



Auto-FBI: A User-friendly Approach for Secure Access to Sensitive Content on the Web

Mohsen Zohrevandi Rida A. Bazzi*

Arizona State University
699 S. Mill Ave.
Tempe, AZ, USA
{mzohreva, bazzi}@asu.edu

ABSTRACT

We propose a novel and simple approach for securing access to sensitive content on the web. The approach automates the best manual compartmentalization practices for accessing different kinds of content with different browser instances. The automation is transparent to the user and does not require any modification of how non-sensitive content is accessed. For sensitive content, a Fresh Browser Instance (FBI) is automatically created to access the content. Our prototype system **Auto-FBI** can provide support for novice users with predefined sensitive content sites as well as for more experienced users who can define conflict of interest (COI) classes which allows content from sites in the same user-defined class to coexist in a browser instance. Our initial performance evaluation of **Auto-FBI** shows that the overhead introduced by the approach is acceptable (less than 160 ms for sites that already have fast load time, but for slow sites the overhead can be as high as 750 ms).

1. INTRODUCTION

Content delivered through web browsers continues to be a major vector of attack in personal computing systems. A large variety of attacks [1, 14, 15, 19, 17] exploit what might be termed *convenience features* of browser content. Executing scripts, such as JavaScript [8], in web browsers enable a richer browsing experience and enabling cookies [11] allow for the convenience maintaining a state across multiple pages. Unfortunately, convenience features also enable richer attack modalities. The tension between providing richer content and providing security is real and it is not only due to technical reasons; the more features a system has, the less likely it is for a novice, or even for an experienced user, to get the system settings right. Default browser security settings do not do much to mitigate the situation. Unless security settings are draconian in nature, such as disabling

all active content for example (which will also diminish the browsing experience) they do not solve the problems introduced by these convenience features. The possibility that a browser process is compromised by malicious content increases with the usage of the process. The more sites are visited, the more likely it is that malicious content is running in the browser.

In this paper we are concerned with the protection of the content associated with *sensitive* websites, such as websites of banks or other financial institutions. Compromising access to such websites can have direct financial implications. We consider both passive compromise in the form of information leakage and active compromise in the form of unauthorized access through cross site request forgery [1] for example, amongst many other active attacks. Our approach is to automate what an extra careful user would do. A careful user can *manually* improve security through strict compartmentalization of access. At the extreme, a user can have a computer system dedicated to the access of one website, for online banking for example, and nothing else. A less extreme, but still quite cumbersome approach, would require the user to delete all saved data (history, cookies, passwords for example) after each access and start a new *fresh browser instance* (FBI) for every new access. We call this approach *fresh instance browsing*. For added protection, the most restrictive privacy settings are also used if they do not interfere with site functionality. A less extreme approach would have the user use one machine but multiple browsers; one browser (Firefox for example) can be used for everyday browsing and one browser (Chrome for example) for browsing sensitive websites (following FBI). While such an approach can indeed significantly increase security, it is not practical as most users would not be willing or even able to go through the hassle it involves. The common user pays for convenience features through increased vulnerability. The advantage of using multiple browsers to protect against a variety of attacks has long been folk wisdom and has been established by Chen et al. [5] who propose to implement the isolation and entry-point restrictions (obtained from using multiple browsers) inside one browser. They implement their approach by modifying the Google Chrome browser. In this paper, we also argue that using multiple browsers can significantly increase security, but we take a completely different approach in achieving a user-friendly multiple browser experience. Instead of providing the user with one browser that implements isolation internally, we automatically launch an FBI whenever the current browser instance should not

*Contact author

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org. ACSAC '13 Dec. 9-13, 2013, New Orleans, Louisiana USA Copyright 2013 ACM 978-1-4503-2015-3/13/12 ...\$15.00. <http://dx.doi.org/10.1145/2523649.2523683>

be used to handle the requested website. The launched browser need not be the same as its launcher; the launched browser can be Firefox while the launcher can be Chrome, for example.

We recognize that not all users are equally savvy in setting security preferences. For novice users, we have a very simple default usage model that allows safe browsing for *sensitive sites* and does not change the way users access *non-sensitive sites*. For more experienced users, we have a more nuanced usage model that allows users to define simple yet powerful security preferences to allow them to access multiple sites in the same conflict of interest (COI) instance. This is addressed in more details in Section 3.

Auto-FBI.

We have developed a prototype system called **Auto-FBI** for 64-bit Linux. It works with Google Chrome and Mozilla Firefox. The prototype is mostly written in C except a small part that is written in JavaScript (for Chrome and Firefox extensions).

Our prototype system **Auto-FBI** implements our approach using a number of security techniques. It uses system-call interception to restrict connections to web servers. This requires tracing all the browser's system calls. **Auto-FBI** uses a simple browser extension to obtain the URL that the browser is attempting to access (we give a detailed justification of this approach in Sections 4 and 5). In order not to interfere with browser functionality, sites for which access is restricted for a given instance are serviced through a combination of DNS-query rewriting and replacement of restricted pages with default pages. In order to keep track of all active instances and their COI classes, a daemon outside the browser keeps track of active instances and launches new instances as needed. To communicate the URL of the website being accessed to the daemon, an intermediate process (browser-dependent, written in C) forwards messages between the browser extension and the daemon. Finally, to avoid buffer overflow attacks, URLs are read using constant-bounds while loops. While these techniques are individually not new, we believe that we are using them together in a novel way to achieve our objectives. Details of **Auto-FBI** are given in Sections 4 and 6.

We measured the amount of overhead incurred by our system using a benchmarking software from the Chromium project to compare the amount of time it takes to load a webpage with and without our system. As illustrated in the performance evaluation section later on, the incurred delay is not perceptible to the user for most sites, so we argue that this security scheme that we are proposing is practical.

Contributions.

