BLADE: An Attack-Agnostic Approach for Preventing Drive-By Malware Infections

Long Lu[†] Vinod Yegneswaran[‡] Phillip Porras[‡] Wenke Lee[†]
[†]College of Computing, Georgia Institute of Technology [‡]SRI International
{long, wenke}@cc.gatech.edu {vinod, porras}@csl.sri.com

ABSTRACT

Web-based surreptitious malware infections (i.e., drive-by downloads) have become the primary method used to deliver malicious software onto computers across the Internet. To address this threat, we present a browser-independent operating system kernel extension designed to eliminate drive-by malware installations. The BLADE (Block All Drive-by download Exploits) system asserts that all executable files delivered through browser downloads must result from explicit user consent and transparently redirects every unconsented browser download into a nonexecutable *secure zone* on disk. BLADE thwarts the ability of browser-based exploits to surreptitiously download and execute malicious content by remapping to the filesystem only those browser downloads to which a programmatically inferred *user-consent* is correlated, BLADE provides its protection without explicit knowledge of any exploits and is thus resilient against code obfuscation and zero-day threats that directly contribute to the pervasiveness of today's drive-by malware. We present the design of our BLADE prototype implementation for the Microsoft Windows platform, and report results from an extensive empirical evaluation of its effectiveness on popular browsers. Our evaluation includes multiple versions of IE and Firefox, against 1,934 active malicious URLs, representing a broad spectrum of web-based exploits now plaguing the Internet. BLADE successfully blocked all drive-by malware install attempts with zero false positives and a 3% worst-case performance cost.

Categories and Subject Descriptors

D.4.6 [**OPERATING SYSTEMS**]: Security and Protection

General Terms

Security

Keywords

Malware Protection, Drive-by Download,
Unconsented-Content Execution Prevention

1. INTRODUCTION

As highlighted by the recent Aurora exploit used in the Google espionage attack [2], vulnerable web browsers and the growing plethora of their complex applets have now become the Achilles heel of enterprise network security. This problem is compounded by the increasing diversity of browser plug-in applications where a wide variety of published and zero-day browser-based exploits provide malware developers with a continual stream of infection vectors from which to disseminate malware throughout the Internet. In fact, client-side exploits now dominate the traditional push-based malware propagation strategies that were well established by Internet scanning worms and viruses [21, 27].

Surreptitious delivery exploits, which are the focus of this paper, represent a particularly insidious form of exploit, whereby the mere connection to a web server can result in the installation of malware on the client machine. Here, no user consent is required, and no symptoms of the infection may ever manifest on the victim's host. Rather, the malicious web server silently passes malicious shellcode to the victim browser, which then forces the browser to download, store, and silently execute a malicious application. The installed malware could then be used for identity theft, to exfiltrate information [19], launch denial-of-service attacks [12], or participate in botnet activity [9, 29].

Our Approach: As a significant and tangible countermeasure against such attacks, we propose BLADE, a system that effectively immunizes a host against all forms of drive-by download malware installs. A distinguishing aspect of the BLADE design is that it is both attack and browser agnostic, i.e., it neither requires exploit signatures nor changes to the browsers. Rather, BLADE relies on limited semantic knowledge about a handful of user interface (UI) elements common across web browser applications. While our implementation is focused on browser protection, the approach could be generalized to other network-capable applications (e.g., email clients, instant messengers, media players) that are subject to drive-by exploits.

BLADE's design principle, *unconsented-content execution prevention*, is motivated by a fundamental practice in the way browsers handle web stream content. That is, when receiving data content from a web server, browsers handle the content they receive in either of two basic ways: supported file types (e.g., html, jpeg, pdf) and unsupported file types (e.g., exe, zip). While browsers silently fetch and render all supported file types, they must prompt the user when an unsupported type is encountered. However, the objective of the drive-by download is to deliver malicious binaries through the browser using methods that essentially bypass the standard unsupported-type user prompt interactions. BLADE's approach is to intercept and impose execution prevention of all downloaded content that has not been directly consented to by any user-to-browser interaction. This is in contrast to sandboxing techniques [10] that do not track user interaction and permit limited execution of untrusted code in tightly controlled environments.

To prevent unconsented-content execution, BLADE introduces

CCS'10, October 4–8, 2010, Chicago, Illinois, USA.

three key operating system (OS)-level capabilities by coordinating its components in the kernel space, as shown in Figure 1. First, BLADE introduces *user-interaction tracking* as a means to collect user download authorizations expressed during interactions with browsers. BLADE captures on-screen consent-to-download dialog windows via a Screen Parser module and tracks the user's physical interactions (e.g., mouse clicks) with these dialog windows via a Hardware Event Tracer. Second, BLADE introduces *consent correlation* as a means to discern "transparent" downloads from those that involve direct user authorization. Third, BLADE uses *disk I/O redirection* to efficiently contain disk footprints of unconsented data delivered through supervised processes.[1] File content created and manipulated by supervised processes is stored in a nonexecutable location created and managed by BLADE. This secure zone represents virtual offshore storage accessible only from supervised processes as a result of the transparent disk I/O redirection. The redirection logic provides supervised processes with a modified file system view to maintain functional consistency, which renders the impression that all disk operations are carried out in their respective locations. Files in the secure zone are prevented from being loaded into the memory as executables (in the case of Windows, load operations of .exe, .dll, .sys, and etc. are disallowed). BLADE's ability to identify unconsented file content and enforce execution prevention distinguishes it from techniques such as filesystem sandboxing, where all file operations are blindly jailed while allowing the malware to execute inside the sandbox.

We have implemented a prototype version of BLADE for Windows platforms and evaluated it with browsers commonly targeted by drive-by download attacks, including multiple versions of Internet Explorer (IE) and Firefox. Our evaluation results show that BLADE accurately intercepts drive-by download infections from all the 1,934 sites we have tested, including those that are heavily obfuscated to circumvent other defense techniques. Our performance testing shows that BLADE introduces only minimal overhead on variable workloads.

Contributions: In summary, the contributions of our work include the following:

- A novel and effective model for describing and thwarting drive-by download attacks.
- Unconsented-content execution prevention as a new methodology for web-based malware prevention.
- Design and implementation of the BLADE prototype on Windows platforms.
- Comprehensive empirical evaluation of BLADE on real-world threats and benign workloads.

The remainder of the paper is organized as follows. In § 2, we discuss the BLADE approach, design objectives and challenges. In § 3, we describe the architecture and implementation of BLADE components. In § 4, we measure BLADE's performance impact and evaluate the efficacy of the system against real and obfuscated exploits. In § 5, we analyze the security of BLADE system and its limitations. In § 6, we discuss related work. Finally, § 7 concludes with a summary and discussion of potential future work.

2. THE BLADE APPROACH TO DRIVE-BY EXPLOIT DEFENSE

[1]We use the term "supervised processes" to broadly refer to processes actively monitored by BLADE, including all browsers and their child processes such as plug-ins, to which remote executions may be injected by drive-by exploits.

The web is an increasingly treacherous place. The mere act of connecting one's web browser to the wrong website may result in the installation of an application without the user's authorization or knowledge. Furthermore, attempting to limit one's browsing behavior to reputable and well-known websites is becoming a less effective strategy, as malware developers are actively infiltrating these sites to spread their malicious links [21].

To understand how BLADE defends client browsers from the current generation of drive-by exploits, we first provide a more refined explanation of how drive-by exploits operate. We can then identify the underlying common transaction performed by drive-by exploits that BLADE ultimately aims to stop.

2.1 Drive-By Exploits

A drive-by download can be described as a series of steps that the adversary performs to achieve the surreptitious download and installation of malware via the victim's browser. The goal of the drive-by exploit is to take effective, temporary control of the client web browser for the purpose of forcing it to fetch, store, and then execute a binary application (e.g., .exe, .dll, .msi, .sys) without revealing to the human user that these actions have taken place. We present the drive-by exploit strategy as a series of phases.

