A Dynamic Browser Containment Environment for Countering Web-based Malware

Octavian Purdila [*] Andreas Terzis[†] [*]

Abstract

During the last few years we have experienced a rise in malware that use so called *drive-by* web downloads to infect end-hosts. In response, several research efforts have proposed host-based mechanisms to prevent such attacks or to minimize their impact. These mechanisms sandbox the browser either through virtual machines (VMs) or via system call interposition. However, the effectiveness and usability of these techniques depend on the policies set to control either the browser's system calls or the VM's access to the host environment and the network. In this paper we present the first, as far as we know, interposition technique that *dynamically modifies* the policy ruleset to allow only system calls that originate from user requests while denying all other system calls. We do so by intercepting user GUI actions, parsing the contents of web pages downloaded by the user, and tracking the application state, safely, through stack tracing. Our evaluation results show that the proposed technique introduces limited overhead; 11.8% increase in latency and 13.4% decrease in throughput. Moreover, it generates no false positives and contains attacks against the browser itself as well as attacks on plugins and libraries used by the browser.

1 Introduction

Over the last few years we have experienced a rise in the number of attacks that exploit vulnerabilities in web browsers and their associated plugins and libraries to install malware on users' desktops [20, 27]. Attackers have innovated in this way because web-based attacks enable them to infect otherwise unreachable users (*e.g.* users behind firewalls that block external scans). In more detail, a web-based attack works by including one or more exploits in webpages that users visit. Once downloaded, the exploit code attacks vulnerabilities in the browser itself or plugins and libraries that the browser uses. In most cases, a successful exploit results in the automatic installation of a malware binary, also called *drive-by download*. The installed malware often enables an attacker to gain remote control over the compromised computer system and can be used to steal sensitive information such as banking passwords, to send out spam or to install more malicious executables over time. In response to these attacks a number of research efforts have focused on developing sandboxes—frameworks that confine the browser from the rest of the system. Depending on how the sandboxes are implemented, we broadly divide them into virtual machine (VM) based (*e.g.* [3]) and those that use interposition to intercept the browser's system calls [6, 7].

We present a system that uses system call interposition to confine the browser so that even if it is compromised it cannot cause any harm to the user's data or to other networked end-hosts. The default policy of the containment system is to deny the browser (and any threads or processes that it spawns) all access to the local filesystem and the network. The system then allow access to resources that are explicitly requested by the user. It does so, by tracking the user's GUI requests (*e.g.* typing a URL, or opening the "Save As" dialog box) to determine whether the corresponding `connect()` or `open()` system calls should be allowed. Moreover, it parses the HTML pages that the browser downloads to find references to embedded links (*e.g.* images, frames, etc.) and subsequently allows the browser to download them. However, HTML parsing cannot detect embedded links in SSL-encrypted pages or pages that use Javascript because the browser transforms the pages after downloading them. In order to allow such links to successfully load, we implement *fine grain application tracing* in which we block and investigate the state of the browser after it decrypts and/or interprets the page. The same technique is used to allow browser plugins to operate correctly. Finally, we implement *stack trace based authentication* by which we determine whether a system call should be allowed not only based on the resource requested but also on the call stack that led to that system call. In this way we can identify system calls initiated by malicious code that exploited the browser.

Our evaluation results, based on a prototype implementation using Linux and Firefox, show that the proposed approach introduces limited overhead, $\sim$ 12% increase in download latency and 13.4% decrease in throughput. Because all sensitive system calls are denied, unless explicitly allowed by one of the tracing mechanisms described above, the proposed approach contains a variety of attacks, including attacks to the browser's plugin(s) as we

[*]Department of Computer Science, Politehnica University of Bucharest

[†]Computer Science Department, The Johns Hopkins University

show in Section 6.2. At the same time, the proposed approach generates no false positives (*i.e.*, denied resource requests) as we show by crawling through a large number of web pages.

This paper has eight sections. We review related work in Section 2 and outline the proposed approach in Section 3. Section 4 describes the system's components and the necessary policies to support normal browser operation. In Section 5 we elaborate on the techniques developed to track the user's requests and the browser's state. We evaluate our approach in Section 6 and discuss its limitations in Section 7. Finally we conclude in Section 8 with a summary and directions for future work.

2 Related Work

Two general approaches have been proposed for tackling web-based malware: the ones that try to detect and prevent web browser vulnerabilities [13, 18, 21] and the ones that try to contain the browser so that even if it is compromised, it can not compromise the rest of the user's environment or interact with the outside world [3, 6, 7, 12].

Browser containment can occur at two levels: at the virtual machine (VM) level (*e.g.* Tahoma [3]) and at the operating system level. VM-based containment has the advantage of not being susceptible to kernel vulnerabilities, but on the other hand it has a high performance overhead [3]. Moreover, because the browser is isolated inside the VM, its interaction with the host operating system is constrained, which leads to poor interaction with the user. For example, documents that the browser downloads are stored into ”bins” from where the user has to copy them using special tools. Finally, running the browser inside a VM does not prevent it from accessing other hosts over the network. For this reason Tahoma uses per ”web application” manifests which explicitly (and statically) state the network services and locations that each web application is allowed to use [3].

Instead, we contain the browser at the operating system level. That is, we propose running the browser natively and relying on operating system mechanisms to control its access to the user's environment and the outside world. An example of such containment is the IE7 protected mode in Windows Vista [2]. When activated, the browser's process is degraded to a lower security level, denying most writes to the local filesystem and registry. However, our work differs significantly from this approach, both in scope and implementation. Our system completely denies access to the local filesystem (be it for reading or writing) and access to remote hosts, only permitting requests which are the results of the user's implicit or explicit actions. AppArmor is a Linux-based mechanism for controlling the resources an application can access [9]. Specifically, AppArmor allows system administrators to restrict the capabilities of a program by defining a profile. It has a learning mode, in which profile violations are logged and subsequently added to a profile manually. However, AppArmor can not alter the policy dynamically as our approach can.

A number of commercial systems provide OS-level sandboxes [8, 15, 17, 25]. Such systems are mostly used to isolate user-level servers from the rest of the system but can also be used to isolate web browsers [8]. Our system is able to contain actions they cannot. For example, we can prevent unauthorized access to the user's private data (*e.g.* cookies) and prevent a user's machine from becoming a drone to be used in DDoS attacks. This is possible because we deny all unauthorized connections to remote hosts.

Our work is most closely related to Janus [7] and Ostia [6]. These systems do not specifically target web browsers; instead they are designed as generic operating system level sandboxes whose goal is to contain the impact of compromised applications. While they also user system call interposition, our approach differs in a number of significant ways. First, we use system call interposition not only to enforce policy but also to trace the application and user actions and dynamically change the policy. In contrast, policies in Janus and Ostia are static, read from the policy file when the sandboxed applications start. Furthermore, we employ deep application tracing in order to trace the application at a level that is finer than system calls. Finally, we use `utrace` [1], the new tracing infrastructure to be included in the Linux kernel, which fixes several problems related to system call interposition of the old `ptrace` interface.

3 Overview

The goal of our system is twofold: First, it needs to ensure that even if the browser is compromised, the exploit will not be able to damage the host (including gaining access to the user's data) or use the host to launch attacks against other networked hosts. Second, it must perform these tasks without significantly affecting the user's browsing experience. This second requirement is equally important because approaches that sacrifice ease of use for the sake of security are unlikely to be widely adopted.

