

Isolated Web Browsing

Enforcing OS-like Isolation for Browsers

Conny Franke, James Shearer, Daniel Zinn

December 16, 2005

Abstract

The rising popularity of web-based applications has forced the browser into a role for which it was not designed. Multiple applications run within the same process and have access to the browser's disk-based state (e.g. cookies), despite being conceptually separate. Most browsers attempt to address these issues, yet bugs are inevitable in software of such complexity. We investigate an orthogonal solution to these issues. Using our system, called *CleverNameHere*, we are able to separate distinct applications into separate OS processes, and appropriately restrict access to disk-based state.

1 Introduction

1.1 Problem Description

Aside from typical “surfing”, many people utilize their web browser for shopping, banking – even renting films. In fact, there is a staggering amount of human activity currently transacted online within a browser. And with web development techniques like AJAX [1] blurring the distinction between web applications and desktop applications, the trend towards browser-intensive computing is likely to grow. While these applications are certainly interesting and useful (e.g. Google Maps [3]), they force the browser into a role for which it was neither conceived nor designed. Browsers are not designed to provide protection between multiple applications running within them. In concept, they have become a application platform, while in design they are little changed from their earliest ancestors (rendering engines with navigation controls). If we use the browser as an application deployment platform, then it should provide the expected protection between running applications, much like any decent operating system (OS). Google Mail [2], for example, should not crash simply because the user browsed an ill-crafted web site in another tab. A malicious page should not have access to the state, either in memory or on disk, of the user's browser-based banking application. We propose a solution which address both of the aforementioned problems.

1.2 Threat Model

Our threat model is straightforward. We assume that the browser is benign, but potentially buggy. The sheer size of the typical browser code base supports this assumption. Further, we assume that there exist malicious web sites which attempt to use these browser bugs as a vector for attack. These attacks range from relatively harmless to severe. In the minimal case, a page could use a bug in the Javascript engine to cause the browser to crash, resulting in lost work and user frustration. More worrisome, a clever attacker could craft a page that harnesses browser flaws to inspect the Document Object Model (DOM) of another currently displayed page and examine, for example, banking information. Also, improper cookie access could allow a malicious (or simply unethical) site to track a user's browsing habits, resulting in a potentially serious privacy violation.

The remainder of our paper is organized as follows: In section 2, we discuss our general approach to the problem. In section 3, we provide a detailed description of our prototype architecture. We discuss our results, related work, and future work in sections 4, 5, and 6.

2 Approach

2.1 Goals

Our primary goals are to address the problems described in section 1.1. We desire a system in which each application runs in a separate OS process in order to achieve “crash protection,” and to mitigate the impact of implementation errors. Further, we want to appropriately restrict access to filesystem-based state; each instance should in-effect “own” its information.

We seek to accomplish the above in a manner orthogonal to the browser code base. We chose this requirement primarily for two reasons. First, it allows us to completely distrust the browser's code base and developers. For non-opensource browsers, like the widely-deployed Internet Explorer, this is our only option. Secondly, an orthogonal approach is more likely to be applicable to numerous platforms and browsers. An orthogonal approach is resistant to browser code restructuring, provided the behavior the browser exhibits (at a low level) remains unchanged.

We also strive to provide a solution which does not require user interaction to achieve the separation. Ideally, the browsing experience should be unchanged from that to which the user is already accustomed. Where our solution does depart from the typical browsing experience, we expect that it will not detract from usability.

2.2 High-Level Design

Our solution is simple in concept: make the OS perform the work it was built to do to achieve the desired browser behavior, and use low-level interception to

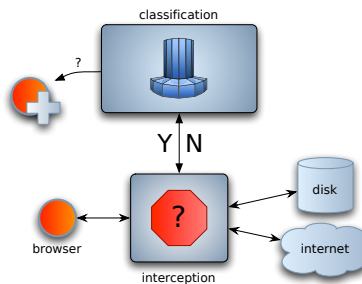


Figure 1: High-level System Description

enforce any other desired restrictions.

2.2.1 Terminology

The following terms describe concepts which are frequently used in the remaining portion of the paper.