Shellcode injection phase: The first challenge in delivering the drive-by exploit is that of gaining temporary control of the browser. Uniformly, all drive-by exploits begin with a remote code injection, such as buffer overflow exploit against some component within the browser process, e.g., the ActiveX interpreter, a multimedia plug-in, the PDF helper object, the Flash player etc.

Shellcode execution phase: Regardless of which exploit technique is selected by the malware author, the objective of this exploit is to inject a small shellcode segment within the browser process to conduct covert binary installation (this essentially defines the attack as a drive-by exploit).

Covert binary install phase: The final phase of the drive-by exploit is the sequence of steps leading to the final, permanent infection of the client host. Here, the shellcode effectively coerces the now tainted browser into fetching a remote malware application from some remote source on the Internet, storing it within the filesystem and executing it on the victim's host.

2.2 The BLADE Threat Model, Design Objectives, and Challenges

In our threat model, an adversary conducting drive-by download attacks is allowed to hijack control of a vulnerable browser and inject remote code. BLADE assumes that this attacker should have no persistent malware deployed on the target host in advance, as otherwise the goal of the attack would have been already achieved. Specifically, there is no rootkit from the adversary installed on the system, i.e., the OS kernel is trusted. Although scenarios where the assumed attacker can remotely exploit a kernel vulnerability via a browser exist, which are out of the scope of this model, we argue that they are extremely rare and could be addressed by integrating orthogonal OS integrity protection technologies, such as hypervisor-based protections, with BLADE.

BLADE does *not* attempt to halt the drive-by exploit at the shellcode injection phase or the shellcode execution phase. Given the overwhelming diversity of browser extensions, modules, and code changes that are continually churned out by the high-paced browser development community, the task of stopping all shellcode injections is a truly daunting challenge. Even with the introduction of OS-level protections such as DEP and ASLR and browser-level sand-boxing, drive-by exploits are still succeeding [17].

Rather, BLADE incorporates a different tactic in fighting drive-by attacks. From BLADE's perspective, the drive-by download attack conducts a series of steps designed to bypass the normal user-

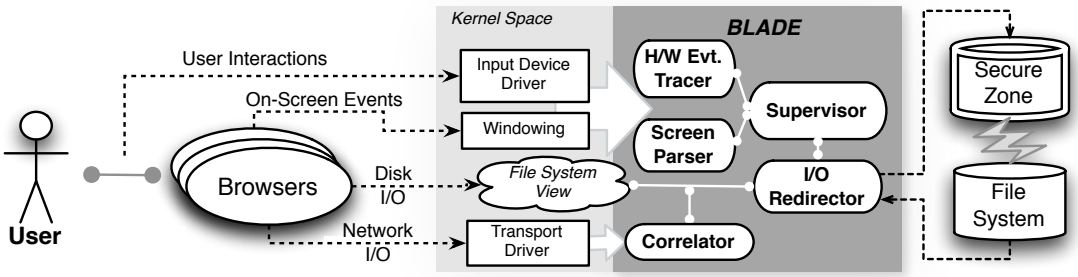


Figure 1: Overview of the BLADE system architecture

content-handling procedure that should be performed whenever a browser attempts to store this data to disk. The fetched binary itself represents an unsupported browser type that cannot be handled and rendered directly by the browser, but must be delivered through the standard user-initiated consent-to-download dialog. BLADE aims to disrupt the covert binary install phase, completely agnostic of which browser component was exploited or which shellcode injection strategy was employed to achieve the initial browser hijack.

BLADE's core mission is to foil the execution by any program entity (including the OS), of any on-disk data content received through the browser process tree, unless that content can be correlated with a user consent dialog event. BLADE enforces this requirement while not interfering with normal browser operations in any way. Specifically, we can accommodate automated software updating that is a common practice among browsers and their plug-ins through source domain whitelisting [2]. Browser native code execution mechanisms (e.g., Native Client) are not affected by BLADE since they rely on the preinstalled client, rather than the OS, to load and execute the code.

Inherent in this task are several key technical challenges, which we outline here and further cast as design goals that we directly address in this paper:

• *Real-time user authorization capture and interpretation* – BLADE must monitor user-to-browser interaction events to capture explicit user authorizations that permit upcoming download actions. From each captured authorization, BLADE must extract identity information pertaining to the expected download (i.e., remote URL, file name, and local path) in order to uniquely identify the resulting file.

• *Robust correlation between authorization and download content* – BLADE must programmatically distinguish user-initiated browser downloads from unauthorized ones and reliably correlate every authorization event with the corresponding binary stream that is downloaded by the browser from the network.

• *Stringent enforcement of execution prevention* – Files containing unauthorized download content must be stringently prevented from execution, while other types of access from supervised processes are allowed. This enforcement must not impede normal operations of browsers as well as other programs.

• *Browser agnostic enforcement* – BLADE must not depend on either the integrity of browsers or their internal handling of tasks. We must assume that new browser attack strategies will continue to evolve along with the rapid development of new browser technologies. Browser updates or potential browser compromises caused by inevitable software vulnerabilities must not affect the protection quality BLADE provides.

• *Exploit and evasion independence* – BLADE's enforcement mechanism must be entirely agnostic to exploits employed as the first step to subvert the browser into performing drive-by downloads, and thus be immune to all kinds of sophisticated evasion techniques including code obfuscations and zero-day vulnerabilities.

• *Efficient and usable system performance* – BLADE's performance impact on browser content handling must be negligible. Overall, BLADE should not impose perceptible delays to normal browser operations, and have no impact on non-browser host operations.

Considering the threat model and design trade-offs, we believe that placing BLADE as a dynamic loadable driver into the OS is a viable design choice to achieve our goals listed above. To reliably capture and interpret user interactions and guarantee unforgeability, BLADE has to reside at least as low as the OS. Even in scenarios where virtual machine monitoring systems are deployed, having BLADE inside the kernel is more efficient than placing BLADE-equivalent functionalities inside the hypervisor and more accurate than solely using virtual machine introspection.

3. THE BLADE SYSTEM ARCHITECTURE

Figure 1 illustrates the BLADE software architecture and its core components. The front-end components, including the Screen Parser, Hardware-Event Tracer and Supervisor are responsible for collecting information displayed on the screen and tracking user interactions when necessary. The Screen Parser monitors kernel windowing events as the status of on-screen UI changes in real time. It signals the Supervisor upon the appearance of a *download consent dialog* (or authorization dialog) on the screen foreground and reports necessary information parsed from the screen (see § 3.1). A download consent dialog is defined as any prompt (dialog box) created by browsers or plug-ins seeking download permission from the user. Due to the well-defined application interface used by commodity browsers to implement download confirmation dialogs, a small number of signatures (one or two per browser family) are needed to capture all download consent events. Each signature captures the external appearance and the internal hierarchy shared by all UI instances of that class. The Screen Parser uses these signatures to discover download consent dialogs, locate the respective positions of confirmation elements on these UI dialogs (e.g., the "Save" button), and extract the download identity information (e.g., URL, file name) to be used in the correlation process. Upon receiving the signal from the Screen Parser, the Supervisor invokes the Hardware-Event Tracer to intercept subsequent mouse and key-

[2]Our current prototype does not implement this capability.

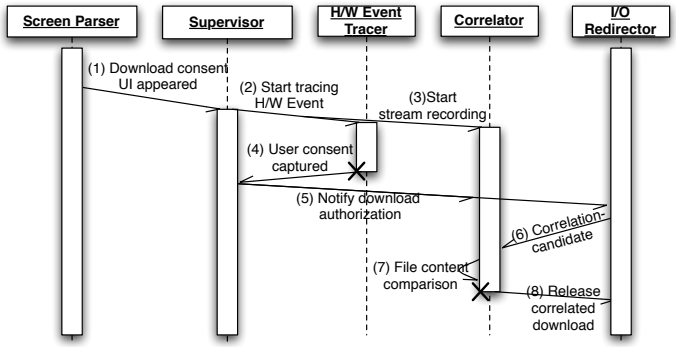


Figure 2: Download authorization workflow

board input events that would trigger the download confirmation. BLADE relies on hardware events as the only dependable source of extracting user consent information due to their unforgeability in our threat model (see § 3.3).

