A Safety-Oriented Platform for Web Applications

Richard S. Cox[†], Jacob Gorm Hansen[‡], Steven D. Gribble[†], and Henry M. Levy[†]

[†]Department of Computer Science & Engineering
University of Washington
{rick, gribble, levy}@cs.washington.edu

[‡]Department of Computer Science
University of Copenhagen
jacobg@diku.dk

Abstract

The Web browser has become the dominant interface to a broad range of applications, including online banking, Web-based email, digital media delivery, gaming, and e-commerce services. Early Web browsers provided simple access to static hypertext documents. In contrast, modern browsers serve as de facto operating systems that must manage dynamic and potentially malicious applications. Unfortunately, browsers have not properly adapted to their new role. As a consequence, they fail to provide adequate isolation across applications, exposing both users and Web services to attack.

This paper describes the architecture and implementation of the Tahoma *Web browsing system. Key to Tahoma is the* browser operating system *(BOS), a new trusted software layer on which Web browsers execute. The benefits of this architecture are threefold. First, the BOS runs the client-side component of each Web application (e.g., on-line banking, Web mail) in its own virtual machine. This provides strong isolation between Web services and the user's local resources. Second, Tahoma lets Web publishers limit the scope of their Web applications by specifying which URLs and other resources their browsers are allowed to access. This limits the harm that can be caused by a compromised browser. Third, Tahoma treats Web applications as first-class objects that users explicitly install and manage, giving them explicit knowledge about and control over downloaded content and code.*

We have implemented a prototype of Tahoma using Linux and the Xen virtual machine monitor. Our security evaluation shows that Tahoma can prevent or contain 87% of the vulnerabilities that have been identified in the widely used Mozilla browser. In addition, our measurements of latency, throughput, and responsiveness demonstrate that users need not sacrifice performance for the benefits of stronger isolation and safety.

1 Introduction

The 1993 release of the Mosaic browser sparked the onset of the modern Web revolution [24]. The nascent Web was a hypertext document system for which the browser performed two functions: it fetched simple, static content from Web servers, and it presented that content to the user. A key Web feature was the ability for one Web site to link to (or embed) content published by other sites. As a result, users navigating the early Web perceived it as a vast repository of *interconnected, passive documents*.

Since that time, the Web has become increasingly complex in both scale and function. It provides access to an enormous number of services and resources, including financial accounts, Web mail, archival file storage, multimedia, and e-commerce services of all types. Users transfer funds, purchase tickets, file their taxes, apply for employment, and seek medical advice through the Web. Their perceptions of the Web have evolved, as well. Today's users see the modern Web as a portal to a collection of *independent, dynamic applications* interacting with remote servers. Moreover, they expect that Web applications will behave like the applications on their desktops. For example, users trust that Web applications are sufficiently isolated from one another that tampering or unintended access to sensitive data will not occur.

To respond to the demands of dynamic services, the browser has evolved from a simple document rendering engine to an execution environment for complex, distributed applications that execute partially on servers and partially within clients' browsers. Modern Web browsers download and execute *programs* that mix passive content with active scripts, code, or applets. These programs: effect transactions with remote sites; interact with users through menus, dialog boxes, and pop-up windows; and access and modify local resources, such as files, registry keys, and browser components. The browser, then, has transcended its original role to become a de facto *operating system* for executing client-side components of Web applications.

Unfortunately, current browsers are not adequately de-

Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P'06)
1081-6011/06 $20.00 © 2006 IEEE

signed for their new role and environment. Despite many attempts to retrofit isolation and security, the browser's original roots remain evident. Simply clicking on a hyperlink can cause hostile software to be downloaded and executed on the user's machine. Such "drive-by downloads" are a common cause of spyware infections [23]. Trusted plug-ins may have security holes that permit content-based attacks. Browser extensibility features, such as ActiveX components and JavaScript, expose users to vulnerabilities that can potentially result in the takeover of their machines.