Unlike most other approaches to browser security whose aim is to protect against information leakage and are based on information flow control [2, 6, 7], our approach provides protection against information leakage (privacy protection) as well as protection against some active attacks (integrity threats). Cross site request forgery (CSRF) [1] and clickjacking (UI redress) [18] attacks are prevented with our approach as far as access to the sensitive websites is concerned. We emphasize here that we are not claiming that our approach prevents all CSRF and clickjacking attacks. We are claiming that no unauthorized access to sensitive content can be done using these attacks. We do not rule out

access to sensitive content through other kinds of attacks (such as phishing [16]). This protection is due to the strong compartmentalization of access to sensitive content. On the downside, our approach does not provide fine-grain control and, for the novice user case, can be considered restrictive in its limitation of access to third-party content through sensitive sites. The security guarantees are discussed in Section 5.

The main contributions of the paper are the following:

1. We identify and introduce the automatic fresh browser instance approach (**Auto-FBI**) as an effective, easy to use, and general approach for providing secure access to sensitive content for novice users. Unlike other approaches, our approach is tolerant of privacy as well as integrity attacks while being browser-independent.
2. We generalize the basic approach to a COI-based **Auto-FBI** approach that affords experienced users more flexibility and allows them to tailor it to their specific needs.
3. We provide a prototype implementation, **Auto-FBI**, that is the first implementation that we are aware of that provides automatic support for separating accesses to sensitive sites from those to other sites while requiring almost no modification of how the user uses the browser.

The rest of the paper is organized as follows. Section 2 introduces the system and attack model. The usage model from user's perspective is discussed in Section 3. Section 4 introduces the basic **Auto-FBI**'s design and Section 5 discusses the system security guarantees. Section 6 presents **Auto-FBI**'s implementation details and Section 7 presents the performance results. Related work is presented in Section 8. Section 9 concludes the paper.

2. SYSTEM AND THREAT MODEL

We identify the relevant components of the system. The system consists of a client side and a server side. Web clients typically run in *web browsers*. Web browsers retrieve *web content* delivered by *web servers*. Web servers and web content have *origins* associated with them. We consider an origin to be a principal (whether or not it is authenticated). Web content is acted upon by the web browser. Content might be *active* which requires the browser to execute some code to process it. We consider a browser that processes content from a given origin to be a *delegate* of the origin. Web content is retrieved using *web queries*. A query can be a *simple query* that specifies the address of the content in the form of a URL (uniform resource locator) which consists of a *domain name* and file path or an *active query* that consists of the above and parameters that are passed to server-side code – *scripts* – that use the parameters to process the query. A web server might require *credentials* from a web browser in order to authenticate the principal requesting queries. Credentials are typically provided when initial access is requested, typically in the form of a *login id* and *password*. Once credentials are approved, a *cookie* is stored by the browser to be used for future authentication. At that point the browser acts as a delegate of the authenticated principal.

An *instance of a web browser* is a software process that executes the browser code and has associated with it a *profile*

which includes information such as preferences, browsing history, cookies, bookmarks, and saved passwords. A profile captures an *execution state* of the browser. When a browser exits, some information might be deleted from the profile, non persistent cookies for example, but other information in the profile is saved. When a browser is restarted after exiting, the profile provides continuity between the old browser process and the new process.

Given that a browser instance can be processing content from different sources, and therefore is a delegate for these sources, it is important that these various roles do not get mixed together in the browser. For example, a script running in a browser on behalf of one source should not be able in general to read data associated with another source in the browser – such as passwords being entered in a form. Such restrictions are expressed as security policies and are enforced by the browser. Unfortunately, given the great flexibility provided by the browser, scripts have the ability to obtain information indirectly through attempted access to forbidden resources and to communicate directly and indirectly with their origins. This flexibility makes it hard to enforce policies and even when policies are correctly enforced, they might still allow undesired behavior. Single Origin Policy (SOP) is an example of a policy that can be circumvented rather easily [12, 15]. In general, a *security policy* restricts what actions can be done by the various system processes. Abstractly, one can think of a policy as a *set of allowed system executions*. A policy for a web browser expresses what executions of the browser are allowed and which executions are not allowed as seen by an external observer. In other words, a policy on a web browser is a restriction on the input/output behavior of the browser.

A browser is susceptible to an attack relative to a security policy if some browser executions do not satisfy the security policy. We define attacks not in terms of the specific technical modalities to achieve them, but in terms of what can be achieved. In general, an attack requires the browser to get and process content from one or more *attack origins*. In describing an attack we assume that there are at least three classes of origin: HIGH, LOW, and AUTH (authenticated). We assume LOW is disjoint with both HIGH and AUTH, but HIGH and AUTH need not be disjoint. The following are attack models we consider in this paper.

1. Information Leakage (Leakage): An attacker is able to launch an information leakage attack if a LOW origin is able to receive content from a HIGH origin through the browser.
2. Cross Site Request Forgery (CSRF): An attacker is able to launch CSRF attack if requests to an AUTH origin to which the browser is already authenticated *depends on* content from a LOW origin.
3. Clickjacking (UI Redress): UI redress attack is similar to to CSRF attack, but requires user participation by interacting with user interface elements. It is typically achieved by exploiting features of the user interface to let a user interact with a target website when they think that they are interacting with other content.