The Correlator and the I/O Redirector form the back end of the BLADE system. They correlate inferred authorizations from the front end with resulting downloads and enforce the nonexecution policy for downloads that are not directly requested by the user. The Correlator ensures BLADE's resistance to spoofing attacks such as forged UI dialogs (discussed in § 5), by virtue of its capability to validate the authenticity of the consequent file corresponding to a user download consent. We define the *download identity information* as $(URL, Path)$, i.e., a 2-tuple of the remote URL and the local storage path, to uniquely delegate a user download authorization. The Correlator matches a file f with a tuple (u, p) when f is saved at p with data content received from u (see § 3.4).

The I/O Redirector persistently guarantees that uncorrelated downloads can do no harm by establishing the *secure zone*. As its name suggests, the I/O Redirector intercepts disk write operations initiated from the browser process tree (namely, *supervised processes*) and redirects them to the secure zone, where execution is explicitly prohibited by blocking *memory-section synchronizations*. We describe this in more detail in § 3.5. By default all files downloaded by supervised processes are transparently redirected to the secure zone. Files that pass the download correlation process (i.e., where the content written is indeed from the user-authorized remote URL) are subsequently moved out of the secure zone back to their original destination in the file system. This move is accomplished by modifying file system metadata as opposed to copying the downloaded data, which can be finished in constant time. Our design of the secure-zone-based I/O redirection with the capability to discern user-initiated downloads enables a generic defense strategy that targets the common behavioral pattern shared by all drive-by download attacks.

We now discuss the design details of the BLADE architecture components, in the order of web download and authorization workflow as shown in Figure 2.

3.1 Screen Parser

BLADE's download authorization lifecycle is triggered by the appearance of download consent dialogs, which seek user's permission on downloads. Internally, every status change of UI elements causes a certain windowing event to be sent to the operating system, which express the change by re-drawing the screen. For instance, creating a new window causes an `OBJ_CREATE` event to be generated on Windows platforms, which contains information needed by the operating system to draw the new window on the screen (e.g., position, size, text). The Screen Parser component of BLADE relies on accurate interpretations of these windowing events intercepted from within the OS to discover download consent UI elements and effectively monitor content displayed on the screen.

Since significant performance degradation can be introduced if suboptimal methods are employed, this component merits considerable care in implementation. For example, a naive option to implement the Screen Parser is as a direct hook into windowing event handlers. However, such implementation would block the window drawing process while trying to recognize newly visible UI elements, and in turn, result in perceptible UI delays when the window being parsed contains too many elements.

To optimize performance, BLADE implants an *agent* in user space to prefilter irrelevant windowing events. It runs in parallel with the window management routines, asynchronously filtering and preparsing windowing events in the user space that would otherwise incur significant kernel CPU cycles if directly handled by the Screen Parser. The agent pipes its output to the Screen Parser, which *may* represent a user consent dialog currently in focus. To secure against interference from untrusted user-level programs, an independent sanity checker in the Screen Parser cross-validates the input from the agent by inspecting kernel memory objects representing the UI elements.

On the Windows platforms, handling only three types of events is sufficient to completely cover the real-time changes of the currently focused window: `EVENT_SYSTEM_FOREGROUND`, `EVENT_SYSTEM_MOVESIZEEND`, and `EVENT_SYSTEM_MINIMIZEEND`. Key strokes triggering a particular UI element can also be obtained as one of the associated attributes. Screen information is parsed only if the newly focused window is deemed to represent a request for download permission.

UI signatures are used to identify download consent dialogs and guide information extraction from these dialogs. Each signature describing the internal composition shared across all UI instances of a class is sufficiently general and accurate in capturing all dialogs with the same look and feel as the sample used for signature generation. Due to the uniform use of interfaces by current browsers to request download permissions, there are only a handful of UI classes that serve this purpose, which also remain highly stable across browser versions and regular updates. Hence, using only two signatures for Firefox and one signature for IE, we can successfully capture all forms of download notifications in these browser families across versions.

Note that attempted evasions by faking user consent dialogs may trigger a signature match, but cannot elude the Correlator (see § 3.4 for the correlation process).

3.2 Supervisor

As the first component loaded upon BLADE startup, the Supervisor serves the role of coordinator for carrying out all tasks of BLADE. It is charged with assigning tasks to other BLADE components and coordinating their execution, as responding to the different event notifications from the Screen Parser. The Supervisor also takes care of internal communications among all BLADE components, including user-kernel communication backed by IOCTLs (device input and output control), and kernel-kernel communication implemented by simply sharing a nonpaged pool across all kernel components as a means of information exchange. Here, spin-lock-based synchronizations are used to protect the integrity of shared data.

Upon notification of the appearance of a download consent dialog, or a status change to an existing one, the Supervisor initiates other kernel components accordingly, or resets them in response to status changes. As shown in Figure 2, when a new relevant UI element is discovered, the Hardware Event Tracer (H/W Event Tracer)

is triggered, with input information such as the on-screen locations of download consent dialogs. Its task is to sense the user-invoked hardware device signals that may indicate the user's consent to permit a pending download request. The Correlator also receives a command from the Supervisor, indicating that the corresponding stream recording process should start. A download authorization is not recognized by the Supervisor until user consent is captured by the H/W Event Tracer (in the form of physical mouse clicks or keystrokes).

The Supervisor also actively maintains a complete list of supervised processes, on which most BLADE routines rely to function correctly. For example, the I/O Redirector and the Correlator only intercept file operations and record inbound network streams of supervised processes. The list is initialized to be empty when BLADE starts. A process p will be added into the list when (a) it is a newly created browser process, (b) it is a newly created process spawned by a supervised process, or (c) a remote thread is created within the process by a supervised process. Tracking remote thread creations is critical for blocking I/O redirection evasions, which may employ a remote thread to carry out disk I/O on behalf of an unsupervised process. The consequent list of this logic covers all possible execution entities that might either initiate a legitimate browser download or be exploited to deliver surreptitious downloads. Listed processes will be removed as they are terminated. The Supervisor registers a callback routine for process creation and termination events by calling `PsSetCreateProcessNotifyRoutine`.

3.3 Hardware Event Tracer

Once a download consent dialog is identified by the Screen Parser, the next task is to interpret the user's response. We developed the Hardware Event Tracer (HET) to track user interactions with this UI element by monitoring signals generated from the hardware to the OS. Signals at this level can never be forged by attackers in our threat model; thus, BLADE is immune to attempted evasions by faking an affirmative response to user download consent events.

The HET starts with a notification from the Supervisor indicating the appearance of a certain download confirmation UI. The HET's role is to capture responses from the user's mouse clicks or keystrokes. During the tracing interval, which normally lasts a few seconds, the HET looks for any mouse click whose on-screen coordinates fall in the areas of download consent dialogs, and any keystroke that can trigger these UIs. The HET also maintains some state information in order to make accurate decisions regarding whether the intercepted hardware events could finally trigger the download consent. The HET terminates the tracing activity due to status changes from the on-screen consent dialog (e.g., minimized, unfocused).

Our current prototype implements the tracking routines only for pointer input devices, which means that users can express their consent only by using the mouse. However, adding support for keyboard input using the same principle should be straightforward. Moreover, the performance overhead introduced by the addition of keyboard tracing is expected to be minimal simply because keyboard events are less frequent than mouse events in web browsing.

3.4 Correlator

One of the key challenges in BLADE is establishing the 1-1 mapping between user download authorizations and downloaded files. The Correlator addresses this problem and ensures the authenticity of user-consented file downloads. Guaranteeing authenticity prevents potential attacks seeking to deliver a malicious download, either by prompting deceptive dialogs or subverting benign browser downloads.