Users assume that Web applications cannot interfere with one another or with the browser itself. However, today's browsers fail to provide either kind of isolation. For example, attackers can take advantage of cross-site scripting vulnerabilities to fool otherwise benign Web applications into delivering harmful scripted content to users, leaking sensitive data from those services. Other browser flaws let malicious Web sites hijack browser windows [26] or spoof browser fields, such as the displayed URL [37]. Such flaws facilitate "phishing" attacks, in which a hostile application masquerades as another to capture information from the user.

Overall, it is clear that current browsers cannot cope with the demands and threats of today's Web. While holes can be patched on an ad hoc basis, a thorough re-examination of the basic browser architecture is required. To this end, we have designed and implemented a new browsing system architecture, called *Tahoma*. The Tahoma architecture adheres to three key principles:

1. *Web applications should not be trusted.* Active content in today's Internet is potentially dangerous. Both users and Web services must protect themselves against a myriad of online threats. Therefore, Web applications should be contained within appropriate sandboxes to mitigate potential damage.
2. *Web browsers should not be trusted.* Modern browsers are complex and prone to bugs and security flaws that can be easily exploited, making compromised browsers a reality in the modern Internet. Therefore, browsers should be isolated from the rest of the system to mitigate potential damage.
3. *Users should be able to identify and manage downloaded Web applications.* Web applications should be user visible and controllable, much like desktop applications. Users should be able to list all Web applications and associated servers that provide code or data, and ascribe browsing-related windows to the Web applications that generated them.

By following these principles, Tahoma substantially improves security and trustworthiness for both users and Web services. Users gain knowledge and control of the active Web content being downloaded and executed. Web services gain the ability to restrict the set of sites with which their applications can communicate, thereby limiting damage from hijacked browsers. Active Web content and the browser that interprets and renders it are isolated in a private virtual machine, protecting the user's desktop from side-effects, malicious or otherwise.

The idea of sandboxing Web browsers is not new. For example, VMware has recently release a virtual-machine-based "Web browser appliance," containing a checkpointed image of the Firefox browser on Linux [32]. As another example, GreenBorder [12] augments Windows with an OS-level sandbox mechanism similar to BSD jails [17], in order to contain malicious content arriving through Internet Explorer or Outlook. Our work uses VMs to provide strong sandboxes for Web browser instances, but our contribution is much broader than the containment this provides. Tahoma isolates Web *applications* from each other, in addition to isolating Web browsers from the host operating system. As well, Tahoma permits Web services to customize the browsers used to access them, and to control which remote sites their browser instances can access.

We have implemented a prototype of the Tahoma browsing system using Linux and the Xen virtual machine monitor [4] and modified the Konqueror browser to execute on top of it. Our experience shows that the Tahoma architecture is straightforward to implement, protects against the majority of existing threats, and is compatible with existing Web services and browsers. We also demonstrate that the benefits of our architecture can be achieved without compromising user-visible performance, even for video-intensive browsing applications.

The remainder of this paper is organized as follows. Section 2 defines Tahoma's high-level architecture and abstractions. Section 3 describes the implementation, while Section 4 evaluates our prototype with respect to both function and performance. Section 5 presents related work, and we conclude our discussion in Section 6.

2 Architecture

The Tahoma architecture has six key features:

1. It defines a new trusted system layer, the *browser operating system* (BOS), on top of which browser implementations (such as Netscape or IE) can run.
2. It provides explicit support for *Web applications*. A Web application consists of a *browser instance*, which includes a client-side browser executing dynamic Web content and code, and a *Web service*, which is a collection of Web sites with which the browser instance is permitted to communicate.