We realize that the way the attacks are described does not necessarily follow the way they are typically described in the literature, but we opted for a more general description that

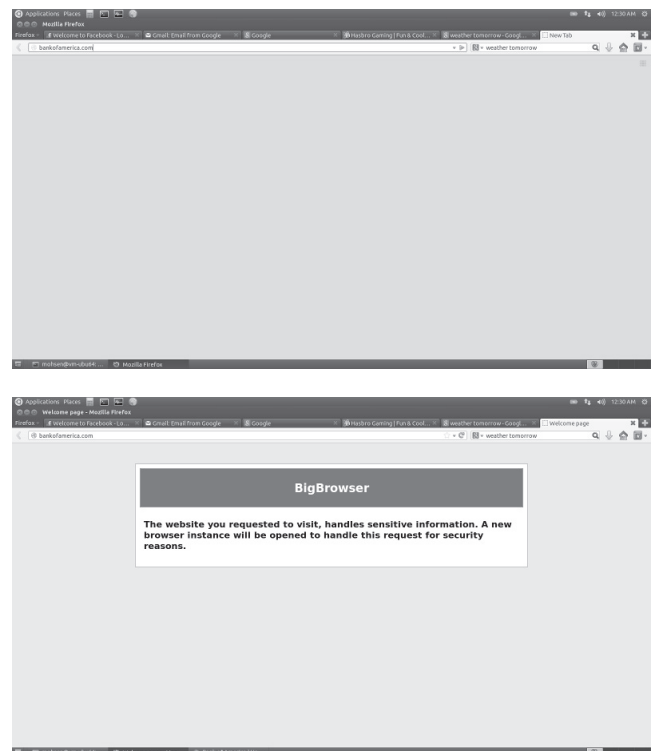


Figure 1: A request is made to www.bankofamerica.com (top figure). Instead of displaying the requested page in the current instance, Auto-FBI displays a message (bottom figure)

emphasizes the attacks capabilities rather than the minutiae of the attack mechanisms.

In this paper we assume that a browser cannot be compromised in a way that enables it to access system resources it is not supposed to access. For example, we assume that the browser will not arbitrarily read and write files it is not supposed to access.

3. USE MODEL

In this section, we describe Auto-FBI from a user's perspective without consideration to system design or implementation details. We have two implementations, one for novice users and one for experienced users.

Novice User.

A security mechanism has a better chance to be effective if it is well understood and easy to use. This is especially true for novice users. Our basic usage model is quite simple. For a novice user there are two kinds of sites. There are sensitive sites and *everything else*. Access to a sensitive site is done through a dedicated FBI. Access to all other sites is done through separate browser instance(s) as it would be done without Auto-FBI. When a user attempts to access a sensitive website by clicking on a link or entering a URL directly in the address bar, an FBI is created to handle the access. When the user finishes accessing the sensitive site, the user can close the FBI or minimize it. If the user later attempts to access the sensitive site from another browser instance, a new instance is activated again, or, if there is an existing active instance for the site, that instance is used to process the request. *For added security, the FBI is opened*

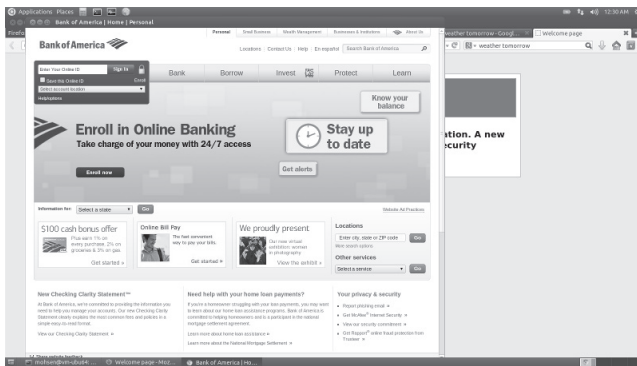


Figure 2: The requested page with sensitive content is accessed by a fresh instance of a different browser (Chrome in the figure, top window) while the browser that made the original request (bottom window) only displays a warning message

without an address bar, menus or tabs to emphasize that the access is meant for only one site. This simple usage model is preferable to a model that requires the user to define security policies which is not uncommon in mechanisms that are claimed to be transparent [6]. Access for a novice user is illustrated in Figures 1 and 2

Experienced User.

For the experienced user, we generalize our FBI approach to allow content from more than one domain to coexist within a browser instance. The idea is to allow content originating from a conflict of interest (COI) class to coexist within a browser instance. This is essentially the chinese wall policy [3] applied to content producers, which we view as principals, and to the browser which we treat as a delegate of the content producers (it executes code on their behalf). As an example, an experienced user might define a COI class that contains some specific financial institutions and another class that contains some specific news sources. In a given browser instance, content from sources that belong to the same COI class can coexist as separate tabs or as content within the same page if a page links to content from multiple sources in the same COI class.

An experienced user defines conflict of interest classes in a `classes` file. An example file is shown in Figure 3. The file consists of multiple sections, one for each COI class, but the first section specifies common domains, such as certificate authorities, that should be accessible by all COI classes. Each section starts with a line that defines the browser to be used followed by one or more lines that list the domains for the COI class. Any domain that does not appear explicitly in one of the COI classes or the first section is assumed to belong to an implicitly declared *general* COI class.

When a browser is launched by the user, it is associated with the general class. If the user attempts to access a site from a domain in a class other than the general class, then, if there is no active browser instance associated with that class, a new browser instance for that class is created to process and display the site. If there is an active browser instance for the class of the requested site, then the request is processed and displayed in that instance. To allow for using different browsers for different classes, e.g. using Google Chrome for e-commerce and Mozilla Firefox for banking, we specify the browser type rule that should be followed for each class with

```
# Common: includes Certificate Authorities, etc.
>
verisign.com
comodo.com
digicert.com
entrust.net
globalsign.com
tt.omtrdc.net
sb-ssl.google.com

# C1: Banking
>!
americanexpress.com
aexp-static.com
aexp.demdex.net
bankofamerica.com
usbank.com

# C2: E-Commerce
>C
amazon.com
ebay.com
ebaystatic.com
ebayimg.com
```

Figure 3: Example COI classes written by an experienced user

a type descriptor (that comes immediately after a `>` in the sample file):

- 'F': Mozilla Firefox instance should be used
- 'C': Google Chrome instance should be used
- '=': the launched browser instance should be the same type as the launcher
- '!': the launched browser instance type should be different from that of the launcher
- '_': places no restriction on the type of the browser

The implicit general class has the implicit type descriptor '_', so both Google Chrome and Mozilla Firefox can be used to browse websites from the general class.