To understand what is necessary to achieve the first goal, we describe a typical malware infection and its possible aftermath. Web-based attacks exploit a vulnerability in the browser, its libraries, or its plugins in order to take control and start executing some malicious code in the context of the browser. The code usually injected to the browser at this stage (sometimes known as *shellcode*) is very simple; its main function is to install the rest of the malware code. This process involves downloading the actual malware code from an external URL, saving it to the local filesystem, and finally executing it. This second stage will then either scan the local filesystem for sensitive data to be sent to the attacker, start DDoS attacks, start network servers waiting for inbound connections, or attempt to infect other hosts [20]. One can then see that for the malware to achieve its purpose, it must be able to read and write files, open connections to arbitrary hosts, and execute programs. All of these actions are enabled by the operating system and invoked through specific system calls such as `open()` and `connect()`. Therefore, by denying these sensitive system calls one can contain the malware's actions even if the browser has been compromised. This observation is the basis of all system call interposition mechanisms. However, while blocking system calls is relatively easy, given the appropriate OS-level support, the effectiveness of these approaches is determined by the set of policies implemented to enforce containment.

It is easy to see that one cannot indiscriminately prevent the browser from accessing system calls without severely affecting its usability – the browser should still be able to perform the user's requests. The challenge therefore is to differentiate between system calls that correspond to user requests and those that represent the actions of the malicious code that compromised the browser. In turn this challenge translates to the need to identify the user's explicit and implicit requests. In the paragraph that follow we outline the tracing techniques we developed to achieve this goal.

GUI tracing. For an example of GUI tracing, let us look at a scenario in which the user types an address at the URL bar. This action signals the user's intent to visit the specific website and therefore the corresponding network connection should be allowed. To do so, we record this address by tracing the GUI requests that the browser makes. Then, when the web browser subsequently attempts to open a connection to the corresponding webserver, we check that the connection is made to an IP address that corresponds to the DNS name described in the URL bar. We extend this GUI tracing technique to allow users to browse the local filesystem for loading and saving files to/from the filesystem.

Input tracing. GUI tracing enables the browser to download the base HTML page associated with a URL that the user types. This page however might include embedded references to objects (*e.g.* images) located at different webservers. For the page to load properly access to these servers must be enabled. To do so, we developed a tracer that inspects the HTML page that the browser loads and records all external references. HTTP connections to these hosts are then explicitly allowed. Figure 1 presents an example in which GUI and input tracing are used to allow proper rendering of a page that includes embedded links.

Fine grain application tracing. The combination of GUI and input tracing allows the browser to properly load complete HTML pages. This combination however does not support pages that use Javascript, SSL, or plugins. Javascript can be used, directly or indirectly, to access the network. For example, Javascript is used to generate HTML code that retrieves an image from an external site. The issue is that the URL of the image is not statically encoded into the HTML page. The input tracer therefore cannot record the correct URL and thus access to the image will be denied. The use of SSL-based pages has a similar effect: the input tracer parses the cyphertext and thus finds no embedded links [1].

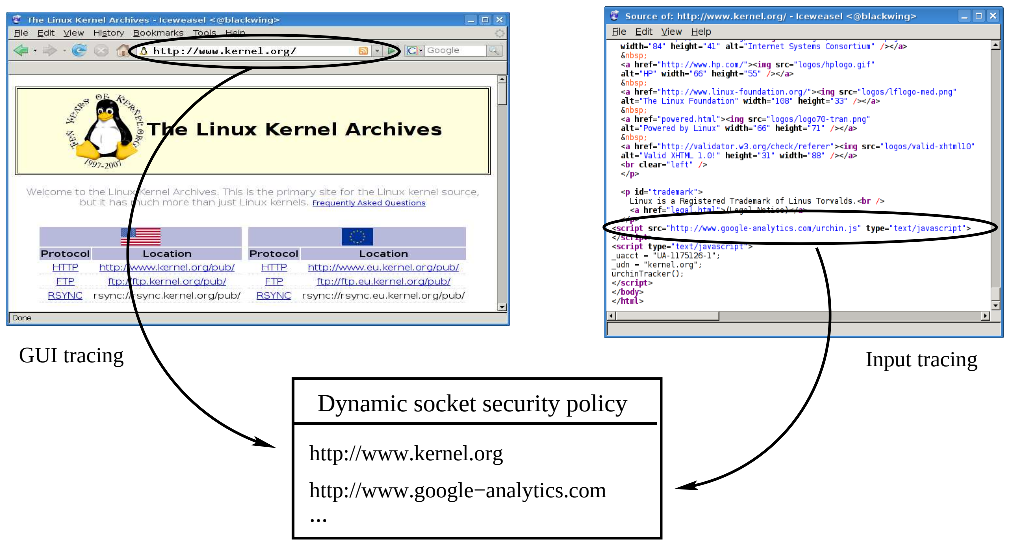
A potential solution to these problems would be to augment the input tracer with a Javascript engine and to use an SSL trusted proxy [4]. Doing so would however increase the tracer's complexity thus increasing its susceptibility to attacks. Moreover, this approach would duplicate the effort of parsing Javascript and encrypting/decrypting HTML pages thereby increasing the system's overhead. Instead, we developed a mechanism for peering deeper into the browser's control flow to extract the necessary information. In the case of Javascript, we extract the HTML code that the Javascript engine generates before it passes to the browser's HTML parse. Likewise, for SSL we extract the decrypted HTML before it passes to the HTML parser. Figure 2 illustrates this process in the context of Javascript parsing.

Stack-based tracing. One of the shortcomings of the mechanisms described so far is that if the malicious code connects back to the site that hosts the exploit (*i.e.* the site originally visited at the user's request), it will be allowed to do so. Moreover, in order to allow local caching to function properly, we must allow writes to a specific part of the filesystem (*e.g.* the `.mozilla` directory inside the user's home directory in the case of Linux/Firefox). This however would mean that the malicious code would be able to write files to that part of the filesystem.

Fortunately, the same fine grain application tracing techniques addresses these problems as well. Before allowing access to a resource, the tracer inspects the browser's stack trace to determine if the resource request follows a "proper" execution trail. When the browser issues a request, it will leave on the stack a trail of the actions that it performed before issuing that request. By comparing the trail that leads to the current request to a list of previously-derived valid trails, the tracer determines whether the request is benign or rogue.

Because we allow the browser access to system resources only under a very limited set of conditions that are directly related to user actions, we argue that we can contain all the previously described malware activities. Moreover, as we show in Section 6.3, these techniques generate no false positives and thus do not affect the user's browsing experience.

[1]We note that SSL encryption/decryption happens in userspace, thus after the `read()` system call.


The Linux Kernel Archives - Iceweasel <@blackwing>
File Edit View History Bookmarks Tools Help
http://www.kernel.org/
Google
The Linux Kernel Archives
Done
GUI tracing
Source of: http://www.kernel.org/ - Iceweasel <@blackwing>
File Edit View Help
Input tracing
Dynamic socket security policy

Welcome to the Linux Kernel Archives. This is the primary site for the Linux kernel source, but it has much more than just Linux kernels. Frequently Asked Questions

Protocol	Location	Protocol	Location
HTTP	http://www.kernel.org/pub/	HTTP	http://www.eu.kernel.org/pub/
FTP	ftp://ftp.kernel.org/pub/	FTP	ftp://ftp.eu.kernel.org/pub/
RSYNC	rsync://rsync.kernel.org/pub/	RSYNC	rsync://rsync.eu.kernel.org/pub/

```
  width="84" height="41" alt="Internet Systems Consortium" /></a>
  &nbsp;
  <a href="http://www.hp.com/"><img src="logos/hplogo.gif"
  alt="HP" width="66" height="55" /></a>
  &nbsp;
  <a href="http://www.linux-foundation.org/"><img src="logos/lflogo-med.png"
  alt="The Linux Foundation" width="108" height="33" /></a>
  &nbsp;
  <a href="powered.html"><img src="logos/logo70-tran.png"
  alt="Powered by Linux" width="66" height="71" /></a>
  &nbsp;
  <a href="http://validator.w3.org/check/referer"><img src="logos/valid-xhtml10"
  alt="Valid XHTML 1.0!" height="31" width="88" /></a>
  <br clear="left" />
  </p>

  <p id="trademark">
    Linux is a Registered Trademark of Linus Torvalds.<br />
    <a href="legal.html">(Legal Notice)</a>
  </p>
<script src="http://www.google-analytics.com/urchin.js" type="text/javascript">
</script>
<script type="text/javascript">
_uacct = "UA-1175126-1";
_udn = "kernel.org";
urchinTracker();
</script>
</body>
</html>
```

http://www.kernel.org

http://www.google−analytics.com

...