Since BLADE is independent of the browser and treats it as a black box, only the external behavior of the browser (e.g., interactions with OS) is visible to it. Hence, the Correlator analyzes information available in the OS kernel, oblivious to the internal download handling of browsers. As browsers invariably rely on the OS to provide network and file system capabilities, all kernel drivers including the Correlator have the chance to peek into each transaction and retrieve a wealth of information about it. For example, network traffic incurred by a browser is fully transparent to the Correlator (at multiple kernel system levels) while it is being processed in the OS network protocol stack. Similarly, the Correlator can intercept file system access operations from the browser.

The Correlator associates a file download with a user authorization in two steps: it discovers the *correlation candidate* file, and then validates its authenticity. We now demonstrate why a file that passes these two steps while being correlated with a given user authorization is assured to be in compliance with that authorization.

Recall the tuple form of an inferred user authorization discussed earlier in this section $(\textit{URL, Path})$. Here, we use the second element, the destination path in the local storage, plus the file name as a criteria for discovering the correlation-candidate. Whenever a file has been written with the same path and name as that of a pending authorization, the Correlator marks it as a correlation candidate and starts the validation process immediately. We call it a candidate because the adversary in our threat model is able to replace the file content after fully compromising the browser.

The first element of the authorization tuple, the source domain, indicates the origin location of the file content and is used for source validation. We implement the following source validation technique based on content comparison. First, we keep a log of inbound transport-level stream for each TCP session created by supervised processes, which is later compared with the download candidate. If the content of a particular download-candidate appears in a stream log that corresponds to the source URL recorded in the authorization tuple, the candidate is validated and the correlation process completes with the candidate being correlated with the user authorization. Our content-comparison approach works even when encryption is used (e.g., HTTPS, VPN), because browsers rely on OS support to process transactions of this kind. The user-level APIs are simply wrappers for kernel functions and therefore plaintext content can always be obtained by kernel drivers prior to encryption (when sending) or after decryption (when receiving). Furthermore, browser-level compression/encoding schemes (e.g., SDCH), which only apply to web streams that are natively rendered by browsers, do not interfere with the BLADE correlation process.

Although the source validation idea is straightforward, an efficient comparison algorithm and a reliable implementation require careful consideration. From a performance perspective, the Correlator should avoid unnecessary stream logging and halt ongoing log writes once they are deemed unnecessary. Moreover, stream recording needs to be performed only when there is an incoming authorized download file and needs to consider only inbound content. Hence, a new logging process will be initiated, only when a download consent dialog requesting a download permission pops up, and only on streams sharing the same remote endpoint as the authorization dialog. A subtle issue is that the source of the download file is identified by a URL on the authorization dialog, while the remote end of a stream is identified by an IP address. The Correlator performs a domain name lookup in the local DNS cache to resolve the corresponding IP address(es). Integrity of the local DNS cache is guaranteed in our threat model because it is being maintained by a trusted kernel component. The logging terminates either when the user denies the download request or after the last stacked authorization permitting that source has been correlated with a file download. As native browser downloaders are all single-threaded, our current prototype does not support the case of multi-threaded downloads where content of a single file comes from multiple streams.

3.5 I/O Redirector

BLADE introduces the *secure zone* (i.e., a virtual storage area) as a mechanism to restrict execution of disk footprints that are caused by supervised processes but not explicitly allowed by users. Unlike sandboxing, which blindly isolates execution of untrusted code, the secure zone of BLADE is intelligent enough to *selectively contain* potential threats and ultimately prevent them.

The design philosophy of the secure zone is based on the *closure property* of browser disk writes derived from our study of generic disk access patterns by browsers. The robustness of this property was evaluated by exercising popular browsers with multiple web browsing workloads (see § 4.1 C).

Closure property: On a clean computer running commodity OS and browsers, let $P = \{p \mid p : any\ browser\ process\}$, and F, F_{auth}, F_{int} and F' be four sets of files on disk:

$$F = \{f \mid f : any\ file\ written\ by\ p,\ where\ p \in P\};$$
$$F_{auth} = \{f_a \mid f_a : any\text{-}authorized\ browser\ download\};$$
$$F_{int} = F - F_{auth} \quad (given\ F_{auth} \subset F\ is\ always\ true);$$
$$F' = \{f' \mid f' : any\ file\ opened\ by\ p',\ where\ p' \in \bar{P}\};$$

We observe that $F_{int} \cap F' \approx \emptyset$. This implies that, except for user-authorized download files (F_{auth}), any other file to which browser process (P) writes data is not normally accessed by non-browser processes ($\bar{P}$). More generally, it indicates that the disk data that is written by browsers without explicit user consent is well-contained within an implicit scope on disk, and thus should not be accessed by other processes or executed by any program entity. Discovering this scope inspired our design of the secure zone. The I/O Redirector plays a central role in managing the secure zone by enforcing the following policies:

- **P1** : Any new file created by a supervised process is redirected to the secure zone.
- **P2** : Any existing file modified by a supervised process is saved as a shadow copy in the secure zone, without change to the original file.
- **P3** : I/O redirection is transparent to supervised processes.
- **P4** : I/O redirection only applies to supervised processes. Files in the secure zone can only be accessed via redirection.
- **P5** : No execution is allowed for files in the secure zone.
- **P6** : Any file correlated with a user download authorization is remapped to the filesystem.

Together these policies enable a complete containment of disk footprints affected by content delivered through browser processes without user knowledge, while still preserving the browser usability due to transparency.

Figure 3 provides a high-level overview of how the I/O Redirector handles the two types of file accesses in order to enforce $P1$ – $P3$ listed above: the upper subfigure shows that the browser is trying to *write* `C:\a.exe` to the disk (i.e., opening a file handle with write privilege). Upon receiving the request, the I/O Redirector first checks the existence of the file's shadow copy "`\SecureZone\C\a.exe`". If it exists, i.e., the file has been previously created or modified by the supervised process, the I/O Redirector immediately forwards the request down to the file system driver with the target being modified into the path of the shadow copy. Otherwise, the I/O Redirector might need to create such a shadow copy before modifying and redirecting the request, depending on whether the request is to create a new file "`Disk1\C\a.exe`" or to modify/replace an old one. Finally, the browser that obtains the returned file handle is unaware that it is operating on a shadow copy of the file in the secure zone. The lower subfigure shows that a *read* request is redirected to the shadow copy "`\SecureZone\C\a.htm`" if it exists. Otherwise, the request is passed down to the file system without the need for redirection. The I/O Redirector also provides

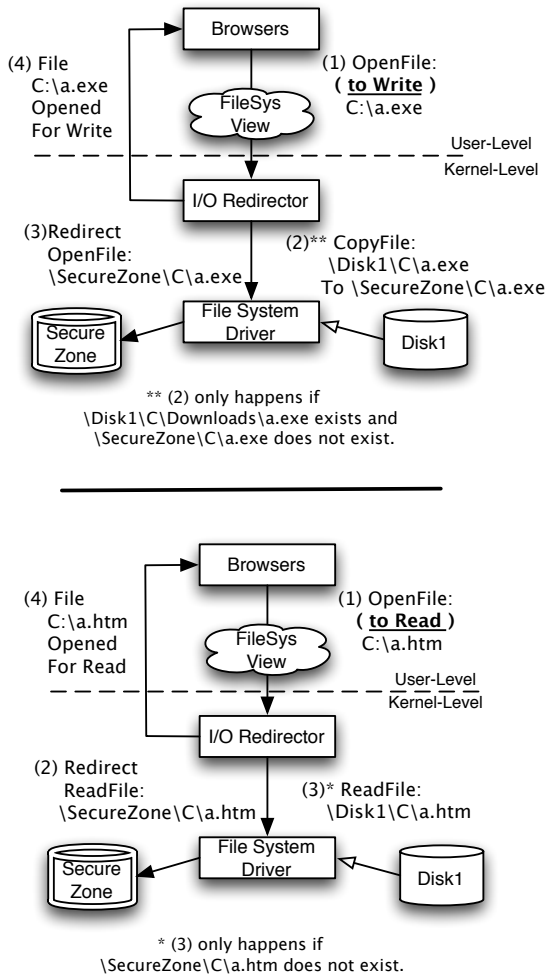


Figure 3: Browser file access (top: write; bottom: read) request processing by the I/O Redirector

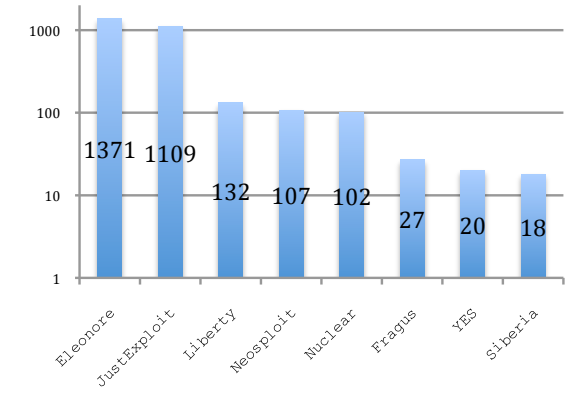
a different file system view to supervised processes, which hides the separation of files inside and outside the secure zone.