The example file shown in figure 3 has two explicit COI classes (we will refer to them as C_1 , and C_2 as shown in the comments). The first class C_1 has type descriptor '!' and the second class has type descriptor 'C'. To demonstrate the user experience, let's consider the following scenario: user opens Google Chrome to start browsing, an instance of Google Chrome of the general class is created. The user can navigate to any site that is not listed in C_1 or C_2 in that browser instance. If the user tries to navigate to `www.bankofamerica.com` which is listed in C_1 , a fresh instance of Mozilla Firefox will be created to handle the request. That is because the type descriptor for C_1 specifies that an instance with a different type from that of the launcher (Google Chrome in this example) should be used. If the user tries to access `www.facebook.com` from the Mozilla Firefox instance, the request will be handled by the existing Google Chrome instance of the general class. That is because the general class has the implicit type descriptor '_' which allows the system to use the available instance. If the user navigates to `www.ebay.com` from any of the previous instances, a new instance of Google Chrome will be launched

to handle the request regardless of the browser type used to initiate the request.

4. SYSTEM DESIGN

The basic functionality that we are interested in is an *enforcement mechanism* with the ability to prevent a browser instance from communicating with sources that are not in its COI class and an *enabler mechanism* for starting FBI's as needed. We start by discussing the design alternatives.

4.1 Design Alternatives

There are essentially three ways to provide the proposed functionality: (1) inside the browser through modification of the browser code, (2) as a browser extension, or (3) outside the browser. We argue that accessing sensitive sites should be enforced through an enforcement mechanism that resides outside the browser and that launches FBI's. We briefly discuss each of the design options:

- **Enforcement inside the browser:** The desired functionality could be provided by modifying the browser to add the desired enforcement rules. This approach is highly browser-dependent and requires detailed understanding of the browser's code to be implemented. Also, it is not easily maintainable for new versions of a browser and requires different implementations for different browsers. Furthermore, if there are bugs in the implementation, it can compromise the browser's security. We do not recommend this approach because it does not provide a clean separation between the browser's code and the enforcement mechanism.
- **Enforcement in a browser extension:** Providing the desired enforcement as a browser extension is an improvement over enforcement inside the browser. In a browser like Chrome, extensions can be a combination of JavaScript and native code that is controlled by JavaScript. This is not advisable because the JavaScript engine is potentially exposed to malicious scripts from various sources. Any compromise of the engine through such malicious scripts can compromise the enforcement mechanism.
- **Enforcement outside the browser:** Enforcement outside the browser can be achieved by intercepting the browser's system calls and only allowing those calls that are according to policy. This has the advantage of separating the enforcement mechanism from the browser's code. In our implementation, the enforcement mechanism is around 2K lines of code and is browser independent. Adding that code as an extension would require writing different versions for different kinds of browsers. Also, enforcement outside the browser instance obviates the need to make additional restrictive assumptions about the power of the attacker. So, instead of adding an executing browser to the trusted computing base (TCB), we only consider an FBI to be part of the TCB to access sensitive content. The possibility that such an instance is compromised is smaller than that of an executing instance. We elaborate on this point in Section 5. Enforcement outside the browser does not preclude

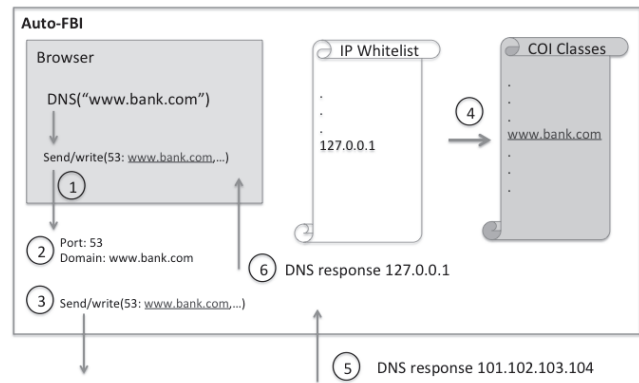


Figure 4: Execution scenario for handling access to content from a different COI class

implementing part of the *enabler mechanism* inside the browser. In fact, for technical reasons that we explain later in this section, this is the approach we take. Compromising the enabler mechanism would not allow an attacker to access sensitive content through the wrong browser instance.

4.2 System Components

The system has two main components. One component, *enforcer*, intercepts system calls and modifies them as needed to ensure that sensitive content is only accessed through the appropriate instance of the browser. Another component, *enabler* determines when sensitive content is requested and, if needed, launches a new fresh instance of the browser to access the content. These two components are independent and do not communicate with each other.

4.2.1 Enforcer

Intercepting System Calls.

To intercept system calls, we employed the `ptrace` API available in Linux. This API allows one program (tracer) to be notified and take control of every system call made by another program (browser). The kernel puts the browser to sleep when it makes a system call and notifies the tracer. The tracer can then modify the contents of process virtual memory or registers and then let the browser resume execution. When the system call gets executed and returns from the kernel, and before the control gets back to the browser, the kernel notifies the tracer once more. At this point the tracer can also modify the return value of the system call. Using this API enables us to control every interaction between the browser and the operating system. The `ptrace` API is also used by many sandboxing applications to control system access.

Among hundreds of system calls available in Linux, we are interested in the following calls: `connect`, `recvfrom` and `read`. The browser issues a `connect` system call when it needs to make a network connection, and it uses `recvfrom` or `read` to receive data from a network host.

Redirecting DNS Queries.

When a browser makes a DNS query, the tracer checks whether or not the requested network name is allowed to be accessed by the current browser instance. If it is not

allowed to be accessed, the tracer changes the IP addresses in the DNS response to a predefined IP address which is the loopback address in our implementation. This prevents the browser from connecting to the restricted website. At the same time, the attempted access is detected by the enabler mechanism of **Auto-FBI** and a new instance is launched as needed.

Intercepting IP Communication.

The IP addresses used by the browser are not always preceded by DNS queries. In fact for security reasons, IP addresses obtained from DNS queries are typically cached (IP pinning [13]) by browsers even if the TTL from the DNS response is set to 0. We distinguish between three kinds of IP addresses that can be used by a browser and for each kind we explain how it is handled.