Figure 1: Example of GUI and input tracing. The URL at the address bar is used to dynamically allow connections to the requested webserver, while the base HTML code is parsed to allow connections necessary to fetch objects referenced in the base page.

4 Architecture

We present a proof-of-concept implementation of the architecture outlined above using Linux and Firefox. We selected this combination because it was easier to add support for tracing and breakpoints. Nonetheless, the architecture can be implemented for other combinations of operating systems and browsers – Windows offers similar tracing infrastructures and we do change the browser.

We begin with a description of the policy rules that the architecture needs to support to achieve the dual goal of containment and usability described above.

4.1 Policy Control

We divide policy rules into two categories based on whether they control access to the filesystem or the network.

Filesystem security policy. The filesystem security policy controls which files can be opened for reading or writing. The default policy action is to deny access to the filesystem, unless otherwise allowed by another rule. Part of the policy is static, defined before the browser starts, and part of it is generated dynamically as the browser runs. The static policy is necessary to allow the browser to function properly: mapping libraries into its address space, reading configuration files, and writing to and reading from the file cache.

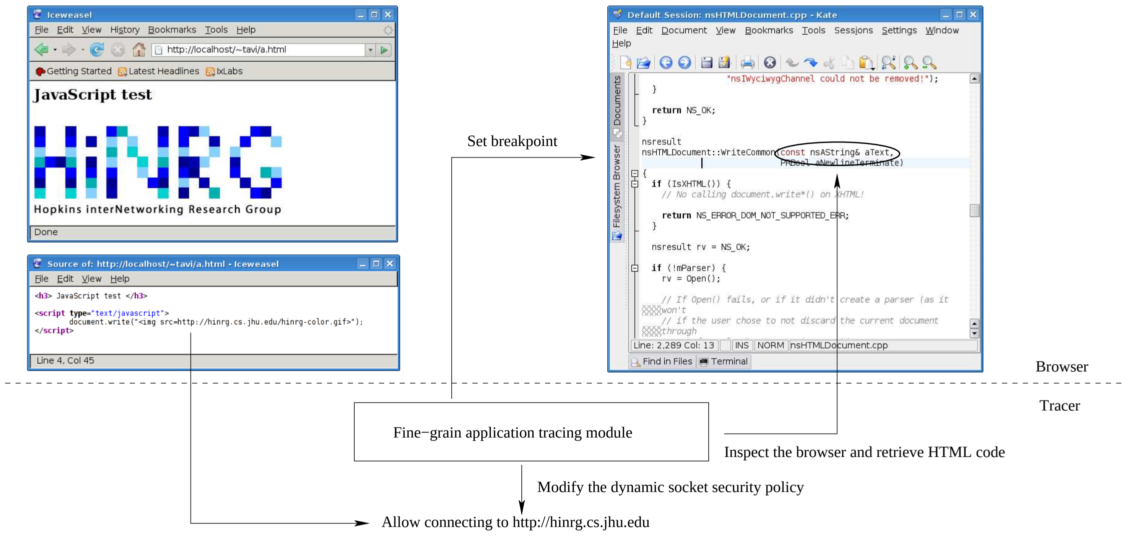
Each policy rule contains:

1. A string that identifies a filename, part of a filename, path or part of a path.
2. Type of match operation: full path match, full filename match, sub path match.
3. Optional stack traces; when this is present, the operation will be permitted only if the stack trace at the time of check equals the one(s) in the rule.
4. A mode which determines the types of operations permitted on the file: read, write or both.
5. A flag that identifies whether the rule is a one-shot rule; one-shot rules are inserted while the browser runs and are removed after the first time they match.
6. A permit or deny action.

Network security policy. The network security policy controls the browser's access to the network by controlling access to all socket-related system calls. As with the filesystem policy the default action is to deny access unless explicitly allowed by one of the policy rules. As before, rules can be either static or dynamic. We use the static rules mainly to allow access to the DNS service. All other network accesses are controlled via dynamic rules.

4.2 Application Tracing and Security Engine

Figure 3 diagrammatically presents the components of our approach. The Application Tracing and Security and Engine (ATSE), seen in that figure, is the system's enabler. It is implemented as a kernel module, leveraging two new infrastructure projects in the Linux kernel: `utrace` and `uprobe` [24]. We use `utrace` as the infrastructure for system call interposition in order to addresses the


Iceweasel
File Edit View History Bookmarks Tools Help
http://localhost/~tavi/a.html
Getting Started Latest Headlines IxLabs
Done
Source of: http://localhost/~tavi/a.html - Iceweasel
File Edit View Help
Line 4, Col 45
Default Session: nsHTMLDocument.cpp - Kate
File Edit Document View Bookmarks Tools Sessions Settings Window Help
Documents
Filesystem Browser
Line: 2,289 Col: 13
INS
NORM
nsHTMLDocument.cpp
Find in Files
Terminal
Set breakpoint
Browser
Tracer
Fine−grain application tracing module
Inspect the browser and retrieve HTML code
Modify the dynamic socket security policy
Allow connecting to http://hinrg.cs.jhu.edu

JavaScript test

Hopkins interNetworking Research Group

```
<h3> JavaScript test </h3>

<script type="text/javascript">
        document.write("<img src=http://hinrg.cs.jhu.edu/hinrg-color.gif>");
</script>
```

```
                    "nsIWyciwygChannel could not be removed!");
  }

  return NS_OK;
}

nsresult
nsHTMLDocument::WriteCommon(const nsAString& aText,
                            PRBool aNewlineTerminate)
{
  if (IsXHTML()) {
    // No calling document.write*() on XHTML!

    return NS_ERROR_DOM_NOT_SUPPORTED_ERR;
  }

  nsresult rv = NS_OK;

  if (!mParser) {
    rv = Open();

    // If Open() fails, or if it didn't create a parser (as it
won't
    // if the user chose to not discard the current document
through
```

Figure 2: Example of fine grain application tracing. The panel to the left shows the rendered page and a snippet of Javascript code that fetches an image. The right panel shows the breakpoint reached after the Javascript parser decodes the Javascript code. At that point the input parser inspects the generated HTML code and allows access to the image.

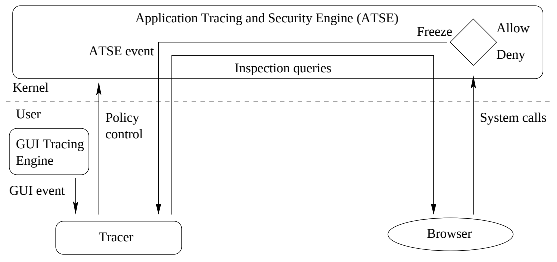


Figure 3: Architecture components and their interactions. The browser's system calls are intercepted by the ATSE and allowed or denied based on the policy rules inserted by the Tracer. The Tracer can freeze the browser's execution to inspect its state. The GUI Tracing Engine (GTE) passes intercepted GUI events to the Tracer.

shortcomings of the older `ptrace` interface, described at length in [26]. We use `uprobe` to allow the Tracer to insert breakpoints in the browser.

Briefly, the ATSE allows us to monitor the browser by tracing its system calls. Moreover, it is possible to select which system calls will be traced, when the execution will be interrupted (*i.e.*, at the call's entry or exit point), set conditions for tracing only certain invocations of the selected system calls (*e.g.*, `close()` system calls which use a certain file descriptor), and finally allow or deny a system call directly from ATSE –without involving the Tracer. All these features are paramount to good application performance.