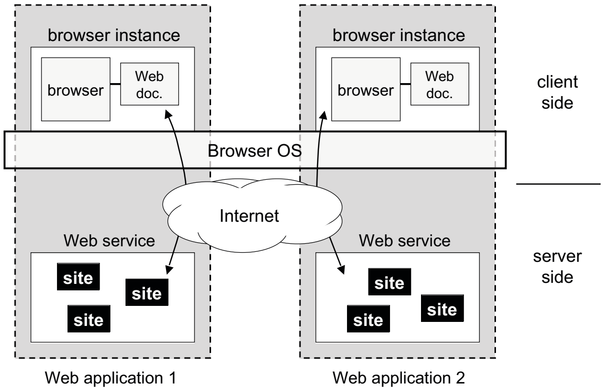

Figure 1. The Tahoma architecture. A Web application consists of two components: a client-side browser instance and a remote Web service. The browser operating system isolates browser instances from each other and also restricts the Web sites with which they can communicate.

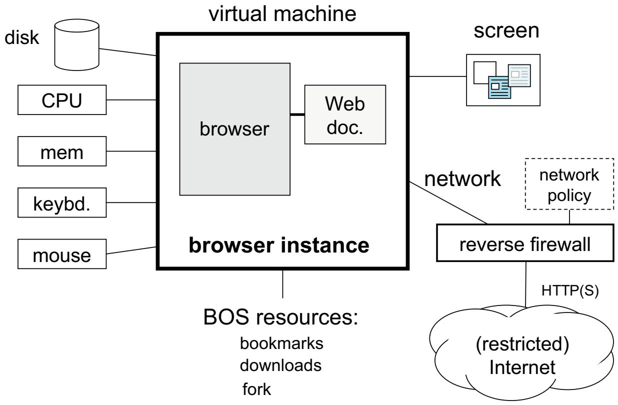

Figure 2. The execution environment of a browser instance. The browser instance executes in a virtual machine sandbox that provides access to private resources and a restricted subset of the Internet.

3. It enforces *isolation* between Web applications, prohibiting one application from spying on or interfering with other applications or host resources. Each Web application has an associated browser instance that is sandboxed within a virtual machine [4, 34].
4. It enforces *policies* defined by the Web service to control the execution of its browser instances, e.g., to restrict the set of Web sites with which a browser instance can interact. A Web service provides the BOS with a *manifest* – an object that defines its policies and other Web application characteristics.
5. It supports an *enhanced window interface* for browser instances. The BOS multiplexes windows from multiple instances onto the physical screen and authenticates the Web application for users.
6. It provides *resource support* to browser instances, including window management, network communication, bookmark management, and the execution or "forking" of new Web applications.

Figure 1 shows a simplified, high-level view of the Tahoma architecture. It identifies two Web applications, each consisting of a client-side browser instance and a remote Web service. Each browser instance comprises a browser and the Web documents that it fetches, executes, and displays. The browser operating system, shown below the browser instances, isolates Web applications from each other, preventing resource sharing or communication between browser instances. The BOS also manages communication between a browser instance and the Internet, permitting access to those sites (and only those sites) in the associated Web service.

We now describe the key components of this architecture in more detail. Table 1 clarifies the terminology that we will use throughout this paper.

2.1 Web Applications

Figure 2 shows the execution environment as viewed by a client-side browser instance. Each browser instance executes in a virtual machine (VM) that has its own private virtual disk, CPU, memory, input devices, and screen. The VM also supports a virtual network, through which the browser instance interacts with remote Web sites. Unlike conventional browsers, which can browse and display multiple Web sites simultaneously, each Tahoma browser instance is associated with a *single*, well-circumscribed Web application, e.g., an application that provides online access to the user's bank. Thus, Tahoma users have a unique browser instance associated with each running Web application.

The VM environment provided for the browser instance has several advantages. First, a Web application is safe from interference by other applications, and it is free to modify any machine state, transient or persistent, without endangering other applications or the user's host OS. Second, the user can easily remove *all* local effects of a Web application by simply deleting its VM. Finally, the VM environment increases flexibility for the programming of Web applications. For example, a service could provide a customized browser rather than using a commodity browser, or it could upload a highly optimized application in x86 machine code.