Process Policy The logic or rules which govern when our system creates a new browser process.

Browsing Domain A grouping of related browsing activity. For example, clicking within a news website constitutes one browsing domain, while shopping at Amazon.com would constitute another.

Classification/Classifier The process by which network connection requests are translated into a browsing domain. The mapping between connection request and domain is deterministic.

To restate our problem in terms of the definitions above: We want to classify browser network activity into distinct domains according to a given policy, so that we can enforce separation based on the process policy.

2.2.2 OS Separation

We desire that each distinct web application, either in a tab or separate browser window, be protected from the other running applications. The process facility built into most operating systems provides this type of separation, gaining our system both crash protection and memory protection. However, we have to be judicious in how we utilize this functionality. We cannot simply open each link the user clicks in a new process, for this would clearly incur excessive overhead and yield a completely unusable system. Rather, we combine the output from the classifier with our process policy to determine when we should create a new browser instance. There is a one-to-one mapping between browser instances and browsing domains.

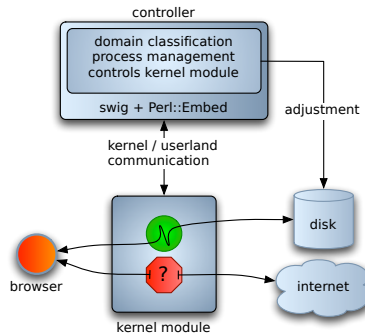


Figure 2: Implementation Details

2.2.3 Low-Level Interception

Our solution requires a mechanism by which we can restrict access to the filesystem based on browsing domain and gather the required network data needed for input to the classifier. We chose to use a system call interception scheme similar to those described in [5] and [4]. This technique is sufficiently low-level that we can intercept any browser-initiated communication we’d like. We can easily “see” when the browser tries to access the cookie storage file, for instance, by intercepting the `open()` system call. Figure 1 provides a graphical representation of our system.

3 Implementation

In order to demonstrate the technical feasibility of our approach, we constructed a working prototype. The following sections detail this construction, including our choices for each utilized technology. Figure 2 summarizes the components of our implementation.

3.1 Prototype Platform

For several reasons, we chose to develop our prototype on Linux. Its open source nature was a major benefit, as we needed to develop kernel code and “hijack” system calls. Access to the system source was helpful in overcoming several obstacles encountered during the development of our prototype. Furthermore, there exists a large body of information regarding kernel development, system call interception, and kernel-userland communication within Linux available on the Internet.

3.2 Mozwatcher: the kernel module

We developed a kernel module, nicknamed Mozwatcher, which was responsible for a variety of activities. First, it is responsible for the interception of

```

1 extern void *sys_call_table;
  EXPORT_SYMBOL(sys_call_table);

```

Listing 1: Exporting the system call table in kernel/kallsyms.c

```

2 static ssize_t (*sendto_old)(int , const void *, size_t, int,
   const struct sockaddr *, socklen_t);

4 void hookSendto() {
   extern long sys_call_table[];
6   // save the old one
   sendto_old = (ssize_t (*)(int, const void *, size_t,
8     int, const struct sockaddr *, socklen_t *))
     (sys_call_table[__NR_sendto]);

10  // hack in ours
   sys_call_table[__NR_sendto] = (unsigned long)sendto_mine;
12  printk("Hook: sendto registered. (hijacking syscall #%d)\n", __NR_sendto);
   }

14 void removeSendToHook() {
16   extern long sys_call_table[];
   printk("Hook sendto Module removed.\n");
18   // put back the real syscall
   sys_call_table[__NR_sendto] = (unsigned long)sendto_old;
20 }

```

Listing 2: Redirecting the sendTo system call

network and filesystem-related system calls for browser instances under observation. In order to achieve this, we redirected the normal system call entries in the `SYS_CALL_TABLE` to our custom crafted functions. Mozwatcher is also responsible for communicating network accesses to the classifier and acting on the responses obtained thereafter.

Kernel modification Starting from version 2.6 the Linux kernel does not export the `sys_call_table` symbol. Therefore, we patched the kernel by adding the lines displayed in listing 1 to *kernel/kallsyms.c*.