Once control passes to the ATSE, it has complete control of the browser: it can deny system calls (and return proper error codes) and inspect the browser's state. The later capability includes the ability to peek into the browser's virtual address space and registers and determining the files corresponding to its file descriptors. Furthermore, it is possible to automatically trace new threads and processes spawned from a browser without any race conditions. Finally, in case the user-level Tracer terminates, ATSE also terminates the browser to ensure safety.

The policies enforced by the ATSE are created by the user-level Tracer. In addition to defining the system calls and the conditions under which they will be traced, the policies specify whether rules are "one-shot" or recurring and the action that should be taken after a match occurs. Supported actions are allowing or denying the system call or freezing the browser and passing control to the Tracer.

The engine also allows the Tracer to insert breakpoints into the browser. The ATSE is notified by a `uprobe` handler when execution reaches one of these breakpoints. It subsequently freezes the browser and informs the Tracer of the condition. The Tracer then inspects the browser's stack frame, registers and address space to extract the information it needs. The ability to set arbitrary breakpoints allows the Tracer to perform the application tracing described above.

The interface between the engine and the Tracer is file based, *i.e.*, the Tracer will open the engine's device file and use it for all the communication with the engine via `ioctl()` calls. The Tracer can also use the `select/poll/epoll` interface for event notification.

4.3 Tracer

The Tracer is the user-space process responsible for creating and modifying the policy that the ATSE enforces. It also monitors the browser's actions and some of its internal state in order to appropriately alter the policy rules. It starts by forking a new process in which the browser will be run, registering this process with the ATSE, installing the default deny policy which sandboxes the browser, and finally running the browser in the newly forked process. Once the Tracer spawns the browser, it waits for events and reacts to them by modifying the policy enforced by ATSE.

In some cases the ATSE does not have enough context to make a decision about a system call request. In these situations the ATSE freezes the browser and passes control to the Tracer which decides whether the system call should be allowed or denied based on its own set of policy rules. These rules are evaluated in the order in which they are generated by the Tracer. There is no priority associated with the rules, because the level of specificity of a rule is always very high and static rules are defined in the order of the specificity (specific rules first followed by general rules).

4.4 GUI Tracing Engine

The GUI Tracing Engine (GTE) is an X-windows proxy that runs in user space. Its role is to forward GUI requests from the browser to the actual X server while allowing the Tracer to peek into these requests first. In this way, the Tracer "sees" what the user sees and it is able to modify the security policy in response to the user's actions.

The GTE is based on the `xtrace` [16] Linux utility, which we had to rewrite considerably, in order to address a number of performance and integration issues but also to simplify it according to our modest needs—`xtrace` contains a complex request decoding machine to pretty print most of X protocol's requests and replies. While this functionality is important when debugging X-windows it only increases overhead in our case.

In order to avoid potential races between components of the system running in different processes or threads we implemented the GTE and the Tracer as part of the same single-threaded process. Because of this decision and the fact that we have multiple event sources, we use asynchronous I/O for event notifications.

5 Implementation

5.1 GUI Tracing

While the GTE receives GUI requests from the browser, it evaluates them against a set of policy rules set by the Tracer. If a request matches one of the rules, the engine invokes a callback function installed by the Tracer. Note that the callbacks into the Tracer are made synchronously, *i.e.*, the request is forwarded to the actual X-server only after the callback returns. This behavior ensures that the Tracer will gather the necessary information and potentially alter the security policy, before the browser finishes the GUI rendering and proceeds to other tasks (such as issuing a request to a remote site). While some of the rendering operations the browser performs are asynchronous with respect to the I/O operations it issues, the types of operations we trace always precede any I/O operations.

We use GUI tracing to determine the text shown in the status and URL bars. This text is subsequently used to allow access to a file in the local filesystem or to a remote host. Specifically, URLs starting with the `file:/` prefix typed in the URL bar are used to modify the filesystem policy, allowing read access to the corresponding local file. Likewise, the Tracer resolves the hostnames of non-local URLs and inserts network policy rules allowing subsequent `connect()` system calls to the IP addresses(es) associated with those DNS names. Note that in addition to the destination address, the network rule also specifies the destination port allowed. Finally, the Tracer uses the text retrieved from the text bar to identify the URL that the user clicks on[2]. While this method has security issues because Javascript code can print arbitrary messages to the status bar, these issues can be resolved by using a browser plugin to replace the regular URL bar.

The GUI policy rules used by the GTE and Tracer are composed of atoms, which in turn are blobs of data to be matched at a certain offset within the request. This matching technique is very flexible, allowing us to match any X Windows Protocol [22] type of request, sub-request or even requests that are to be applied to a particular window. At the same time, this type of matching affords a straightforward and lightweight implementation. The rule also specifies whether it should match client and/or server requests. The action of a rule is to execute the callback set by the Tracer.

In order to track the text, we intercept `CompositeGlyphs` X11 sub-requests of the `Render` request. These requests are used by modern X11 server/-clients to draw fonts on the display. For the current proof of concept implementation we modified the browser configuration to use particular fonts and font sizes for both the URL and status bar. We chose a fixed font in order to avoid the kerning performed by modern X11 client/servers. Otherwise a more complicated tracking procedure would have been necessary in order to determine the kerning pairs. We initially planned to track the URL/status bar text by determining the position of this text relative to the browser's window. Unfortunately, the X-windows requests specify the coordinates of the fonts to be rendered relative to the drawing element. This means that the GTE would need to track multiple kinds of requests in order to get the positions relative to the window coordinates. By changing the font sizes we fix a specific set of relative coordinates which we subsequently

[2]The browser changes the text in the URL bar to the URL that the user has clicked on after initiating the connection.

use to identify the requests made to render the text into the URL or status bars.

GUI tracing is also used to track the “Save as” dialog. For this purpose, the GTE tracks `ChangeProperty` requests to identify when the browser changes the name of the window to “Save as”. It also tracks `DestroyWindow` requests to determine when the window is closed. To detect when the user requests a file to be saved to the local filesystem, the Tracer inserts a breakpoint in the code that handles the “Save as” dialog file selection, at line 292 in the code excerpt shown in Figure 4. When execution reaches this breakpoint, the Tracer queries the GTE to check that indeed a “Save as” dialog was presented to the user. It then retrieves the function’s return value which is the address to the string that identifies the file. Finally, it accesses the browser’s address space and retrieves the actual filename which it uses to insert a one-shot rule that allows write-only access to that file.

While the current proof-of-concept implementation of GUI tracing requires some modifications to the browser’s appearance, these shortcomings can be addressed either by implementing a more complex GUI tracing module to track the required intermediate states, or by adding an extension to the browser that replaces the existing URL bar with one that simplifies the tracking of GUI requests.

5.2 Input tracing

Input tracing is necessary to allow the browser to fetch embedded objects stored on hosts other than the one on which the current HTML page is stored. Input tracing starts when a `connect()` system call, previously identified as a valid browser connection, returns successfully. When that happens the Tracer adds the socket to the list of sockets it tracks, intercepting all subsequent receive system calls on that socket. In Linux (as well as other variants of UNIX) all socket system calls actually share a single system call, with the following prototype: `int socketcall(int call, unsigned long *args);` To improve the performance the network policy rule that describes intercepting the `recv()` system call states that the ATSE should intercept the `__NR_socketcall` system call number but also that the first argument should be `SYS_recv`. At the same time the Tracer adds a new rule to intercept any `close()` system calls for that socket. The rule is marked as a one-shot rule, to be automatically deleted by the ATSE, since the rule is irrelevant after the socket has been closed.

During a receive operation, the Tracer retrieves the receive buffer from the browser’s address space and parses it looking for externally stored content. To do so, we developed a simple HTTP parser that recognizes 30x HTTP responses (used to redirect the browser to alternate locations) and 200 (*i.e.* successful) HTTP responses. The 200 HTTP responses are further parsed with a simple HTML parser that recognizes tags with `src` attributes (such as `img`, `javascript`, `frame`, etc.) and extracts the external references. The parser also recognizes the `http-equiv=’’refresh’’` tag and extracts external refereces from this construct as well.