- IP addresses that are cached by the browser and obtained from a previous DNS query. IP addresses that are obtained through DNS queries are cached by **Auto-FBI** in a *whitelist* if the original DNS query is not redirected. A later use of such an IP address is compared against **Auto-FBI**'s whitelist of IP addresses. If the IP address is not on the whitelist, it is blocked.
- Browser hardcoded IP addresses. We do not allow the use of any hard-coded IP addresses by the browser. We consider hardcoded addresses to be suspicious and a security risk.
- Hand-coded IP addresses. IP addresses that are directly entered by the user have legitimate usages such as setting up some LAN services (NAT setup for example). Such usages should be allowed. As a policy one can allow some LAN IP addresses and associate a class with each allowed address.

It is interesting to note here that associating a separate class with LAN addresses can prevent DNS rebinding attacks which are not completely solved by DNS pinning [13].

Figure 4 illustrates a scenario in which a browser instance whose COI class is different from that of the website it is trying to connect to. First it makes a DNS query to resolve the IP address (1), the DNS response is detected by the system (2) and the requested domain name is checked against the list of COI classes (3). Since the browser is running in a different class, the system rewrites the IP address in the DNS response (4) and returns the result to the browser. The browser would not have access to the actual IP address for **www.bank.com** and hence it is not able to connect to it. The browser could not communicate directly with **www.bank.com** using the actual IP address of **www.bank.com** because that IP address of **www.bank.com** is not added to the whitelist of the current instance.

4.2.2 Enabler: Launching Fresh Instances

To manage browser instances running in the system and launching new instances, we implemented a daemon which keeps track of all browser instances. The information on which new instances need to be launched comes from the extension component. The extension reports all URLs requested by the user to the daemon and also listens for incoming messages from the daemon in which case it will open the URL in a new tab. The daemon decide how to handle the URL sent by the extension: either open it in

an existing browser instance by sending a message to the extension running in that instance or open a fresh instance in case an appropriate instance is not running.

As we discussed in the previous section, having part of the enabler mechanism in the browser does not affect the security of our solution. Even if a browser is compromised and the extension is also compromised, all that would affect is the URL communicated by the extension, but that has no effect on the tracing of system calls and blocking them. That part is independent and is enforced through a mechanism that is completely outside the browser by the *Enforcer*. So, compromising the browser might affect the *progress* requirement of the browser, but does not affect the *safety* requirements of the enforcement.

5. SECURITY GUARANTEES

Arguing for the security of our approach is based on the fact that all access to content from one COI class is not allowed in a browser instance associated with another COI class. This is enforced by restricting communication to IP addresses that are on the whitelist for the COI class of the executing browser instance. The IP addresses on the whitelist of one class can be one of the following: (1) IP addresses corresponding to a domain from the common domain list, or (2) IP addresses of a domain in the COI class. So, if DNS is not compromised, only content from sources in a given COI class can run on a browser instance associated with that class. For sensitive content whose servers are typically authenticated, the assumption on DNS not being compromised is not needed and the system provides the desired isolation that ensures that there is no leakage of sensitive content between various instances (assuming the authentication mechanism is secure). This means that attacks such as CSRF or Clickjacking cannot be used to access sensitive content. In fact, content (scripts) used to launch such attacks are assumed to belong to COI classes that are not part of the sensitive COI classes. This is especially evident in the novice user scenario. For the experienced user, a non-judicious choice of classes can pose a security risk.

6. IMPLEMENTATION DETAILS

The prototype is composed of two separate modules: (1) Enforcer and (2) Enabler. These modules are sometimes referred to by different names: we use *tracer* to denote the enforcer module and *daemon* to refer to the main component of the enabler module. The implementation is entirely in user space and no modification to the Linux kernel is required.

6.1 Enforcer

Using Linux `ptrace` API, we can trace every system call made by a child process to keep track of the application's access to system resources including network and file system. This technique has been employed by sandboxing systems to limit the sandboxed program's access to system resources. However, this method inherently suffers from the Time Of Check Time Of Use race conditions.

There are two options for a tracer to start tracing the tracee: (1) `TRACEME` and (2) `ATTACH`. The first option is usually used in the following way: the tracer forks a new child process and in the child process calls `ptrace` with

TRACEME command, which causes it to be traceable by its parent process (the tracer), then it runs the target program by calling `exec`. The second option can be used to trace an already-running-process by attaching to it. The enforcer uses the TRACEME option by default (including the implementation for Firefox), but due to Chrome's use of the SETUID access flag in its sandboxing process¹, the enforcer uses the ATTACH option with Google Chrome. Using ATTACH leaves an enforcement gap: from the time that tracer launches Chrome until the tracer attaches to the browser, the browser is not traced. This should not be a source of vulnerability if the default home pages are safe. Using ATTACH means that part of the enforcer that is involved in setting up the tracing is browser dependent, but the rest of the enforcement when the ATTACH option is used and the whole enforcement using the TRACEME option are generic and not dependent on the browser.

The ptrace API allows one to trace a process and all its child processes and threads by specifying certain options: PTRACE_O_TRACEFORK and PTRACE_O_TRACECLONE. So we can make sure that a process cannot escape the tracer by creating a child process or a new thread.

For every system call made by the tracee, the tracer is notified twice: once before the call is handed over to the kernel for execution and once after the call returns from the kernel but before it is handed back to the tracee. So the tracer has two chances to modify a system call: it can change the parameters provided by the tracee to the kernel, or change the values returned by the kernel. To restrict the browser's access to websites, we need to monitor the following system calls:

- **connect**: Connects a socket file descriptor to a network address. It is also used for IPC.
- **rcvfrom**: Receives data through a socket file descriptor
- **read**: Can be used to receive data from the network (through a socket file descriptor)

6.1.1 DNS Queries

To prevent the browser from accessing certain websites, we keep track of the DNS queries made by the browser to find out if it is trying to connect to a restricted website. We can prevent the access by rewriting the IP addresses in the DNS response to point to a predefined address.