Tahoma Web applications are first-class objects and are explicitly defined and managed. The Web service specifies the characteristics of its application in a manifest, which the BOS retrieves when it first accesses the service. The man-

Tahoma term	analogous OS term
Web documents: static and active content (e.g., HTML or JavaScript) that a browser fetches from a Web service	documents and scripts
browser: client-side software (e.g., Firefox, IE) that interacts with a user to fetch, display, and execute Web documents	program
virtual machine: a virtual x86 machine that provides the sandboxed execution environment for an instantiated browser	process
browser instance: a browser executing in a virtual machine; a client has one browser instance per executing Web application	program instance
Web application: the union of a client-side browser instance and a remote Web service that cooperate to provide the user with some application function (e.g., online banking, Webmail)	distributed application

Table 1. Terminology. This table explains key terms in the Tahoma architecture, and provides the closest analogous term in a conventional operating system environment.

ifest includes several key pieces of information. First, it presents a digital signature authenticating the Web service to the client. Second, it specifies the code that will run in the browser instance; it can name a conventional Web browser, or it can specify arbitrary code and data to be downloaded, as described above. Third, it specifies Internet access policies to be enforced by a reverse firewall. These network policies define the set of Web sites or URLs that the browser instance is allowed to access.

Network policies protect the Web application from compromised browsers. Browsers are easily compromised by malicious plug-ins or through active Web content that exploits security holes in the browser or its extensions. A compromised browser could capture confidential data flowing between the browser instance and its Web service and send that information to an untrusted Internet site, or it could use the browser instance as a base to attack other Internet hosts. The network policy and reverse firewall aim to prevent these attacks by restricting communication from the browser instance to legitimate sites within the Web service.

Users accessing a Web application for the first time must *approve* its installation. Only then will Tahoma create a new virtual machine, install within it the browser code and data, and execute the new browser instance. The BOS caches approvals, so the user need not re-approve a Web application on subsequent executions.

2.2 The Browser Operating System

The browser operating system is the trusted computing base for the Tahoma browsing system. It instantiates and manages the collection of browser instances executing on the client. To do this, it must multiplex the virtual screens of each browser instance onto the client's physical display, enforce the network policies of each instance, and durably store state associated with browser instances, bookmarks, and manifests.

Figure 3 shows a detailed architectural view of the browser operating system. The figure contains three Web applications and their isolated browser instances. Two of the browser instances contain conventional browsers, while the third is executing a custom radio application instead of a conventional Web browser.

The BOS provides the highest level user interface, letting users manipulate the virtual screens of each browser instance. In addition, it wraps each virtual screen with a border that the browser instance cannot occlude. In the border, the BOS provides trusted information to the user, such as the name and credentials of the Web application with which the screen is associated. The BOS routes input events to the appropriate browser instance, similar to the way conventional window systems operate.

The BOS also provides users with control panels and bookmark management tools. These let the user install, execute, and uninstall Web applications, or create bookmarks that point to documents within a Web application. A bookmark has a familiar meaning in the context of a conventional Web browser. However, a Web application that provides its own custom browser instance may co-opt bookmarks for its own purposes. For example, a streaming radio service could use bookmarks to implement radio channels.

The BOS mediates all network interactions between a browser instance and remote Web sites. To access the Web, a browser instance invokes a BOS system call that fetches Web documents over HTTP. The BOS will service the connection only if the document falls within the network policy specified in the instance's manifest. If not, the BOS refuses the request. If the document is allowed by the manifest of a different Web application, the BOS gives the user the option of loading it into that Web application's browser instance.

Web applications have durable state that the BOS must manage. Sandboxes provide private virtual disks to browser instances, and the BOS maintains the state of these disks between invocations of the Web application. It also stores a set of "stock" browser instances (e.g., Mozilla) that can be cloned when installing a Web application. Finally, the BOS stores manifests and bookmarks associated with Web applications. It treats all long-term storage as a soft-state cache.