Once the parser identifies a references to external content, it alters the network security policy to allow access to the sites that the content refers to. For example, if the parser encounters in the HTML input the tag `<script src=’’http://www.google-analytics.com/urchin.js’’ type=’’text/javascript’’>` it adds to the dynamic network policy a set of rules that allow the browser to connect to any of the IPs that `www.google-analytics.com` resolves to, as long as the destination port is 80 (because the URL’s prefix is `http://`).

5.3 Fine grain application tracing

While the combination of input and GUI tracing enables the browser to load complete HTML pages requested by the user, it is not adequate to appropriately modify the security policy in all cases. Specifically, it fails for pages the use Javascript, SSL, as well as pages that use plugins such as Adobe’s Flash Player.

As previously argued, we prefer to use the browser’s Javascript and SSL engines to gain access to the required information rather than duplicating the effort. For example, in order to parse HTML pages after they are decrypted, the Tracer inserts a breakpoint to the browser at the runtime location that corresponds to line 1334 in the code excerpt shown in Figure 5. This location was experimentally found by a combination of source code examination and debugger inspection during browsing. Once reaching this breakpoint, the Tracer retrieves the function’s result (which provides how many bytes were decrypted), retrieves the address of `buf`, as well as the value of the `fd` pointer. This value is used to identify the particular stream of decrypted text, because the browser can be decrypting multiple SSL streams at the same time. Once the Tracer extracts the decrypted text from the browser’s buffer it hands it to the input parser which modifies the network security policy accordingly.

We use the same approach to support Javascript constructs such as `document.write()` which is used to insert HTML code in a page, using Javascript to set the `src` attribute of DOM objects for images, frames, iframes, and scripts (*e.g.* `document.createElement(’iframe’).src=’’http://www.kde.org’’`), or using the `setAttribute` Javascript function to set the same `src` attribute (*e.g.* `document.createElement(’iframe’).setAttribute(‘‘src’’,‘‘http://www.kernel.org’’);`).

Fine grain application tracing is also necessary to support browser plugins. The reason is that plugins such as Flash use the browser to connect and retrieve data from

```
void
nsFilePicker::ReadValuesFromFileChooser(GtkWidget *file_chooser)
{
  mFiles.Clear();

  if (mMode == nsIFilePicker::modeOpenMultiple) {
    mFile.Truncate();

    GSList *list = _gtk_file_chooser_get_filenames(
                           GTK_FILE_CHOOSER(file_chooser));
    g_slist_foreach(list, ReadMultipleFiles,
                    NS_STATIC_CAST(gpointer,&mFiles));
    g_slist_free(list);
  } else {
    gchar *filename = _gtk_file_chooser_get_filename(
                              GTK_FILE_CHOOSER(file_chooser));
    mFile.Assign(filename);
    g_free(filename);
  }
```

Figure 4: Browser code that executes during a "Save as" operation and that the Tracer interposes to extract the filename and alter the filesystem policy.

```
static PRInt32 PR_CALLBACK PSMRecv(PRFileDesc *fd, void *buf,
                                   PRInt32 amount, PRIntn flags,
                                   PRIntervalTime timeout)
{
  nsNSSShutDownPreventionLock locker;
  if (!fd || !fd->lower) {
    PR_SetError(PR_BAD_DESCRIPTOR_ERROR, 0);
    return -1;
  }

  nsNSSSocketInfo *socketInfo = (nsNSSSocketInfo*)fd->secret;
  NS_ASSERTION(socketInfo,''nsNSSSocketInfo was null for an fd'');

  if (flags == PR_MSG_PEEK) {
    return nsSSLThread::requestRecvMsgPeek(socketInfo, buf, amount,
                                           flags, timeout);
  }

  return fd->lower->methods->recv(fd->lower, buf, amount, flags,
                                  timeout);
}
```

Figure 5: Browser code that we interpose to extract decrypted HTTP data from SSL sessions.

a particular site which they then display (*e.g.* YouTube movies). On one hand, since the plugins use the browser to communicate with the external world we can use the same architecture to interpose these requests. However, to do so the Tracer must intercept the requests from the plugins and change the network security policy accordingly. We do this by adding breakpoints to the `GetUrlNotify`, `GetUrl`, `PostUrlNotify`, `PostUrl` API functions the browser offers.

5.4 Stack trace based authentication

It is helpful to review what we have achieved so far. We start with a default policy that denies the browser access to all sensitive system calls. Doing so provides a fully contained, but non-functional browser. We then insert static policy rules to allow the browser to read and write to certain parts of the local filesystem (*e.g.* cache directory) and access certain network services (*e.g.* DNS). Furthermore, through the use of GUI, input, and fine grain application tracing we recognize and allow user requests to access remote and local resources. We thus arrive at a fully functional browser from the user's perspective. However, allowing access to (a limited set of) resources, raises the concern that some exploit that overtakes the browser might leverage this privilege for malicious purposes. For example, an exploit could read files from the user's computer and transmit them back to the malicious site it was downloaded from.

It is thus necessary to develop additional mechanisms for granting access to resources, based not only on the identity of the resource requested but also on the identify of the requester. In this context requester identity is defined as an execution code path. Intuitively, the browser employs a finite number of execution paths to access resources from the operating system. Then, when an ex-

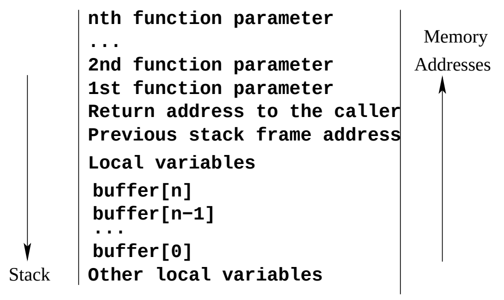


Figure 6: A sample stack memory layout. The stack allocates memory for all of the function's local variables, its arguments, and the memory address from which execution should continue after the function returns.

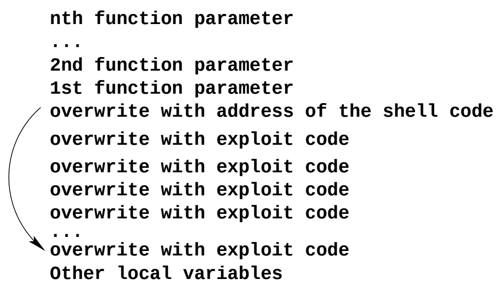

nth function parameter
...
2nd function parameter
1st function parameter
overwrite with address of the shell code
overwrite with exploit code
overwrite with exploit code
overwrite with exploit code
overwrite with exploit code
...
overwrite with exploit code
Other local variables

Figure 7: The stack from Figure 6 after a buffer overflow attack. Note that the return address has been overwritten to point to the buffer provided by the attacker.

ploit interjects itself into the browser code and requests a resource, it will invariably generate different execution paths. This observation can then be leveraged to deny access to the malware's requests.

We implement the idea outlined above through stack trace based authentication, in which the Tracer inspects the browser's stack trace (*i.e.*, the sequence of function calls) at the time of a system call and compares it with a list of reference stack traces. If no match is found, then the Tracer denies the request. Note that stack trace based authentication offers an additional layer of protection. Browser requests allowed based on the identity of the requested resource are further evaluated based on the browser's stack trace at the time of the system call.

We collect the necessary reference stacks during a training phase in which we enable stack trace recording in the Tracer. We then perform a large number of browsing sessions to empirically exercise a large number of execution paths in the browser and thus collect the necessary stack traces. Overall we collected approximately sixty different stack traces during the training phase. While the generation of reference stack traces is currently a manual process, generating them automatically is part of our future work.

We present the benefits of stack trace based authentication by describing how it blocks illegal resource requests, when such attempts are staged through two common buffer overflow attacks: a stack buffer overflow and a "return to `libc`" attack [19, 23]. For reference, Figure 6 illustrates the stack before either of the two attacks.