Kernel module initialization and removal During the module initialization, Mozwatcher redirects the system calls `open`, `sendto`, and `recvfrom` to its own implementations. Listing 2 shows the main steps that are needed for redirecting the `sendto` system call. In lines 1 and 2 a static variable is used to save the old address. “Hijacking” the system call is straightforward: after saving the old address (line 7-9), the address of our customized version is written into the kernel’s dispatching table. When the module is removed from the kernel, the changes are undone (lines 15-20).

During browser execution The kernel module maintains a list of PIDs of currently executing controller processes, as well as PIDs of browser instances observed by those controllers. This is required to properly intercept system calls

initiated by the browser and to accurately direct messages regarding intercepted system calls to the controllers.

3.3 Classifier

As we expected the classifier to change repeatedly during the development of the prototype, we wrote it in Perl. This allowed for very rapid development and eased mundane development tasks, such as saving discovered domains to file. Further, the high-level data structures and strong pattern matching that Perl provides made it an excellent choice for this component.

The classifier is essentially the heart of our system, and its domain decisions determine when to create a new processes. As such, it's key that the classifier makes accurate decisions: inaccuracy decreases system performance and enlarges the delta between typical browsing behavior and behavior under our system. When classifying an instance, we have the following information available: the host name accessed, the HTTP request string, and the server response. This response includes not only data, but also the server-provided MIME type of the data.

As our goal was not to research the best possible classification rules, but to prove the technical feasibility of our system, we chose a simplistic classification rule. Network activity is classified strictly on the accessed network domain. For example, `http://www.ucdavis.edu/` would map to the same domain as `http://www.ucdavis.edu/~user`. While this rule would lead to very coarse domains, and thus, limited file system-state protection, it was sufficient for our purposes.

3.4 Messaging Framework

Because our prototype required communication between kernel space and user land, we developed a framework for easily passing messages between Mozwatcher and the controller. Linux Netlink sockets, a framework specifically designed to provide communication between Linux kernel modules and userland processes, provided the backbone of this communication system. We implemented an abstraction layer atop Netlink sockets such that we could easily pass data of arbitrary types back and forth in a type safe manner. We used C++ templates to ease development of the messaging userland code, yet such constructs are prohibited in kernel code. Instead, we provided similar functionality using C macros. Listing 3 demonstrates the messages used in our prototype and shows usages in both kernel and userland code. Listing 4 demonstrates message creation, packaging, and transmission to the kernel module. Similarly, listing 5, shows the kernel's type-sensitive message dispatching.

3.5 Controller

The controller is the central switchboard of our prototype. As such, its responsibilities are fairly mundane. It manages communication to and from the kernel for

```

2 // declare MessageIDs
enum MessageID {
3     // ...
4     REQUESTOK,
5 };
6
7 // data structures to pass in between kernel and userland
8 struct requestStatus {
9     int pid;
10    int socket;
11 };
12
13 // Associate MessageIDs with data types
14 DECLARE_MSG_T(INITIALIZE, int);
15 // tells Mozwatcher the pid of the supervisor process
16 DECLARE_MSG_T(NEWMOZINSTANCE, int);
17 // tells Mozwatcher pid of new browsing instances
18 DECLARE_MSG_T(REMOVEMOZINST, int);
19 // tells Mozwatcher to remove a watched browsing instance
20 DECLARE_MSG_T(SENDCAPTURED, struct sendCap);
21 // pass captured send request to userland daemon
22 DECLARE_MSG_T(REQUESTOK, struct requestStatus);
23 // tell Mozwatcher that receiving the answer is allowed
24 DECLARE_MSG_T(REQUESTNOTOK, struct requestStatus);
25 // tell Mozwatcher that receiving the answer is not allowed
26 // ...