To enforce $P4$, which guarantees the nonpropagation property of files in the secure zone, the I/O Redirector simply passes through file access requests from processes that are not supervised (i.e., no redirection happens), except for denying those that are obtaining handles to files in the secure zone. The policy $P5$, file execution prevention, is performed by blocking executable images from being mapped into the memory. Specifically, the I/O Redirector intercepts `AcquireForSectionSynchronization` operations on files located in the secure zone. This is a necessary operation performed by the Windows kernel to load all forms of executables including normal program (.exe, .msi) startups, dynamic library (.dll) loads and driver module (.sys) installations.

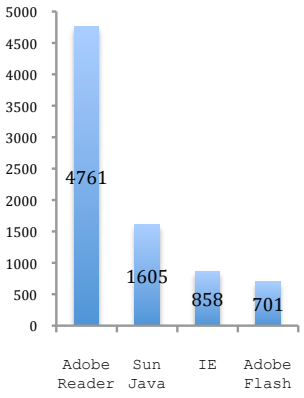
When the Correlator successfully matches a previously inferred user download authorization with a file written to the secure zone, the I/O Redirector is notified and the file is remapped back to the filesystem instantly.

4. EVALUATION

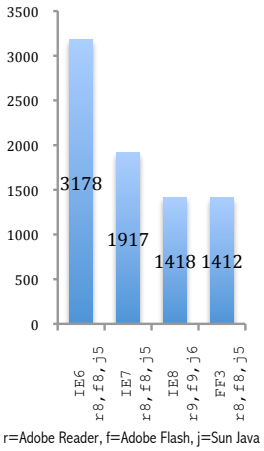
We evaluate our BLADE prototype in terms of system effectiveness and performance overhead. Our effectiveness evaluation demonstrates the solid defense that BLADE provides against real-world drive-by download attacks and its resilience to false posi-



(a) Encountered Exploit-Kit Distribution



(b) Vulnerability Distribution



(c) Attack Distribution

Figure 4: Statistics from daily malicious URL experiment

tives. The second part of the evaluation reveals that BLADE imposes a negligible overhead on the host machine.

4.1 Measuring System Effectiveness

We describe three experiments collectively designed to measure the completeness, accuracy, and overhead of BLADE's protection.

A. Empirical Daily Evaluation on Malware URL Lists: One way to demonstrate the effectiveness of a security system is to exercise it against contemporary real-world threats. Our testbed automatically harvests malware URLs from multiple whitehat mailing lists on a daily basis and evaluates BLADE against potential drive-by URLs that were reported in the past 48 hours. To validate BLADE's browser and exploit independence, each URL is tested against multiple software configurations covering different browser versions and common plug-ins.

Our evaluation platform is a VMware virtual machine running on a lightly loaded PC with a 2.0 GHz single-core CPU and 512 MB RAM. The VM runs Windows XP SP2 (without any additional security patches) and is equipped with versions of Internet Explorer, Firefox, and vulnerable browser plug-ins (e.g., PDF reader, Flash player, JVM). Each software configuration is saved as a separate VM snapshot. The testbed drives the VM to iterate through the URL list for each software configuration. Prior to visiting each URL, the testbed reverts the VM and prepares the environment wherein BLADE and independent instrumentation tools are loaded. These include `procmon` [3] to monitor Windows system call events and `tcpdump` on the host to capture packet traces. The purpose of instrumentation is to gather auxiliary evidence about potential drive-by installs for false negative evaluation. An in-VM agent then creates a browser instance and navigates it to the malicious URL. After a delay of one minute, BLADE's output and instrumentation logs are synchronously redirected outside the VM for offline post-mortem analysis.

We parse the instrumentation log to identify specific events that we then interpret in evaluating several potential experimental outcomes. In particular, we track three key experiment outcomes:

- $\mathbf{C_1}$:$(T|F)$ *URL test session caused a BLADE alert*
- $\mathbf{C_2}$:$(T|F)$ *URL test session attempted to load/execute a file from the secure zone*
- $\mathbf{C_3}$:$(T|F)$ *URL test session produced a file write outside the secure zone*

C_1 can be confirmed from BLADE's alert log. System call event logs, especially events related to filesystem and process management, can help identify occurrences of C_2 and C_3. The testbed then derives the following *evaluation metrics*, which are defined as logic combinations of *conditional* occurrences.

True Positive	:=	$C_1 \wedge (C_2 \vee C_3)$
False Negative	:=	$\overline{C_1} \wedge (C_2 \vee C_3)$
False Positive	:=	$C_1 \wedge \overline{(C_2 \vee C_3)}$
True Negative	:=	$\overline{C_1} \wedge \overline{(C_2 \vee C_3)}$

The testbed has been operational for 3 months and visited 3,992 unique malicious URLs (please see `www.blade-defender.org/eval-lab` for updated statistics and results). The dataset that was collected also offers a glimpse into the Internet's contemporary drive-by malware landscape as summarized in Figure 4. Figure 4 (a) shows the distribution of exploit kits encountered during our experiments illustrating the growing popularity of commercialized exploit kits. Eleonore and JustExploit seem to be the most popular. Figure 4 (b) shows the distribution of attacks by vulnerable software. We find that (i) pdf exploits currently dominate, (ii) attackers increasingly prefer targeting plug-ins over browsers because of the wider attack surface, and (iii) they largely rely on commercialized exploit kits to launch reliable attacks.

Figure 4 (c) displays the distribution of successful attacks based on browser and software configuration. Not surprisingly, we find that all tested browsers are vulnerable. We find that the Internet Explorer 6.0 system configured with Adobe Reader 8.0, Adobe Flash 8.0, and JVM 5.0 is the most vulnerable. A similarly configured Firefox 3.0 system experiences less than half the number of exploits and is comparable to Internet Explorer 8 running Adobe Reader 9.0 and JVM 6.0. Table 1 summarizes results from our daily evaluation. The number of trials is more than the number of unique URLs because each URL might appear on the list for multiple days and is tested on multiple VM configurations. As shown in Table 1(a), BLADE was successful at blocking all 7,925 attempted drive-by malware installs while generating zero false alarms. Furthermore, all downloaded malicious binaries were safely quarantined into the secure zone. While these results might be surprising at first glance, they are expected because BLADE is designed in an exploit oblivious manner. It is worth noting that at no point did our system design or implementation necessitate additional tuning to handle a new exploit or shellcode type. Table 1(b) provides a summary of the malware binaries captured. The 7,925 trials pushed 9,745 binaries (certain sites push more than one binary) which included 8,126 EXEs and 1,619 DLLs. The average detection rate of these binaries from `virustotal.com` was only 28.43%.

Only about half of the malicious URLs were observed to be delivering drive-by download attacks when tested. While we do not

(a) Evaluation Metrics

	Total	True Positive	False Positive	True Negative	False Negative
Trials	18896	7925	0	10971	0
URLs	3992	1934	0	2058	0

(b) Dropped File Statistics

Malware Intercepted	File Types		AV coverage Day-0 (VT rate)
	EXE	DLL	
9745	8126	1619	28.43%

Table 1: Results from daily malicious URL experiment

know the exact reason why attacks fail in each instance, they include the following: (a) malicious sites that have been cleaned up, (b) misclassified sites (e.g., phishing sites) that do not attempt surreptitious drive-by downloads, (c) sites that employ IP tracking to blacklist repeated visitors, and (d) sites that target vulnerabilities not present in our configuration.