The buffer overflow attack (see Figure 7) works by overflowing a local buffer on the stack with input provided by the network. For example, if the size of the network-derived `input` is larger than the size of the `buffer` allocated in the stack, then `strcpy(buffer, input)` will overrun the stack). The purpose of the overflow is to replace the return address so that it points somewhere in the local buffer. Since the buffer is filled with data provided by the attacker, he can craft the data to execute arbitrary code. In particular, let us suppose that the attacker's

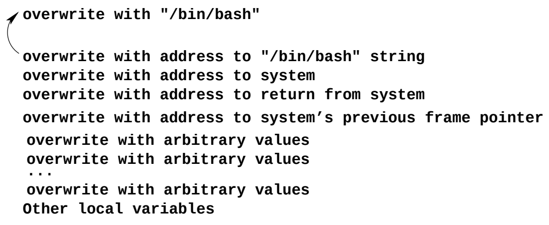

overwrite with "/bin/bash"
overwrite with address to "/bin/bash" string
overwrite with address to system
overwrite with address to return from system
overwrite with address to system's previous frame pointer
overwrite with arbitrary values
overwrite with arbitrary values
...
overwrite with arbitrary values
Other local variables

Figure 8: An example of a return to `libc` attack which spawns a shell.

code attempts to gain access to a protected resource by invoking a system call. The ATSE will then freeze the browser and pass control to the Tracer which will inspect the browser's stack. The stack trace will promptly indicate the fact that the request originates from an unauthorized place (*i.e.* the stack) and will thus be denied.

The return to `libc` attack is a more subtle. It does not use the stack to execute the exploit, instead it overwrites the stack above the return address of the current frame and then changes the return address to point to an existing function in the standard library, such as `system`. When the function in the current frame returns, program control is redirected to the `system` function and the overwritten portions of the stack are treated as arguments (see Figure 8). The attacker is then able to execute arbitrary code by crafting the arguments to the library routine.

Suppose that such a return to `libc` attack exploits function `f()` to spawn a shell and that the original stack trace just before the attack is:

```
f + x
g + y
h + z
```

where `x,y,z` are byte offsets and `g` and `h` are other functions on the stack. Then, if the malicious code successfully exploits `f()` it will transform the stack to:

```
system + t
g + y
h + z
```

However, the resulting stack trace cannot be in the list of reference stack traces because in normal execution `h+z` would precede `g+y`, `f+x` and nothing else. Thus the Tracer can conclude that the trace corresponds to an unauthorized access attempt.

We have so far discussed exploits that overflow the stack. Nonetheless, stack trace based authentication is also effective against heap or BSS overflows [14]. These attacks work by modifying a function pointer to point to some shell code stored in an input buffer allocated from either the stack or the heap. However, the effect from the stack trace's point of view is the same: the browser's execution path is changed and thus it will be detected.

Finally, while we present how stack trace authentication can prevent access to resources directly requested by the exploit code, the same model extends to cases in which the exploit attempts to manipulate the browser to confuse the Tracer. For example, there are multiple places in which the Tracer uses data provided by the browser to aid application tracing. Doing so is dangerous from a security perspective because the exploit code can call these function to provide the Tracer with fabricated data. To prevent such an attack, the Tracer checks that the stack trace matches a valid, pre-recorded trace before actually using data provided by the browser. Doing so assures that malicious code can not impersonate the browser to bypass the security checks.

5.5 Implementation Details

Thus far we presented fine grain application tracing and stack trace authentication using C-like language abstractions. This section describes the details behind those abstractions.

As described in Section 4.2, the ATSE offers the ability to read and write to the browser's virtual address space and registers. Moreover, it can retrieve the name of a process given its PID and the name of a file given the combination of file ID and PID. Finally, the ATSE can insert breakpoints at specified address locations in the virtual address space of the browser. By using the features the ATSE provides and using the calling convention of the Linux i386 ABI (Application Binary Interface) [10], the Tracer is able to access high level constructs such as function parameters, return values, etc.

Figure 9 represents the stack layout of a standard Linux application. It is evident that there is a strict relation between the current stack frame, parameters of the current function, and parameters of functions called by the current function. The Tracer leverages knowledge of this structure to extract the information necessary for tracking the browser. For example, in order to retrieve the n-th parameter of a function the Tracer retrieves the value stored at address `EBP+4+n*4`, where `EBP` stores the address of the current frame pointer. Similar calculations can be used to retrieve the n-th argument of the called function,

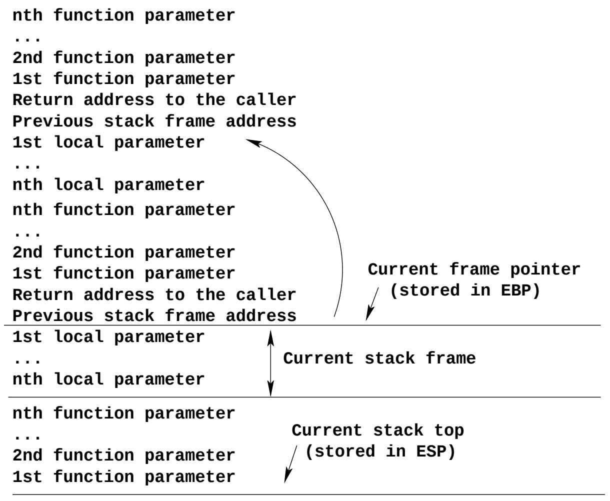


Figure 9: Stack layout for a standard Linux application.

construct a stack trace, and retrieve the return value of the called function.

In order to consistently use stack traces for authentication across multiple runs of the application we have to transform absolute stack traces to relative ones. An absolute stack trace has the addresses in its list as absolute addresses, while the addresses in a relative stack trace are offsets from the start address of the module they belong to. The reason for using relative stack traces is that libraries are loaded at different locations during each run. This is because not only the operating system does not guarantee that a library will be loaded at the same address, but it may proactively load it at different addresses in order to reduce the chance of a successful buffer overflow (this address space randomization process has been present in the Linux kernel since version 2.6.12 and enabled by default starting with version 2.6.20).

It would be possible to translate an absolute stack trace to a relative one by querying the virtual address space of the application during each authentication operation. Doing so however would create a high performance overhead. Instead, we developed a technique for learning the locations in which libraries are loaded in the browser's virtual memory. We do this by tracing two event types: `execve` and `mmap` system calls. Specifically, the ATSE is instructed to freeze and notify the Tracer whenever an `execve` system call completes successfully. The Tracer determines at that point the load address of the program and the length of the region by inspecting the `/proc/pid/maps` file. Likewise, each time an `mmap` system call with a positive `fd` argument (*i.e.*, the application is mapping a file in memory) completes successfully, the Tracer gains controls and determines the zone load address, length, and the file name associated with the zone (determined by requesting the engine to resolve the file descriptor to a file name). All this information is kept in a map which describes the relevant parts of the browser's virtual address space.

6 Evaluation

6.1 Performance evaluation

The goal of the performance experiments is to measure the overhead associated with the containment environment for the browser. The test setup we use consists of a server running Linux and the Apache web server and a client environment running in a virtual machine. The virtual machine communicates with the web server over a 100 Mbps switched Ethernet LAN. We choose to run the client environment in a virtual machine for two reasons. First, a VM provides a contained environment for running our software (user space applications and kernel modules) that is easy to modify. Second, the execution slowdown that the VM induces, simplifies the process of quantifying the performance difference between the unprotected and the contained environments. The client environment runs on a Debian GNU/Linux OS with a custom 2.6.22 kernel that includes the `utrace` and `uprobe` patches. We perform the tests by running Firefox in the normal or and contained client environment and forwarding all rendering operations to a remote X-server through an encrypted `ssh` connection. We use this approach so we can easily quantify the rendering (*i.e.*, GUI-related) overhead associated with each test.