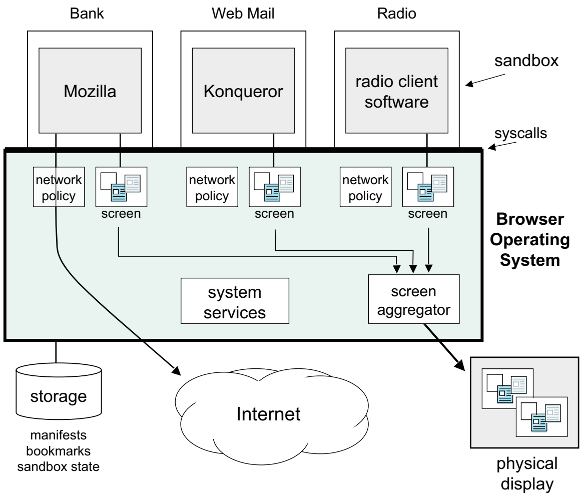

Figure 3. The browser operating system (BOS). The BOS instantiates and manages sandboxes, stores long-term state associated with browser instances, enforces the network access rights of browser instances, and aggregates their virtual screens into the client's physical display.

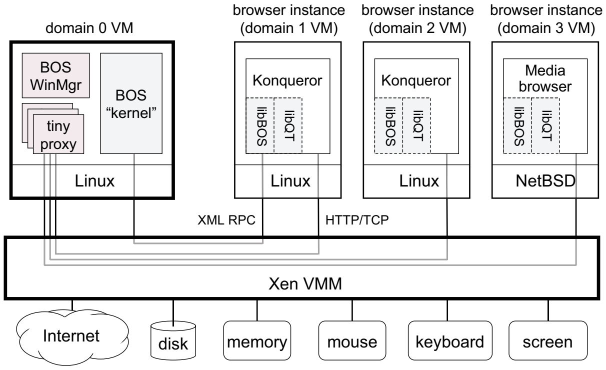

Figure 4. The Tahoma implementation. The Xen virtual machine monitor isolates browser instances by sandboxing them in virtual machines. Processes running within the privileged "domain 0" VM provide BOS services. Additional services are provided by libraries embedded in browsers.

Accordingly, durable state can be evicted, but at the cost of having to re-download manifests or re-install browser instances when the user next accesses a Web application.

We describe our implementation choices in Section 3. However, the BOS is designed to be implementable in different ways. For example, it could run in its own virtual machine, with browser instances running in separate virtual machines with their own guest operating systems. Alternatively, it could be implemented as a virtual machine monitor running directly on the physical hardware, with browser instances running in VMs above it.

2.3 Summary

The Tahoma architecture is driven by the principles described in Section 1: distrust of Web browsers and applications, and the empowerment of users. The resulting architecture isolates Web applications, protecting other applications and client resources from malicious downloaded code. In addition, it permits Web services to build safer, more powerful Web applications. Overall, our goal is to accept the enhanced role of modern browsers in managing the client-side components of complex, non-trusted, distributed applications.

3 Implementation

This section describes the central components of our Tahoma prototype implementation. These components include: the browser operating system, which consists of a BOS kernel, a network proxy, and a window manager; the browser instances; and the underlying Xen virtual machine monitor.

3.1 Xen and the Browser Operating System

The Tahoma browser operating system is a trusted layer that executes multiple browser instances, each within a private, sandboxed execution environment. As shown in Figure 4, our Tahoma implementation uses the Xen virtual machine monitor (VMM) [4]. Xen is an open-source VMM that provides para-virtualized x86 virtual machines. It executes in the most privileged mode on bare hardware, as shown in the figure. Above the VMM are individual virtual machines, which Xen names $Domain0$ through $DomainN$. Each domain executes a guest operating system, such as Linux, which runs at a lower privilege level than the VMM. User-mode applications run on the guest OS at the lowest privilege level.

Xen's $Domain0$ is special; it performs many of the management functions for Xen, but outside of the VMM. $Domain0$ has access to all physical memory. It can create and destroy other domains, make policy decisions for scheduling and memory allocation, and provide access to network devices. In this way, many VMM supervisor functions can be programmed as user-mode code on top of the $Domain0$ guest operating system.