B. In Situ Attack Coverage Evaluation: The first experiment demonstrates BLADE's effectiveness against thousands of drive-by download attacks in the wild. However, it is possible that attacks in the wild are dominated by a few exploit kits and exercise only a limited set of common exploits. To compensate for this potential limitation, we conducted a second experiment that specifically evaluates BLADE against a wider set of hand-crafted attacks and more browser versions. Specifically, this customized attack set is composed of diverse shellcodes and exploits targeting several vulnerabilities in browser/plug-in software including 11 recently disclosed zero-day exploits listed in Table 2. In each case, BLADE successfully prevented the execution of the drive-by exploit binary, reaffirming our design premise that BLADE delivers complete and accurate protection in a browser-agnostic and exploit-oblivious manner.

C. Benign Website Evaluation: We evaluate BLADE's effectiveness on benign web sites, i.e., the false positive rate. For BLADE, a false positive implies that the execution of a legitimate (authorized) executable download is blocked by BLADE. Under BLADE's design, there are two potential reasons why an authorized executable download may be inadvertently hindered by BLADE: (i) the user's authorization cannot be inferred, which leaves the resulting download in the secure zone as untrusted; (ii) a legitimate browser download seeks to execute benign logic without the user's consent, which represents a violation of our root assumption. Thus, we tried to create a workload that might trigger (i) or (ii).

To address (i), we first tested the signature coverage of download consent dialogs for each browser by looking for an unknown method for requesting download consent. We downloaded 30 different software applications from 15 highly ranked freeware sites, with varying file types (.exe, .zip, .msi etc.). We also checked whether download consent UIs can be reliably discovered when noise is introduced onto the screen. Neither of the above two test cases revealed any false positives. We used a stress-testing-based strategy to create a workload that could lead to false positives incurred from (ii). By manually visiting a URL pool, including the top 5 highly ranked websites from 16 categories [1], we verified that BLADE does not disrupt normal browser interactions with these benign sites.

4.2 Performance Overhead

BLADE's deployment target is the average Internet Windows PC, where it is expected to protect such systems from drive-by attacks while imposing negligible delays on the web surfer. Performance issues have been considered at almost every stage during the design and implementation of BLADE. Here, we discuss the issue of BLADE's performance impact by first measuring the delay certain components caused to related operations on the host computer. We then evaluate the overall performance overhead incurred by BLADE as a whole.

ID	Exploit CVE-ID	Browser	Exploit Payload	Detected By Blade	Vuln. Notes
1	2006-3677	Firefox 1.5	Remote_shell_bind	YES	window.navigator
2	2005-1476	Firefox 1.5	Download_exec	YES	InstallTrigger.install()
3	2007-0038	Firefox 2.0	Download_exec	YES	LoadAniIcon()
			Dll_injection	YES	
4	2009-2477	Firefox 3.5	Download_exec	YES	**TraceMonkey 0-day**
			Dll_injection	YES	
5	N/A	Firefox 3.5	Download_exec	YES	**Wings3D 0-day**
6	2009-0927	Firefox 3.5	Remote_shell_bind	YES	Collab.getIcon()
			Dll_injection	YES	
7	2010-1028	Firefox 3.5	Download_exec	YES	**Font Overflow 0-day**
8	2010-1082	Firefox 3.5	Download_exec	YES	**XML Overflow 0-day**
9	2007-0038	IE 6.0.2900	Download_exec	YES	LoadAniIcon()
			Dll_injection	YES	
10	2006-4777	IE 6.0.2900	Download_exec	YES	KeyFrame()
11	2006-3730	IE 6.0.2900	Dll_injection	YES	setSlice()
12	2009-0075	IE 7.0.5730	Remote_shell_bind	YES	CFunctionPointer
13	2009-1539	IE 6.0.2900	Download_exec	YES	**DirectShow/ quartz 0-day**
14	2009-3672	IE 7.0.5730	Remote_shell_bind	YES	**IE Style-object 0-day**
			Download_exec	YES	
15	2008-0015	IE 7.0.5730	Download_exec	YES	**DirectShow/ATL 0-day**
16	2010-0249	IE 6.0.2900	Download_exec	YES	**IE Aurora 0-day**
17	2010-0886	IE 7.0.5730	Download_exec	YES	**JDK/JRE 6 0-day**
18	2010-0806	IE 7.0.5730	Download_exec	YES	**IEpeers 0-day**
19	2009-4324	IE 8.0.7600	Download_exec	YES	**PDF/ newplayer() 0-day**
			DLL_inject	YES	
			Remote_shell_bind	YES	

Table 2: Test results on targeted attacks and 0-days

Micro performance evaluation: These per-component tests are designed not only to individually measure performance overhead of each component, but also to mitigate factors that might affect measurements such as variable network delay. Among BLADE components, the Screen Parser, the I/O Redirector, and the Correlator are the only three that introduce measurable delays. We evaluate the contributions of each of them below.

Although the Screen Parser is designed to be asynchronous there is a chance of delay while it is matching download consent UI signatures with each appearance of a new UI element. In our tests even the worst-case matching time was not measurable, i.e., less than a millisecond.

To accurately examine the worst-case delay introduced by the I/O Redirector into file system accesses, we conducted a test as follows. We chose three files of varying sizes (1 MB, 10 MB and 100 MB) and copied them from one location to another within the same disk. Each file was copied twice, once with and once without the I/O Redirector turned on. To completely avoid other effects (e.g., file system cache, in-memory cache), we reverted to a clean VM snapshot before beginning each test. The delay and copy times for each file are shown in Table 3 which varies from 1 ms (for 1 MB files) to 7 ms (for 100 MB files). The delays are minimal because the redirector intervenes only in the process of requesting new file handles and does not intercept the disk write operations.

We also conducted performance testing on the Correlator, which

File Size (MB)	Copy Time (sec) w/ Redirection	Copy Time (sec) w/o Redirection	Delay
0.98	0.024	0.025	**4.00%**
9.23	0.243	0.246	**1.21%**
94.66	2.631	2.638	**0.27%**

Trace Size (MB)	File Size (MB)	Time (Sec)
1.12	0.98	**0.026**
9.45	9.23	**0.221**
95.83	94.66	**2.400**

Table 3: Micro evaluation results: I/O Redirector delay (left) and file comparison overhead (right)

puts each correlation candidate on hold until a conclusion is reached. Specifically, we test the running time of the content-based comparison process, which is exactly the period a legitimate user download must wait to be correlated. We performed the test on three downloaded files of different size. Running times for file sizes, calculated using timestamps that were output by the Correlator are summarized in Table 3. For downloads less than 10 MBs in size, the correlation process could finish within 1 second. We believe that such delays will not be perceptible to end users. In the case of huge files (larger than 100 MBs), the observed delays are on the order of several seconds. Given that the download itself is on the order of several minutes, we believe that the few seconds delay will go unnoticed in most (if not all) situations.

Macro performance evaluation: We created two different test cases to measure the average overhead imposed by BLADE on web-surfing workloads.

• *Browser rendering delays:* We used a JavaScript tool for testing rendering time of browsers provided by [4] with the objective of demonstrating that BLADE imposes negligible performance overhead to supervised browsers. To avoid the impact of variable network delays, we downloaded the JavaScript testing tool to the local disk before opening it with the browser. Once the page is fully loaded, a rendering time is calculated and displayed by the tool. We repeated the test 10 times for each browser, by reverting to a fresh snapshot after each test and recorded the average rendering time shown in Table 4. Overall, only slight increases in browser rendering times were observed, as only the Screen Parser and I/O Redirector are active when there are no pending download authorizations.