We first measure the overhead for download operations. For this test we make the browser repeatedly download a 1GB file as fast as possible, with caching disabled. The maximum throughput achieved when protection is turned off is 4.5 MB/s and drops to 3.9 MB/s when all the tracing mechanisms described in Section 5 are enabled. As Table 1 indicates, this 13.4% decrease in throughput is mostly due to the overhead of executing the Tracer. As expected, during this test the number of rendering operations was negligible (quantified by the CPU time consumed by sshd). Further profiling of the containment environment indicates that most of the time was spent copying the received data from the browser process in order to perform input tracing. Better integrating the HTML parser with the input tracing engine would allow us to skip copying non-relevant data to the parser and thus reduce this overhead. Completely disabling the copying and parsing of the received data, significantly decreases the CPU time spent in the Tracer (8% vs. 18%); the throughput however remained fairly the same. Finally, when we do not block the browser on the socket receive calls raises the throughput to approximately 4.2 MBps. We conclude from this last result that the decrease in throughput when containment is active is mostly due to the delay introduced by freezing and resuming the browser during each socket receive operation.

Next, we measure the latency of an HTTP request. In order to accurately measure this latency, we make the browser load a simple page (∼50 bytes long) that reloads itself via a simple Javascript function and measure the elapsed time for 1,000 page reloads. The average page load time when containment is inactive is 127 ms, while it takes 142 ms on average to load the page when all tracing mechanisms are active. Table 2 shows the root causes of this 11.8% increase. Specifically, rendering operations consume a significant percentage of CPU time under both environments, as the increased values in the sshd column indicate (25%-30% compared to 1% in Table 1). In turn, the higher number of rendering operations means that the GTE has to intercept and pass a larger number of events to the Tracer for analysis, thus blocking the browser more frequently. This explanation is supported by the increased percentage of CPU time spent in the Tracer in Table 2.

Environment	**Browser**	**Tracer**	**sshd**
Normal	95%	-	1%
Containment	78%	18%	1%
Containment no copy	84%	8%	1%

Table 1: Percentage of CPU time used by the browser, the Tracer, and the sshd daemon when containment is enabled and disabled during the throughput tests. We also present the case in which copying and parsing of incoming data is disabled.

Environment	**Browser**	**Tracer**	**sshd**
Normal	75%	-	25%
Containment	40%	30%	30%

Table 2: Percentage of CPU time used by the browser, the Tracer, and the sshd daemon during request latency tests.

Next, we analyze the impact that containment has on Javascript intensive pages. To do so we perform 10,000 `document.write` calls from within a page, and reload the page ten times in a row. The average time required to perform a `document.write` operation is 50 μsec when protection is turned off and grows to 590 μsec when tracing is enabled. Table 3 shows that this significant increase in execution time is due to the overhead of freezing the browser after every `document.write` call and running the Tracer to inspect any HTML code generated by the browser's Javascript interpreter. While this increase is undoubtedly large in absolute terms, it is unlikely to be noticeable by users since only a very small number of such calls are present in actual web pages. Finally, the increase in rendering time in the normal environment shown in Table 3 is caused by the fact that more pages per second are rendered in the normal environment than in the secure environment (since Javascript processing takes less time).

Last, we evaluate the impact of containment for SSL-intensive operations. To do so, we measure the maximum

Environment	Browser	Tracer	sshd
Normal	89%	-	10%
Containment	20%	78%	1%

Table 3: Percentage of CPU time during processing of Javascript `write.document` requests.

Environment	Browser	Tracer	sshd
Normal	95%	-	1%
Containment	68%	30%	1%

Table 4: Percentage of CPU times for SSL-protected pages for normal and contained browser environments.

throughput obtained while downloading the same 1 GB file over an SSL-protected connection. Enabling protection results in a 37% decrease in throughput (2.5 MB/s vs. 4.0 MB/s). This decrease in throughput is due to the smaller percentage of time that the browser is actively downloading content, as Table 4 indicates. Further profiling of the contained environment revealed that the top causes of the overhead were the delay associated with the SSL breakpoints and, as in the cleartext throughput test, copying of the received data from the browser process to the Tracer to perform input tracing.

6.2 Containment Evaluation

We test the ability of the proposed approach to block malicious actions through a simulated exploit. We decided to follow this approach because we were not able to find "live" exploits for our evaluation platform (Firefox running on Linux). We acknowledge that this approach does not expose the tracing mechanisms to the panoply of attacks used by actual web-based malware. At the same time we argue that because the default containment policy is to deny access to all sensitive system calls, allowing only a well-defined and small set of access patterns, the proposed approach can contain a wide range of exploit behaviors. In this regard, the results from this section should be treated as a verification of the system's correct operation, rather than a proof of its security properties.

Specifically, we developed a plugin which runs inside Firefox and attempts to upload files from the local filesystem to a remote destination, thus imitating a standard trojan malware used for identity theft. This plugin registers a special ".xpt" stream with Firefox, whereby Firefox passes files with the .xpt extension to the plugin to be executed. In reality these streams contain instructions about what the plugin should do. Some self-explanatory instructions that the plugin supports are: `openr /etc/passwd`, `openw /tmp/xploit`, `connect 64.233.187.99`.

Our tracing mechanisms were able to block all of the plugin's network connection attempts (including those to the same site that the stream was downloaded from), all file operations for writing files (including those located in the browser's cache directory), and all read operations that were not specifically permitted by the static filesystem security policy. Because the static policy file we use explicitly allows access to certain files (*e.g.*, /etc/passwd, /lib/libc.so.6, etc.), the plugin was allowed to open these files in read-only mode. However, because the plugin cannot connect to any remote site, reading the files is not a major security concern by itself. Nonetheless, one can further refine the security policy by adding stack trace based authentication to limit read access to sensitive files.

6.3 Usability

The last set of tests we perform evaluates the usability of the proposed approach. Usability in this context is used to define how much the containment environment interferes with the user's browsing experience. In this regard, these tests are the opposite of the test presented in the previous paragraph. While the earlier test showed that containment can effectively protect the user's sensitive data, the goal of this test is to evaluate whether containment blocks pages, or parts of pages, that the user actually requests.

To do so we generate a list of URLs by crawling a number of popular websites that serve complex webpages with multiple embedded objects and which extensively use Javascript and plugins. The list of crawled websites includes `www.nytimes.com`, `www.espn.com`, `www.cnn.com`, `www.bbc.co.uk`, and `www.youtube.com`. In total, we generate a list of hundreds of URLs which we load through the browser with containment turned on. We also perform tests in which the browser attempts to read and write files to the local filesystem, emulating a user who traverses the filesystem's directory tree. For all these tests we record any operations that were blocked, be it accessing remote sites of reading/writing to the filesystem.

The only blocked operation among all the URLs the browser visited was a link to a private IP address (10.0.0.0/8) included in a business page of a news site. Such a link is most likely due to a benign authoring error and would not lead to a successful download even if it was not blocked. Moreover, Jackson *et al.* recently showed that having URLs which point to local address space can lead to so-called *DNS Rebinding attacks* which can be used to map the user's private network [11]. In this context, our containment mechanism acts appropriately by blocking such URLs.

7 Discussion

The use of system call interposition for containment introduces a number of security concerns [5, 28]. While some of them, such as incorrectly replicating OS state and overlooking indirect paths to resources, can be mitigated, race conditions are fundamentally harder to fix. Such race conditions include: "time-to-check to time-to-use", "time-of-audit to time-of-use", and "time-of-replacement to time-of-use". Exploiting these race conditions requires that either the attacker has at least two conspiring threads in order to bypass the security policy or that a system call's indirect arguments are located in an area of the memory that is shared with another malicious process. Our examination of the Firefox browser (version 2.0.0.5) on Linux concluded that the browser does not expose any shared memory regions, let alone use them to store system call arguments. Regarding the use of multiple threads, because we use stack trace authentication to validate thread creation, an exploit cannot spawn helper threads. Thus, the only way our approach would be vulnerable to multiple-thread attacks, would be for the attacker to exploit vulnerabilities in two or more distinct threads of the browser.