```

Listing 3: Message declarations in mess.h

```

// Usage in userspace ...
2 void requestOK(int pid, int sock) {
3     Message<REQUESTOK> req;
4     req().pid = pid;
5     req().socket = sock;
6     sendMess(req);
7 }

```

Listing 4: Using the messaging framework in userland

```

1 // Inside the Mozwatcher global message receiving callback
struct MESS *m = (struct MESS*)(NLMSG_DATA(nlh));
3 switch (m->id) {
4     // ... more message dispatching here
5     case REQUESTOK:
6         requestOK(((struct REQUESTOK *)m)->content, skb);
7         break;
8     default: printk("ERROR: Unhandled Message Type %d\n", m->id);
9 }
10
11 // Message handlers for messages
12 void requestOK(struct requestStatus s, struct sk_buff *skb) {
13     // callback routine for requestOK message
14     struct mozInst *mi = findMozInst(s.pid);
15     if (!mi) return;
16     printk("For pid %i, it is ok to receive data on socket %i \n", s.pid, s.socket);
17     // ...
18 }

```

Listing 5: Using the messaging framework in kernel space

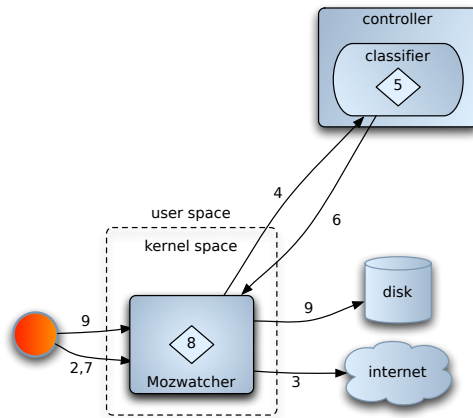


Figure 3: Operational Example

the other userland components. Further, it handles the creation of new browser instances when the classifier determines the user has crossed into a new domain, and messages Mozwatcher with the new process ID. The controller includes an interactive shell to quickly and easily test changes to the framework without recompiling and restarting the entire system. This interactive shell communicates via the messaging framework described above. Because the controller must provide two-way communication with components written in both Perl and C, we utilized the Swig framework and Perl::Embed.

3.6 Filesystem State Manager

The filesystem state manager is responsible for redirecting an attempted access to the global browser filesystem state to state specific to the accessing process' domain. Filesystem state includes cookies, passwords, and cache.

3.7 Example Operation

Figure 3, along with the steps outlined below, provide an example of the typical operation of our system.

1. At startup, the controller tells Mozwatcher its PID and the PID of the initial browsing instance. At this point, the instance is not in any browsing domain, and thus has no access to filesystem-based state. This step is not shown.
2. The browser instance attempts to access a web site using the `sendto` system call.
3. Mozwatcher allows the `sendto` to proceed as normal.

```

1 my $domain = $classSingleton->classify($req);
2 if (not defined $FireFoxInstDomain->{$pid}) {
3     print "Classify $pid into Domain $domain -- ACCEPT\n";
4     $FireFoxInstDomain->{$pid} = $domain;
5     swig::requestOK($pid, $sock);
6 }
7 elsif ($FireFoxInstDomain->{$pid} == $domain) {
8     print "firefox $pid wants to access its Domain $domain -- ACCEPT\n";
9     swig::requestOK($pid, $sock);
10 }
11 else {
12     print "firefox $pid wants to access foreign domain: " .
13           " ($domain != " . $FireFoxInstDomain->{$pid} .
14           " -- DENY\n";
15     swig::requestNOTOK($pid, $sock);
16     newMoz($req->{host} . $req->{URL});
17 }

```

Listing 6: Controller Code (controller.pl)

4. Because this is an PID under observation, Mozwatcher packages the HTTP request and sends it to the controller / classifier via the messaging framework.
5. The classifier classifies the request. If the browser is not yet associated with a domain, it is assigned this new domain, and the classifier responds ACCEPT. If the browser is already in a different domain, the classifier responds DENY. Otherwise, the browser was already in this domain, and the classifier also responds ACCEPT. In the case of DENY, the controller will create a new browser instance and direct it to the requested site and inform Mozwatcher of the new instance (see listing 6).
6. The classifier communicates the response to Mozwatcher, which associates the response with the appropriate PID.
7. The browser attempts a `recvfrom` system call.
8. Based on the earlier response from the classifier, Mozwatcher either blocks or permits this system call.
9. Later, when the browser attempts to access some filesystem-based state, it will receive the data corresponding to its current domain.

