirov. Heap Feng Shui in JavaScript. `http://www.phreedom.org/research/heap-feng-shui/`.

[31] E. H. Spafford and G. Kim. The design and implementation of tripwire: A file system integrity checker. In *2d ACM Conf. on Computer and Communication Security (CCS)*, 1994.

[32] A. Srivastava and J. T. Giffin. Automatic discovery of parasitic malware. In S. Jha, R. Sommer, and C. Kreibich, editors, *RAID*, volume 6307 of *Lecture Notes in Computer Science*, pages 97–117. Springer, 2010.

[33] 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.

[34] 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.

[35] D. Stefan and D. Yao. Keystroke-dynamics authentication against synthetic forgeries. In *Proceedings of the International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)*, November 2010.

[36] B. Stone-Gross, M. Cova, L. Cavallaro, B. Gilbert, M. Szydlowski, 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.

[37] Superbuddy exploits through AOL activeX control. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-5820`.

[38] 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.

[39] 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.

[40] Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, 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.

[41] 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.

[42] 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.

A Pseudo-Code of DeWare Detection Algorithm

B Evaluation of Commercially-Available Security Products

C Accessible Area For Firefox Browser

Product	Driver Engine Version	Definition	Reaction
360 Safeguard	v3.0 360 v6.0.1 360 v6.0.1 360 v6.0.2	2008-6-16 2008-6-16 2009-10-27 2009-10-14	No detection No detection No detection Detected heap spray attack, shutdown iexplorer.exe
Zonealarm Pro	7.0.483 8.0.400 9.1.008	Anti-spyware engine 5.0.189 Anti-spyware engine 5.0.209 Anti-spyware engine 9.1.008	Captured .exe trying to access Internet; user clicked "Deny"; but .exe was still downloaded
Trend Micro Internet Security Pro	v8.952	Pattern version 6.289 Pattern version 6.587.50 2009-10-29	No detection No detection
Microsoft Security Essentials		Virus Definition 1.69.825, Spyware Definition 1.69.825 2009-11-11	Detected threats; user clicked "Clean the threat"; but .exe files were still downloaded
AVG Internet Security	v8.5.423 v9.0.707	Virus DB 270.14.20 2009-03 Virus DB 270.14.68/2507 2009-11-16	Threats detected; page blocked Detected threats; page blocked
McAfee	v8.1 Build 8.1.175	VirusScan v12.1, 2009-10-28, Personal Firewall v9.1 2009-10-28, Anti-Spam v9.1 2009-10-29	Buffer overflow attack captured
Kaspersky Internet Security	v7.0	Database 2007-10-1	Detect suspicious "Data Execution"

Table 3: Comparison of security products' effectiveness against IE 7 XML DBD attack.

```
\DocumentsandSettings\dyao\LocalSettings\Temp\
\DocumentsandSettings\dyao\ApplicationData\Mozilla\Firefox\Profiles\
\DocumentsandSettings\user\LocalSettings\ApplicationData\Mozilla\Firefox\MozillaFirefox\
\DocumentsandSettings\user\LocalSettings\ApplicationData\Mozilla\Firefox\Profiles\
\ProgramFiles\MozillaFirefox\
\DocumentsandSettings\user\ApplicationData\Macromedia\FlashPlayer\
\WINDOWS\system32\Macromed\Flash\
\DocumentsandSettings\user\ApplicationData\Adobe\FlashPlayer\
\DocumentsandSettings\user\ApplicationData\Adobe\Acrobat\
\DocumentsandSettings\AllUsers\ApplicationData\Adobe\Reader\
\ProgramFiles\CommonFiles\Adobe\
\DocumentsandSettings\AllUsers\ApplicationData\Real\
\WINDOWS\Tasks\RealUpgradeScheduledTaskS
\WINDOWS\Tasks\RealUpgradeLogonTaskS
\DocumentsandSettings\user\ApplicationData\Sun\Java\
\DocumentsandSettings\user\LocalSettings\ApplicationData\Microsoft\WindowsMedia\
\DocumentsandSettings\user\LocalSettings\TemporaryInternetFiles\
\DocumentsandSettings\user\Cookies\
```

```
Require:
File creation events filtered from I/O request packets:
C^1, C^2, C^3,… …, C^i, … …
Process creation events from system built-in security auditing:
P^1, P^2, P^3,… …, P^i, … …
Result: alert if safety policy violated

#Define
Threshhold: T
T = latency between user input and corresponding file creation
A set of high-risk file extensions: Hi-risk
Hi-risk = {.bat, .cmd, .exe, .js, … …}
Accessible area: A-area
A-area = {C:\....\Local Settings\Temp,
            C:\... \Local Settings\Temporary Internet Files, … …}
Downloadable area: D-area
D-area = {D:\My Documents\Downloads, … …}

for each C^i do
   if Path(C^i) ∈ A-area
      OK
   end if
   if Path(C^i) ∈ D-area
      if Extension(C^i) ∈ Hi-risk
         if Input-Timestamp(C^i) ⩽ T
            OK
         end if
         Generate an alert
      end if
      OK
   end if
   Generate an alert
end for
while (true) do
      if Path(P^i) ∈ A-area or Path(P^i) ∈ D-area
         Generate an alert
      end if
end while
```

Figure 6: Detection algorithm in pseudo-code.