Another potential concern is that an attacker could try to subvert the stack trace authentication mechanism by constructing a fake stack trace that matches a valid trace. This assumes a powerful attacker because the stack trace will usually contain calls from multiple libraries and thus, due to the fact that these libraries are loaded at different addresses in memory, guessing a valid stack trace is challenging. Given these preconditions, the attacker would be able to execute a protected system call, but will not be able to regain control because of the modifications done on the stack and the application will most likely crash. In other words, while the attacker might be able to issue a `connect()` or an `open()` call by masquerading the stack, he will lose control of the application before issuing a `read()`, `write()` or `send()` call to access the filesystem or the network. Such an attack is thus ineffective.

Even if the attacker executes shell code to spawn a new process and thus does not need to regain control, the spawned process will still be under the Tracer's control and thus all sensitive system calls will be denied because its stack traces will be invalid (being executed from a process other than the browser). If the attacker has compromised two threads he can launch an attack which works by having one of threads modify the stack to fake a valid stack trace and call the operation he wants to perform, while the other thread replaces the fake stack trace with the original (*i.e.* malicious) one after the Tracer inspects it. It is easy to see from this description that this type of attack is particularly difficult to employ. Not only the attacker needs to take over multiple threads, but also needs to guess what a valid stack trace looks like, and finally guess the exact time he should replace the stack.

Our current prototype does not support web proxies for which the user explicitly configures the browser to forward its requests to the them (*i.e.* not redirection or *hijacking* proxies). The reason is that browsers which use such proxies connect to a network destination (the web proxy) that is different from the one entered in the URL bar, or the one added to the policy rules by the input and fine grain application tracing mechanisms. In order to support this usage pattern we would need to insert a static network policy rule allowing connection to the proxy's IP address (and port), while disabling dynamic network rules generated by the GUI, input, and fine grain application tracing. Even with these rules disabled, stack trace based authentication will be able to contain network connections generated by web exploits.

8 Summary and Future Work

In this paper we show that it is possible to use a lightweight, OS-level, approach to effectively contain web-based malware while at the same time having little or no impact on the users' normal browsing experience. We achieve this combination of protection and practicality through the coordinated use of fine grain, application state tracing.

We believe that as application complexity continues to increase, fine grain security will become a crucial capability in the effort to tackle the security implications of this complexity. However, in order for such an approach to be successfully applied in the dynamic world of software, it requires the development of tools for describing the fine grain actions that a program is allowed to perform. We envision an extended security model, in which applications are associated with fine grain security profiles, analogous to existing read/write/execute permissions [3]. In order for this model to be widely adopted, most of this information should be generated during the application's development cycle, with little effort from the programmer. This is a research area which we are actively exploring.

References

[1] Jonathan Corbet. Introducing utrace. Available at `http://lwn.net/Articles/224772/`.

[2] Microsoft Corporation. Protected Mode in Vista IE7. Available at `http://blogs.msdn.com/ie/archive/2006/02/09/528963.aspx`.

[3] Richard S. Cox, Jacob Gorm Hansen, Steven D. Gribble, and Henry M. Levy. A Safety-Oriented Platform for Web Applications. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, 2006.

[4] Mattias Eriksson. An Example of a Man-in-the-middle Attack Against Server Authenticated SSL-sessions. Available at `http:`

[3]Or the more fine grain permissions used by advanced security systems such as SELinux and AppArmor.

//www.cs.umu.se/education/examina/Rapporter/MattiasEriksson.pdf.

[5] T. Garfinkel. Traps and Pitfalls: Practical Problems in in System Call Interposition based Security Tools. In *Proceedings of NDSS 2003*, 2003.

[6] T. Garfinkel, B. Pfaff, and M. Rosenblum. Ostia: A Delegating Architecture for Secure System Call Interposition. In *Proceedings of Network and Distributed Systems Security Symposium (NDSS)*, 2004.

[7] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A Secure Environment for Untrusted Helper Applications. In *Proceedings of Usenix Security*, 1996.

[8] Green Border Technologies Inc. GreenBorder Pro. Available from: http://www.greenborder.org/, 2007.

[9] Novell Inc. AppArmor. Available from: http://en.opensuse.org/Apparmor, 2007.

[10] SCO Group Inc. System V Application Binary Interface - Intel386 Architecture Processor Supplement, Fourth Edition. Available at http://www.caldera.com/developers/devspecs/abi386-4.pdf.

[11] C. Jackson, A. Barth, A. Bortz, W. Shao, and D. Boneh. Protecting Browsers from DNS Rebinding Attacks. In *Proceedings of the 14^{th} ACM conference on Computer and Communications Security (CCS)*, 2007.

[12] K. Jain and R. Sekar. User-Level Infrastructure for System Call Inteposition: A Platform for Intrusion Detection and Confinement. In *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*, 2000.

[13] Ashlesha Joshi, Samuel T. King, George W. Dunlap, and Peter M. Chen. Detecting Past and Present Intrusions through Vulnerability-Specific Predicates. In *Proceedings of the twentieth ACM symposium on Operating systems principles (SOSP)*, 2005.

[14] M. Kaempf. Vudo malloc tricks. *Phrack Magazine*, August 2001.

[15] Poul-Henning Kamp and Robert N. M. Watson. Jails: Confining the omnipotent root. In *Proceedings of the 2^{nd} International System Administration and Networking Conference (SANE)*, 2000.

[16] Bernhard R. Link. X server protocol tracer. Available at http://xtrace.alioth.debian.org/.

[17] Linux vserver. Available from: http://linux-vserver.org/Paper, September 2007.

[18] Alexander Moshchuk, Tanya Bragin, Damien Deville, Steven D. Gribble, and Henry M. Levy. SpyProxy: Execution-based Detection of Malicious Web Content. In *Proceedings of Usenix Security 2007*, 2007.

[19] Aleph One. Smashing the stack for fun and profit, journal = Phrack Magazine, vol=49, num=14, year = 1996, month = nov.

[20] Niels Provos, Dean McNamee, Panayiotis Mavrommatis, Ke Wang, and Nagendra Modadugu. The Ghost in the Browser: Analysis of Web-based Malware. In *Proceedings of the first USENIX workshop on hot topics in Botnets (HotBots'07).*, April 2007.

[21] Charles Reis, John Dunagan, Helen J. Wang, Opher Dubrovsky, and Saher Esmeir. BrowserShield: Vulnerability-Driven Filtering of Dynamic HTML. In *Proceedings of Usenix OSDI 2006*, 2006.

[22] Robert W. Scheifler. X Window System Protocol. Available at http://www.msu.edu/~huntharo/xwin/docs/xwindows/PROTO.pdf.

[23] H. Shacham, M. Page, B. Plaff, E-J. Goh, N. Modadugu, and D. Boneh. On the Effectiveness of Address-Space Randomization. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, October 2004.

[24] Uprobes: utrace-based user-space probes. Available at http://sources.redhat.com/ml/systemtap/2007-q2/msg00108.html.

[25] Jeff Victor. Zones and Containers FAQ. Available from: http://www.opensolaris.org/os/community/zones/faq/, July 2007.

[26] David A. Wagner. Janus: an approach for confinement of untrusted applications. Masters Thesis. Available from: http://digitalassets.lib.berkeley.edu/techreports/ucb/text/CSD-99-1056.pdf, 1999.

[27] Yi-Min Wang, Doug Beck, Xuxian Jiang, Roussi Roussev, Chad Verbowski, Shuo Chen, and Sam King. Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites That Exploit Browser Vulnerabilities. In *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*, 2006.

[28] Robert N. M. Watson. Exploiting Concurrency Vulnerabilities in System Call Wrappers. In *Proceedings of the 2007 First USENIX Workshop on Offensive Technologies Privacy*, 2007.