4 Results

We succeeded in developing an orthogonal, working prototype, and are convinced of the technical feasibility of our approach to achieving isolated web browsing. However, the various difficulties we encountered suggest that there are likely better approaches. We investigate some possibilities below.

5 Related Work

In [8], Miller and Stiegler describe a secure Internet browser based on capabilities. However, this browser requires a specialized, secure environment to operate properly.

Jackson et al. describe several possible attacks on user privacy using cookie manipulation in [6]. Using a Firefox extension they developed, the authors significantly increased the security of the web browser. Note that using the orthogonal approach we have presented would not only solve specific issues related to insufficient cookie checking, but would also greatly reduce the opportunities for information leakage from one browsing domain to another¹.

In [7], the authors suggest dividing the monolithic ssh-daemon into a small, privileged controller process (for forking user shells) and a unprivileged process that handles the remaining work. Our proposed division of the monolithic browser into several processes is inspired by this approach. However, our main goal is to protect *equal-privileged* browsing instances from each other, rather than isolating high-privilege code from normal code.

6 Future Work

6.1 Improved Classification

Our prototype used a rather simple set of classification criteria. When developing this system, we uncovered several situations for which our minimal approach will not suffice. The following are some suggestions for classifier improvement.

With our simple classification scheme, a site which uses off-site images, such as ad banners, would erroneously result in multiple domains. Accounting for such site composition could be solved by

MIME type examination Examining the MIME type of the server response allows the classifier to ignore requests that deal with non-hazardous data, such as images, movies, etc.

HTML content examination A more sophisticated classifier could examine the returned HTML, parse the document structure, and utilize this information for later classification of embedded elements.

Other examples of classification problems which our approach does not address include

classification contradictions If classification includes data from the HTTP request string, contradictions sometimes result. For example, one would expect that `http://www.amazon.com/cart` would reside in the same browsing domain as `http://www.amazon.com/checkout`. However, it is unlikely that a user would expect `http://www.ucdavis.edu/~franke` to be classified identically to `http://www.ucdavis.edu/~zinn`.

¹This does not include cases where the information is encoded in the HTTP request string, in which case the user likely *wants* the information to cross domains.

frames difficulties While this is no longer a dominant web design paradigm, frames usage could pose a classification problem. If a site is composed of frame content from distinct hosts, there is no obvious classification policy.

6.2 Non-orthogonal Approach

With open-source browsers like Firefox, it is possible to investigate a non-orthogonal approach to the problem. One possible modification is to deconstruct the browser into multiple processes with well-defined interfaces. With such a separation, the portion of the browser that contains runtime state could be made very small. In such a system, creating new browsing instances would entail far less overhead than in our example implementation. Further, this modified Firefox would likely provide opportunities for more specific classification, as we would be able to inspect the internal structures of the browser at various points. For example, our prototype was unable to handle HTTPS requests because the classification data was encrypted at our point of observation. A browser with separated HTTPS management would allow us to observe the unencrypted traffic as needed. Also, the described system would allow a user to provide, for example, his or her *own* cookie management code. She can selectively choose to trust or distrust certain parts of the browser, and act accordingly.

References

- [1] AJAX. <http://en.wikipedia.org/wiki/AJAX>.
- [2] GMail. <http://mail.google.com>.
- [3] GoogleMaps. <http://maps.google.com>.
- [4] Tal Garfinkel, Ben Pfaff, and Mendel Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *NDSS*, 2004.
- [5] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications. In *Proceedings of the 6th Usenix Security Symposium*, San Jose, CA, USA, July 1996.
- [6] Collin Jackson, Andrew Bortz, Dan Boneh, and John C. Mitchell. *Protecting Browser State from Web Privacy Attacks*. Manuscript. Stanford Security Lab, 2005.
- [7] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. Technical report, University of Michigan, 2002.
- [8] Marc Stiegler and Mark Miller. A Capability Based Client: The DarpaBrowser. Technical report, Combex, Inc., 2002.