We implemented the Tahoma browser operating system as a collection of processes that execute on Linux in the privileged Xen $Domain0$ VM, as shown in the upper left of Figure 4. The three main BOS processes are: the BOS kernel, which manages browser instances and the durable storage of the system; the network proxy, a reverse firewall that enforces network access policies for Web applications; and the window manager, which aggregates browser instance windows into the physical screen.

Each Tahoma browser instance executes in its own Xen virtual machine. In Figure 4, the first two browser instances are running versions of the Konqueror Web browser, which we ported to Tahoma, on top of a Linux guest OS. The third browser instance is executing a custom multimedia browser on a NetBSD guest OS. The browser instances interface to the BOS through libraries linked into the browser that provide access to BOS system functions (libBOS) and graphics functions (libQT). We describe libBOS below, libQT in Section 3.4, and the browsers we implemented in Section 3.5.

Browser instances must be able to communicate with the $Domain0$ BOS processes, and vice versa. We use the term *browser-calls* to refer to instance-to-BOS communications. In the other direction, BOS-to-instance notifications are delivered as *upcalls* to the instances. Browser-calls are implemented as XML-formatted remote procedure calls, carried over a TCP connection. A point-to-point virtual network link carries the RPCs between each instance and a $Domain0$ BOS process. Because Xen restricts access to point-to-point links to the two VMs involved, the BOS can safely assume messages arriving on these links are authentic; this resembles the way a traditional kernel uses the current process ID to determine the origin of a system call. The library libBOS contains RPC stubs that expose browser-calls to applications as high-level function calls.

Most communication in Tahoma occurs within a Web application, between a browser instance and its Web service. However, the BOS provides three inter-application communication paths – fork, BinStore, and BinFetch – that are implemented through browser-calls. Applications may *fork* other applications. The fork browser-call includes the target URL to be forked as an argument. Based on this URL, the BOS kernel examines (or downloads) the appropriate manifest and determines which browser instance should handle the request. It then launches the browser instance, if needed, and delivers the URL to the instance through an upcall.

The BOS supports strong VM-based isolation between browser instances. However, it must also permit the controlled transfer of objects outside of a VM. For example, a user must be able to copy a photo from a Web-mail application into a photo album, or vice versa. For this purpose, the BOS kernel implements a private "holding bin" for each browser instance. To manipulate the holding bin, the kernel provides two browser-calls – BinStore and BinFetch. A browser instance copies an object to its holding bin by invoking the BinStore browser-call, specifying the object's URL, an object name, and a MIME type. Similarly, the BinFetch browser-call lets the browser instance find and retrieve an object from the holding bin. However, a transfer between the holding bin and the host OS must be initiated *explicitly* by a user through a trusted Tahoma tool; it cannot be initiated by the browser instance. In this way, we permit controlled transfers but prohibit code in the browser instance from directly manipulating host OS resources.

3.2 Xen and the Browser Instance

Browser instances execute in Xen VMs, with Xen handling the low-level details of CPU and memory isolation. The BOS augments Xen by enforcing the manifest-specified network policy of each browser instance. Browser instances are therefore not provided with an unfettered Internet link. Instead, a Xen (virtual) point-to-point network link is established between the browser instance and the $Domain0$ VM. In $Domain0$, we run an HTTP proxy process, derived from tinyproxy [16]. The proxy checks each requested URL against the browser instance's network policy, returning an error if the URL is outside of the manifest-defined Web service.

For unencrypted connections, the proxy can filter based on the full URL. SSL connections, however, encrypt the URL. For these connections, the proxy can filter based only on host and port number. Similarly, other protocols, such as streaming video, can be restricted based only on network- and transport-level connection attributes, since our proxy does not understand their protocols. This limits the trustworthiness of our current proxy to that of the DNS system on which it relies, even for SSL-protected connections.