• *Authorized download delays:* To evaluate the worst-case performance overhead, we designed a second test case, where delays on authorized file downloads were measured. We define the file download delay as the period starting from when the user responds to the confirmation dialog box, to the point that the file is completely written to the disk. All the downloaded files were hosted in another computer on the same local area network, and delays shown in Table 4 hovered over 3% for small files and under 3% for large files.

5. SECURITY ANALYSIS

In this section, we analyze the soundness of the BLADE system design by discussing various attacks that knowledgeable adversaries may pursue to circumvent BLADE. For each attack strategy, we identify the countermeasures that have been incorporated into the BLADE design to address these threats. We then enumerate several limitations in BLADE's threat coverage.

5.1 Attacks and Built-in Countermeasures

Assuming that BLADE gains a high degree of user acceptance, we expect that malware publishers will acquire BLADE and plot circumvention strategies that either involve dropping the malware outside the secure zone or executing it while it is being quarantined. To this end, we see three avenues that future adversaries may explore to evade BLADE and surreptitiously install the malware. Countermeasures for these attacks have already been integrated into BLADE.

Spoofing attacks – The attacker may attempt to drop malware directly onto the local file system, without being redirected to the secure zone. To accomplish this, the attacker must (a) fool BLADE by forging a fake download consent dialog and the user response, or (b) fool user and the BLADE Screen Parser by spoofing browser GUI to display rogue download confirmations [11]. **Countermeasures:** We address (a) by ensuring that the user authorization inference is based on real hardware events that cannot be spoofed at the application layer. The BLADE Correlator (see § 3.4) eliminates the possibility of (b) by taking additional steps to validate the origins of user consented downloads. Although the attacker can launch a denial-of-service attack by disabling the user-level Screen Parser, this action will not lead to the surreptitious infection of the browser's host.

Download injection and process hijacking attacks – The attacker may attempt to move a downloaded malware instance out of the secure zone and then execute it. Having control of the browser process, the attacker could replace an authorized download with malware. The attacker may also hijack an unsupervised process whose file I/O is not redirected, e.g., by creating a remote thread within an unsupervised process. **Countermeasures:** Content-based correlation guarantees the source of authorized downloads and prevents download injection attacks. The BLADE Supervisor follows process creations (i.e., child processes of the browser) and remote thread activations (sibling) to ensure that unauthorized file writes from the supervised processes are appropriately redirected to the secure zone.

Coercing attacks – The attacker may attempt to coerce the operating system to execute the malware directly from the secure zone. **Countermeasure:** Since I/O redirection is implemented and enforced by a kernel driver, we believe that such attacks should be infeasible by design. If one asserts that the malware publisher may have control logic buried within the kernel to halt BLADE's operation, then we argue that the drive-by download attack is entirely unnecessary.

5.2 Limitations

While we believe BLADE represents an effective service for stopping surreptitious drive-by installations of malware, we recognize that it does not provide complete coverage of all threats web users are facing. First, BLADE does not prevent social engineering attacks where the user authorizes the download and installation of malicious binaries disguised as benign applications. Second, BLADE does not prevent in-memory execution of transient malware, which could be scripts such as JavaScript bots or x86 code inserted into memory by exploits. While such attacks are out of scope for our system, the latter attacks could be prevented by orthogonal protection techniques, such as DEP. Third, BLADE is dependent on explicit download-consent UI, which is optional (can be disabled by the user) in certain browsers. Users wishing to use BLADE to protect their web surfing activities must enable download confirmations on their browsers. Finally, BLADE is effective only against binary executables and does not prevent the download and installation of interpreted scripts. However, the overwhelming majority of current drive-by download malware is delivered as binaries. At a minimum, our system raises the bar by rendering the prevalent drive-by download threat obsolete. We intend to explore ways to stop the installation of malicious scripts in the future.

6. RELATED WORK

Browser	Time (sec) w/o BLADE	Time (sec) w/ BLADE	Delay
Firefox 3.5	3.531	3.563	**0.91%**
IE 7.0	4.328	4.401	**1.69%**
IE 8.0	4.028	4.733	**1.18%**

File Size (MB)	Time (sec) w/o BLADE	Time (sec) w/ BLADE	Delay
0.98	2.134	2.201	**3.14%**
9.23	33.201	33.879	**2.04%**
94.66	313.443	316.003	**0.81%**

Table 4: Macro-evaluation results: Browser rendering (left) and authorized download (right) times

We discuss prior measurement studies that inform the design of BLADE and distinguish it from existing URL analysis services, malware defense systems, and browser-based protections.

Internet measurement studies: The problem of drive-by downloads, particularly those resulting in malware installations on unsuspecting victims, has attracted considerable attention from researchers. In 2005, Moshchuk et al. [23] studied the threats, distribution, and evolution of spyware through an examination of more than 18 million URLs, finding scripted drive-by downloads in 5.9% of the pages visited. Seifert et al. [32] examined the prevalence and distribution of malicious web servers using the Capture-HPC client honeypot, identifying more than 300 malicious sites. In [28], Provos et al. provided a detailed dissection of the sophisticated methodology employed by the blackhats and the steps involved in executing a typical drive-by download exploit of a system. In a subsequent study [27], the authors provided extensive quantitative measurements of the global prevalence and distribution of the parties (landing sites, redirection sites and script hosting sites) involved in drive-by downloads by examining billions of URLs in the Google web archive. These studies underscore the significance of the drive-by download malware problem and motivate development of the BLADE system.

Website survey systems and proxy services: Blacklist services such as `stopbadware.org` [6] provide alerts on malicious software systems and websites, currently listing more than 392,000 malicious sites. Strider HoneyMonkey [35] and phoneyc [24] crawl the Internet looking for websites that host malicious code. While the former approach uses Virtual Machines running different operating systems and patch levels, the latter is a lightweight low-interaction system that emulates browser execution of JavaScript.

SpyBye [26] operates as a proxy server and uses simple rules to classify a URL into three categories: harmless, unknown, or dangerous. The classification process can be error prone and is meant to be a tool for webmasters to track the security for sites that they administer. The SecureBrowing software plug-in developed by Finjan [5] scans web pages in real time for viruses and malware. While the details of their detection methodology are proprietary, it is presumed to be a combination of attack signatures and URL blacklists. The BrowserShield [31] proxy system uses script rewriting and vulnerability-driven filtering to transform inbound web pages into safe equivalents by disabling execution of malicious JavaScript and VBScript exploits at runtime. Wepawet is an online submission service for detecting and analyzing malicious URLs with the capability of analyzing exploits in Flash, JavaScript, and PDF files [14]. Unlike these approaches, BLADE does not require attack signatures and is effective against zero-day attacks.

SpyProxy [22] is an execution-based malware detection proxy system, that executes active web content in a virtual machine environment before it reaches the browser. A limitation is that protection is guaranteed only when the host machine and the proxy machine maintain the identical software configuration.

Network- and host-based malware defense systems: Systems such as BotHunter [15] and BotSniffer [16] are meant to detect infected enterprise systems based on post-infection network dialog, but do not prevent the execution of malware. AntiVirus systems [7] and services like CloudAV [25], which attempt to block the execution of malware, are limited by the reliance on binary signatures. For drive-by attacks, BLADE addresses the limitations of these approaches, i.e., it acts like an IPS that thwarts the execution of malware and does not rely on signatures.

Egele et al. [13] proposed the use of x86 emulation techniques to defend browsers against a specific type of drive-by download attack, i.e., heap-spraying code injection attacks. Their objective is similar to that of NOZZLE [30], which uses static analysis of objects in the heap to detect heap-spraying attacks. BLADE differs from these systems in that it does not detect the attack, but rather prevents the execution of the malware. Our approach has the benefit that it defends against all forms of web-based surreptitious-download exploits, including malware installed using heap-spraying code injection attacks.