In order to accomplish this, we need to process the following system calls: `rcvfrom` and `read`. If the port number² matches the port number of the DNS protocol (53), we read the IO buffer from the tracee's memory and analyze the DNS message. If we decide to rewrite the IP addresses

¹Using TRACEME with Chrome would either require root privileges for running the browser which is not desirable or it breaks the browser's internal sandboxing mechanism which uses the SETUID access flag to allow its sandbox process to temporarily run with root privileges. The kernel does not allow a non-root process to trace a process with root privileges using the TRACEME option. In our solution, the enforcer waits until the privilege level is dropped after the sandbox is setup and then then attaches to the main browser process using the ATTACH option of `ptrace`.

²We have used `/proc` to figure out the remote port number of a socket. Alternatively, we could keep track of all sockets owned by the tracee and the port number specified during the connect call, but this method is error-prone. See [9] for a detailed discussion on why one should avoid replicating the kernel state.

in the DNS message, we do so by writing to the tracee's virtual memory³.

6.1.2 IP White-list

To prevent the browser from connecting to a restricted website by using a hard-coded IP address, we implemented a white-list of IP addresses that the browser is allowed to connect to. Initially, it is populated with the DNS service IP address (usually the local DNS client, 127.0.0.1) and a list of pre-defined IP addresses (to allow for LAN addresses). When the browser makes a DNS query, one of the following happens: either the response is changed to prevent access in case the URL is not allowed or the IP addresses in the response are added to the white-list. When the browser issues a `connect` system call, we would only allow it to go through if the IP address is in the white-list.

6.2 Enabler

6.2.1 Browser Extension

All browser instances are equipped with an extension that is responsible for two tasks: (1) report all user-requested URLs to the daemon and (2) open a new tab for URLs requested by the daemon. When the extension reports a URL to the daemon, the daemon decides if the URL should be handled by the reporting instance or by another instance. We have developed separate JavaScript extensions for Mozilla Firefox and Google Chrome⁴. Since the extensions cannot directly communicate with the daemon, we have a native application written in C that is responsible for forwarding messages between the daemon and the extension. The native application communicates at one end with the browser extension through the browser API (*Native Messaging* API in case of Google Chrome and *js-ctypes* in case of Mozilla Firefox) and it communicates with the daemon on the other end through Unix Domain Sockets. The native application's process is a child process of the browser instance and hence can be considered a part of the browser extension module. It is interesting to note that the code size of the extension (and its associated native application) is small compared to that of the enforcer: around 3K lines of code for the enforcer and daemon vs. 460 for the chrome extension and 280 for the Firefox extension.

6.2.2 Daemon

The daemon is a multi-threaded C program that uses Unix Domain Sockets to communicate with browser extensions. It is responsible for deciding how to handle restricted URLs by either dispatching it to an already-running instance that is allowed to access the URL or creating a fresh new instance.

For every browser instance running in the system, it has a separate thread to communicate with the instance. It keeps a record for each running browser instance along with a queue of outgoing messages for that instance in a shared data structure. When it receives a message from a browser extension, it chooses to either send a message to the browser

³We have used `process_vm_readv` and `process_vm_writev` system calls to read/write data from/to the tracee's virtual memory which is much more efficient than using `ptrace` for this task.

⁴Chrome and Firefox have different APIs for writing extensions: Google Chrome provides `chrome.*` API and Mozilla Firefox has Add-on SDK as part of the Jetpack project

extension in one of the running browser instances or run a fresh instance. It compares the COI class of the URL contained in the incoming message with that of the sender. If they match, it ignores the message, otherwise it decides how to handle the message according to the `classes` file.

6.2.3 Browser Instances and User Profiles

A browser instance, as defined in section 2, is associated with a *profile* that captures its execution state. In practice, most browsers support *user profiles* to enable the multi-user use case. User profiles are very similar in different browsers: a folder stored somewhere in the user's home directory that contains information such as the user's web history, cookies, cache, and preferences. Fortunately, most browsers also provide a way of using multiple profiles for a user in addition to the user's default profile. In case of Mozilla Firefox and Google Chrome, creating a fresh instance of the browser is as easy as creating an empty directory and instructing the browser to use that directory to store the user profile (though this would be cumbersome to do manually).

In our implementation, the default user profile (the one managed by the user) is associated with the general COI class. Other COI classes do not have a persistent profile: a temporary profile is created whenever the instance is created and it gets deleted after the instance is closed.

We think this way of managing user profiles is both usable and secure. User spends most of the time using the default profile since most URLs fall into the general COI category, hence allowing the user to manage that profile improves usability. Using temporary profiles for other COI classes ensures tighter security without much effect on usability.

6.3 Implementation Limitations

This section discusses some of the limitations of our current implementation. These are limitations of the implementation and not of the approach.

We have implemented the enforcer on 64-bit Linux system using the ptrace API. Working with ptrace API is highly architecture dependent, so porting the program to 32-bit Linux for example would require taking care of word size differences, system call differences, etc. Porting to other platforms or other operating systems might not be possible without major changes. Another limitation of the enforcer is that it does not support IPv6.

We developed the browser extension only for Chrome and Firefox. However, all major browsers support some form of plugin API that allows one to extend the browser functionality. We believe that the same functionality can be achieved for other browsers as well.

Another limitation is the restricted format of class definition file. A more powerful format can allow complex matching expressions for domain names. However, the format we chose was sufficient for our testing purposes.

The daemon process, which is responsible for opening URLs in fresh browser instances, could perform sanity checks on URLs to achieve higher security. For example it could remove the *query part*⁵ of the URL. This was not implemented in our prototype.

Our abstract definition of *profile* given in section 2 does

⁵An optional part in URLs that comes after a question mark in the URL and is usually in the form of key value pairs. For example in `http://www.example.com/test.jsp?x=1&y=2`, the last part (`?x=1&y=2`) is the query part.

not exactly match *user profiles* employed by browsers. Different browser instances even with different types might still be able to share some information with each other although they don't share the same *user profile*. Local shared objects (LSO for short) are among the methods that can be used for such purposes⁶. Our implementation does not address the problem of isolating such objects among different browser instances.

7. PERFORMANCE EVALUATION

We measured the amount of overhead incurred by the tracer module, by running the benchmarking extension from the Chromium project for a list of 20 most-visited websites. Table 1 shows the comparison of page load times for these websites for Chromium browser version 25 executed with and without the tracer. The benchmarking extension provides the ability to measure the load times with and without caching, which are shown in the table. For every URL the following 4 combinations are tested, each one averaged over 50 iterations:

1. Chromium running on its own and caching disabled (T_{norm}^1)
2. Chromium running with the tracer and caching disabled (T_{tracer}^1)
3. Chromium running on its own and caching enabled (T_{norm}^2)
4. Chromium running with the tracer and caching enabled (T_{tracer}^2)

Then the overhead is calculated by the following formula:

$$\Delta T^1 = T_{tracer}^1 - T_{norm}^1$$

$$\Delta T^1\% = \frac{\Delta T^1}{T_{norm}^1} * 100$$

And similarly for ΔT^2 . The last three rows of the table show the average over different sets of URLs. *Average for fast sites* shows the average for those sites that have $T_{norm} \leq 1000$ and *Average for slow sites* shows the average for those sites with $T_{norm} > 1000$. The last row shows the average over all 20 websites.

Although the relative overhead is large (about 30 %), the absolute delay is not significantly perceptible to the user and we argue that the overhead is acceptable. In fact for fast sites, the average overhead is less than 250 ms which in absolute terms is not significantly perceptible. For slow sites, the overhead is higher, but the relative overhead is lower. While we think that the overhead of the initial prototype is acceptable, we believe that further improvements to the overhead are needed.

We should note that due to variations in data transfer rates that are unavoidable, the overhead values are negative in a few cases.

⁶LSOs are similar to HTTP cookies. Websites which use Adobe Flash may store such objects on a user's computer. They can be shared among browsers that use the Flash Player plugin.

Website URL	Caching Disabled				Caching Enabled			
	T_{norm}^1	T_{tracer}^1	ΔT^1	$\Delta T^1\%$	T_{norm}^2	T_{tracer}^2	ΔT^2	$\Delta T^2\%$
www.youtube.com	800.6	1249	448.4	56.01	696.1	1114.5	418.4	60.11
www.yahoo.com	953.3	890.1	-63.2	-6.63	869.7	1133.3	263.6	30.31
login.live.com	701.1	817.8	116.7	16.65	655.9	498.4	-157.5	-24.01
www.msn.com	368.9	772	403.1	109.27	367.2	613.2	246	66.99
en.wikipedia.org	2101.3	2199.5	98.2	4.67	823.4	1037.6	214.2	26.01
blogsofnote.blogspot.com	522.1	1016.7	494.6	94.73	355.8	525.9	170.1	47.81
www.baidu.com	1433.3	954.6	-478.7	-33.40	848.5	589.3	-259.2	-30.55
www.microsoft.com	1269.5	2175	905.5	71.33	1606.3	1827.9	221.6	13.80
www.qq.com	6430.9	8179.3	1748.4	27.19	2717	3438.5	721.5	26.56
www.bing.com	184.1	267.5	83.4	45.30	172.6	256.7	84.1	48.73
www.ask.com	952	1184.5	232.5	24.42	891.8	1034	142.2	15.95
www.adobe.com	1394.6	2225.9	831.3	59.61	974.8	1626.2	651.4	66.82
www.taobao.com	2435.7	3286.1	850.4	34.91	1802.2	2625.4	823.2	45.68
twitter.com	401.6	593	191.4	47.66	325.4	447.9	122.5	37.65
www.youku.com	6427.5	5731.2	-696.3	-10.83	2009.1	2638.5	629.4	31.33
www.soso.com	947.1	983.5	36.4	3.84	647.4	672.5	25.1	3.88
wordpress.com	601.4	799.6	198.2	32.96	370	548.9	178.9	48.35
www.sohu.com	5601.8	7031	1429.2	25.51	4696.7	6596.9	1900.2	40.46
www.hao123.com	4598.4	4600.2	1.8	0.04	2276.5	2441.5	165	7.25
www.facebook.com	424.9	791.7	366.8	86.33	411.7	534.7	123	29.88
Average for fast sites	623.37	851.4	228.03	36.58	600.74	759.51	158.77	26.43
Average for slow sites	3521.44	4042.53	521.09	14.8	2517.97	3261.45	743.48	29.53
Overall (20 URLs)	1927.5	2287.4	359.9	18.67	1175.9	1510.1	334.2	28.42

Table 1: Performance evaluation for the tracer, times are in milliseconds

8. RELATED WORK

There is a vast literature on approaches to counter web attacks. We only summarize some of the most relevant works that are also representative of other works. We should also mention that private browsing modes provided by many browsers e.g. *Incognito* mode in Google Chrome, do not solve many of the problems addressed in our work. The reason for this is that they only separate browsing history and cookies from regular tabs. Moreover, there is no automatic separation of sensitive content from non-sensitive content in private browsing mode.

App Isolation.

Chen et al. [5] use an approach similar to ours for browser security. They provide a mechanism inside a single browser instance to isolate web applications from each other. Their approach requires the participation of web application providers to opt in and use their method. Our approach does not require such participation and is applicable to every website. Another major difference is that our isolation technique works outside the browser and we do not modify an already-large browser code base. This makes the software product more maintainable and also more trustable because of the smaller TCB size. While the number of lines in our code base is comparable to theirs ($\approx 2K$), their implementation depends on the Chromium's process architecture which already facilitates their job by sandboxing the rendering process and using different render processes for different origins when possible. If one implements their idea for another browser that does not have such process architecture (which currently includes almost all other major browsers including Firefox), it would require

much more changes to be made to the browser code base.