Sandboxing/Isolation systems: Solitude [18] and Alcatraz [20] are two systems that limit the effects of attacks by providing support for application-level isolation recovery. Secure browsers [10, 34] have been developed that use sandbox techniques to prevent malware installations. The Chromium sandbox [10] attempts to mitigate browser exploits by separating the trusted browser kernel (which runs with high-privilege outside the sandbox) and untrusted rendering engine. However, the presence of published client-side exploits for Chrome validates that such strategies are not a panacea. Recently, Barth et al. proposed a browser extension system that uses privilege separation and isolation to limit the impact of untrusted extensions [8]. The Polaris system uses the principle of least authority to restrict the impact of running untrusted applications [33]. BLADE's unconsented-content execution prevention is a similar concept to sandboxing. However, BLADE fully prevents the binary execution from occurring, rather than imposing privilege limitations, and it is significantly more transparent in how it uses user-dialog confirmation to auto-remap user-initiated downloads. More significantly, unlike secure browser frameworks that require the adoption of an entirely new browser, BLADE security protections can be deployed underneath the wide range of current and legacy Internet browsers.

7. CONCLUSION AND FUTURE WORK

We introduced the BLADE system as a new approach to immunizing vulnerable Windows hosts from surreptitious drive-by download infections. The BLADE system incorporates a kernel module to track all browser-to-human interactions, and then uses this information to distinguish consented web-based binary downloads from those cases where covert binary installations are performed. In the former case, the user-consented binaries are transparently remapped to the filesystem, and BLADE imposes no perceptible runtime behavioral changes or performance impacts on the browser. In the latter case, BLADE isolates and reports the malicious link and binary to the user, and unlike traditional sandboxes these malicious binaries are never executed.

We presented results from an ongoing evaluation of BLADE against thousands of active drive-by exploits currently plaguing the Internet (our evaluation results are unfiltered, auto-generated, and posted publicly to www.blade-defender.org). To date, BLADE's interception logic has demonstrated 100% effectiveness in preventing covert binary installations using the most widely deployed browsers on the Internet. Furthermore, over the past six months we have tested BLADE against the newest 0-day drive-by exploit attacks within days of their release and none have circumvented BLADE.

In our next phase, we plan to extend BLADE support to other network-capable applications subject to drive-by download attacks.

8. ACKNOWLEDGEMENT

The authors would like to thank Ashish Gehani and the anonymous reviewers for helpful comments on earlier versions of the paper. This material is based upon work supported in part by the National Science Foundation under grant no. 0831300, the Army Research Office under Cyber-TA Grant no. W911NF-06-1-0316, the Department of Homeland Security under contract no. FA8750-08-2-0141, the Office of Naval Research under grants no. N000140710907 and no. N000140911042. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily refect the views of the National Science Foundation, the Army Research Office, the Department of Homeland Security, or the Offce of Naval Research.

9. REFERENCES

[1] Alexa - Top Sites By Category. http://www.alexa.com/topsites/category.

[2] Microsoft Security Bulletin MS10-002 - Critical. http://www.microsoft.com/technet/security/bulletin/MS10-002.mspx.

[3] Process Monitor. http://technet.microsoft.com/en-us/sysinternals/bb896645.aspx.

[4] Testing rendering time. http://scragz.com/archived/mozilla/test-rendering-time.

[5] finjan: securing your web. http://www.finjan.com, 2009.

[6] stopbadware.org. http://www.stopbadware.org, 2009.

[7] Symantec inc. http://www.symantec.com, 2009.

[8] B. Adam, P. F. Adrienne, S. Prateek, and B. Aaron. Protecting browsers from extension vulnerabilities. In *Network and Distributed System Security Symposium (NDSS)*, 2010.

[9] P. Barford and V. Yegneswaran. An inside look at botnets. Special Workshop on Malware Detection, Advances in Information Security, Springer Verlag, 2006.

[10] A. Barth, C. Jackson, C. Reis, and T. G. C. Team. The Security Architecture of the Chromium Browser. In *Stanford Technical Report*, 2008.

[11] S. Chen, J. Meseguer, R. Sasse, H. J. Wang, and Y.-M. Wang. A systematic approach to uncover security flaws in gui logic. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2007.

[12] S. Dietrich, N. Long, and D. Dittrich. Analyzing distributed denial of service tools: The Shaft Case. In *Proceedings of the USENIX System Adminstrator's Conference, LISA*, 2000.

[13] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *Proceedings of Detection of Intrusions and Malware and Vulnerabilility Assessment (DIMVA)*, 2009.

[14] S. Ford, M. Cova, C. Kruegel, and G. Vigna. Wepawet. http://wepawet.cs.ucsb.edu, 2009.

[15] G. Gu, P. Porras, V. Yegneswaran, M. Fong, and W. Lee. BotHunter: Detecting malware infection through IDS-driven dialog correlation. In *Proceedings of 16th USENIX Security Symposium*, 2007.

[16] G. Gu, J. Zhang, and W. Lee. Botsniffer: Detecting botnet command and control channels in network traffic. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS)*, 2008.

[17] K. J. Higgins. 'Aurora' exploit retooled to bypass Internet Explorer's DEP security. http://www.darkreading.com/security/-vulnerabilities/showArticle.jhtml?articleID=222301436.

[18] S. Jain, F. Shafique, V. Djeric, and A. Goel. Application-level isolation and recovery with solitude. In *Proceedings of ACM EuroSys*, 2008.

[19] B. Krebs. Clamping down the Clampi trojan. http://voices.washingtonpost.com/securityfix/2009/09/-clamping_down_on_clampi.html.

[20] Z. Liang, V. N. Venkatakrishnan, and R. Sekar. Isolated program execution: An application transparent approach for executing untrusted programs.

[21] A. Martinez-Cabrera. Malware infections double on web pages. http://articles.sfgate.com/2010-01-26/business/-17836038_1_malware-infected-sites.

[22] A. Moshchuk, T. Bragin, D. Deville, S. D. Gribble, and H. M. Levy. SpyProxy: Execution-based detection of malicious web content. In *Proceedings of 16th USENIX Security Symposium*, 2007.

[23] A. Moshchuk, T. Bragin, S. D. Gribble, and H. M. Levy. A crawler-based study of spyware on the web. In *Network and Distributed System Security Symposium*, February 2006.

[24] J. Nazario. phoneyc: A Virtual Client Honeyport. In *Proceedings of LEET*, 2009.

[25] J. Oberheide, E. Cooke, and F. Jahanian. Cloudav: N-version antivirus in the network cloud. In *Proceedings of 17th USENIX Security Symposium*, 2008.

[26] N. Provos. Spybye - finding malware. http://www.monkey.org/p̃rovos/spybye/, 2009.

[27] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. All your iframes point to us. In *Proceedings of the 17th USENIX Security Symposium*, 2008.

[28] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. The ghost in the browser analysis of web-based malware. In *1st Workshop on Hot Topics in Understanding Botnets*, 2007.

[29] M. Rajab, J. Zarfoss, F. Monrose, and A. Terzis. A multi-faceted approach to understanding the botnet phenomenon. In *Proceedings of ACM SIGCOMM/USENIX Internet Measurement Conference*, October 2006.

[30] P. Ratanaworabhan, B. Livshits, and B. Zorn. NOZZLE: A defense against heap-spraying code injection attacks. In *Proceedings of 18th USENIX Security Symposium*, 2009.

[31] C. Reis, J. Dunagan, H. Wang, O. Dubrovsky, and S. Esmeir. Browsershield: Vulnerability driven filtering of dynamic html. In *Proceedings of OSDI*, 2006.

[32] C. Seifert, R. Steenson, T. Holtz, B. Yuan, and M. A. Davis. Know your enemy: Malicious web servers. http://www.honeynet.org/papers/mws/, 2007.

[33] M. Stiegler, A. Karp, K. Yee, T. Close, and M. Miller. Polaris: virus-safe computing for Windows XP. *Communications of the ACM*, 49(9):88, 2006.

[34] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The multi-principal construction of the Gazelle web browser. In *Proceedings of the 18th Usenix Security Symposium*, 2009.

[35] 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 *Network and Distributed System Security Symposium (NDSS)*, 2006.