On the other hand, their approach provides two benefits compared to our method: they can isolate web applications with finer granularity than origins, namely for sets of URLs, but in our approach we can not control the browser from outside at that level of granularity. The other benefit of their approach is its performance overhead which is negligible compared to ours.

Multi-Layer MAC.

Hicks et al. [10] address the same problem with a different approach. They propose a multi-layer Mandatory Access Control (MAC) system to enforce end-to-end security goals. They implement a labeling system to achieve separation of data with different sensitivity. Their method requires collaboration from both client and server and also requires a modified browser to handle labeling and MAC enforcement. While their system is quite complicated, it can provide isolation at various levels of granularity. We argue that our proposed system is more practical because of its simplicity and small size while providing a coarse grained isolation of sensitive data.

FlowFox.

De Groef et al. [6] proposed an information-flow control scheme based on secure multi-execution technique [7]. They modified SpiderMonkey, the JavaScript engine used in Firefox and other browsers, to enforce information flow security policies. Their threat model is limited to information leakage, which is significantly weaker than the threat model we consider. Their approach is also different. Their solution works at the browser abstraction level while

our solution works at the operating system abstraction level. We argue that our approach is more flexible and more trustable because we do not introduce any code in the browser code-base. Web browsers have typically very large code-bases, and adding to this complex code might create more vulnerabilities. Also, operating system abstractions are more stable and have much less complexity compared to browser's internal abstractions.

Shadow Execution.

Capizzi et al. [4] addressed the information leakage problem in the context of desktop applications. Desktop applications might connect to network to get the latest updates for example. Since such applications normally run with user privileges, they have access to user's sensitive information and they might leak this information. The authors propose a technique called shadow execution in which they run two instances of the same application, one with access to user private data but no network access and another instance with no access to real user data but able to access the network. As our method, their method works at the operating system abstraction level, but their method is not suitable for controlling browsers.

9. CONCLUSION

We have shown that it is possible to provide effective protection against active and passive attacks on web browsers. Unlike other approaches, we manage to have a browser-independent mechanism to provide the required execution isolation needed to access different kinds of content. The overhead of the approach, while being acceptable, is not ideal and more work is needed to further reduce the overhead.

10. ACKNOWLEDGMENTS

We would like to thank the reviewers for many useful comments. This work has been supported in part through National Science Foundation grant CSR-0849980. The views and findings of this paper do not necessarily reflect those of NSF.

11. REFERENCES

- [1] A. Barth, C. Jackson, and J. C. Mitchell. Robust defenses for cross-site request forgery. In *Proceedings of the 2008 ACM conference on Computer and Communications Security*, CCS'08, pages 75–88, 2008.
- [2] N. Bielova, D. Devriese, F. Massacci, and F. Piessens. Reactive non-interference for a browser model. In *Proceedings of the 2011 international conference on Network and System Security*, NSS'11, pages 97–104, 2011.
- [3] D. F. C. Brewer and M. J. Nash. The chinese wall security policy. In *Security and Privacy, 1989. Proceedings., 1989 IEEE Symposium on*, pages 206–214. IEEE, 1989.
- [4] R. Capizzi, A. Longo, V. N. Venkatakrishnan, and A. P. Sistla. Preventing information leaks through shadow executions. In *Annual Computer Security Applications Conference, 2008, ACSAC'08*, pages 322–331, 2008.
- [5] E. Y. Chen, J. Bau, C. Reis, A. Barth, and C. Jackson. App isolation: get the security of multiple browsers with just one. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 227–238, 2011.
- [6] W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens. Flowfox: a web browser with flexible and precise information flow control. In *Proceedings of the 2012 ACM conference on Computer and Communications Security*, CCS'12, pages 748–759, 2012.
- [7] D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP'10, pages 109–124, 2010.
- [8] D. Flanagan. *JavaScript: the definitive guide*. O'Reilly Media, 2011.
- [9] T. Garfinkel. Traps and pitfalls: Practical problems in system call interposition based security tools. In *Proceedings of the 2003 Network and Distributed Systems Security Symposium*, volume 33 of *NDSS'03*, 2003.
- [10] B. Hicks, S. Rueda, D. King, T. Moyer, J. Schiffman, Y. Sreenivasan, P. McDaniel, and T. Jaeger. An architecture for enforcing end-to-end access control over web applications. In *Proceedings of the 15th ACM symposium on Access control models and technologies*, SACMAT'10, pages 163–172, 2010.
- [11] Internet Engineering Task Force (IETF). Request for Comments: 6265.
- [12] Internet Engineering Task Force (IETF). Request for Comments: 6454, 2011.
- [13] C. Jackson, A. Barth, A. Bortz, W. Shao, and D. Boneh. Protecting browsers from dns rebinding attacks. *ACM Transactions on the Web (TWEB)*, 3(1):2, 2009.
- [14] D. Jang, R. Jhala, S. Lerner, and H. Shacham. An empirical study of privacy-violating information flows in javascript web applications. In *Proceedings of the 2010 ACM conference on Computer and Communications Security*, CCS'10, pages 270–283, 2010.
- [15] M. Johns. On javascript malware and related threats. *Journal in Computer Virology*, 4(3):161–178, 2008.
- [16] E. Kirda and C. Kruegel. Protecting users against phishing attacks. *The Computer Journal*, 49(5):554–561, 2006.
- [17] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. All your iframes point to us. In *Proceedings of the 2008 Security Symposium*, SS'08, pages 1–15, 2008.
- [18] G. Rydstedt, E. Bursztein, D. Boneh, and C. Jackson. Busting frame busting: a study of clickjacking vulnerabilities at popular sites. *IEEE Oakland Web*, 2, 2010.
- [19] Z. Weinberg, E. Y. Chen, P. R. Jayaraman, and C. Jackson. I still know what you visited last summer: Leaking browsing history via user interaction and side channel attacks. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP'11, pages 147–161, 2011.