Each Xen VM executing a browser instance includes several virtual disks, which are initialized and controlled by the BOS kernel. A read-only *root disk* contains the base file system for the browser instance, including the image of its guest OS. A writable *data disk* provides storage for any data the browser instance needs to durably store on the local system. When an application is launched for the first time, its data disk is initialized to a blank file system.

Separating the writable data disk from the read-only root disk permits simple upgrade semantics if the root disk changes: the BOS replaces the root disk, but preserves the data disk. Any data that the browser instance stores in the data disk therefore survives upgrades. More importantly, by making the root disk read-only, we can safely share root disks across browser instances.

Persistent changes made by the application are applied to the virtual data disk on the *guest OS*, not to the file-system of the host OS on which the user is running. In this way, the user's OS is isolated from potentially dangerous changes, such as those made by spyware or other pathogens. Equally

```
<?xml version="1.0"?>
<Manifest Name="http://www.aa.com/#gen"> <Application Name="http://www.aa.com/" GUID=”230884839021434298”>
<NetworkPolicy>
  <Service> <Host>aa.com</Host> </Service>              <Service> <Host>www.aa.com</Host> </Service>
  <Service> <Host>www.touraa.com</Host> </Service>      <Service> <Host>network.realmedia.com</Host> </Service>
  <Service> <Host>www.macromedia.com</Host> </Service>  <Service> <Host>www.latinmedios.com</Host> </Service>
  <Service> <Host>ad.doubleclick.net</Host> </Service>  <Service> <Host>switch.atdmt.com</Host> </Service>

  ...additional advertising partner sites...

</NetworkPolicy>

<BrowserPolicy>
  <Browser>
    <OS>Windows</OS> <arch>ia32</arch> <app>FireFox</app> <url>http://www.mozilla.org/firefox_ia32.vm</url>
  </Browser>

  ...additional browser instances...

</BrowserPolicy>

<Signature xmlns="http://www.w3.org/2000/09/xmldsig#">

   ...DSA signature block...

</Signature></Manifest>
```

Figure 5. A sample manifest. This manifest for the American Airlines Web service permits access to the main Web site and the sites of advertising partners. It indicates that the browser instance should run a stock Firefox browser.

important, the user can remove all durable changes made by an application simply by deleting its browser instance.

3.3 Manifests

A Web service defines a Web application by creating a manifest and making it accessible to the BOS kernel. The manifest describes policies and attributes that control the execution and behavior of all browser instances associated with the Web application. A manifest is an XML document that includes: (1) a *network policy*, which specifies the remote network services with which a browser instance can interact; (2) a *browser policy*, which specifies the code that should be initially installed within a browser instance's sandbox; (3) a digital signature for the Web service; (4) a human-readable Web application name; (5) a machine-readable manifest name; and (6) a globally unique identifier for the application. Figure 5 shows a sample manifest.

3.3.1 Manifest location and authentication

Conceptually, every Web object has an associated manifest. Web servers can supply an "`X-Tahoma-manifest`" HTTP header extension when delivering a Web object. This header specifies the unique name for the object's manifest and a URL from which the manifest can be retrieved by the browser operating system.

There are two ways for the user to launch a Web application for the first time. First, the user can invoke one of Tahoma's management tools and pass the URL to it. Second, the user can type the URL into a browser instance of a *different* Web application. In either case, the result is the same. The BOS performs an HTTP HEAD operation on the supplied URL to find and retrieve the Web application's manifest. It will then `fork` a new browser instance that executes inside a new Xen virtual machine.

In addition to the HTTP header extension, we also support per-server manifest files. If a manifest is not provided in an object's HTTP header, the BOS attempts to download "`/manifest.xml`" from the Web server providing that object. As a final fall-back, the BOS also consults a local database of manually supplied manifest files. If none of these mechanisms succeeds, the BOS automatically generates a new generic manifest that implements the basic Web security model that conventional browsers enforce. The generic manifest permits access to any URLs reached on a path from the top-level URL. However, the BOS forks a new browser instance to execute any document not in the top-level URL's domain.

Tahoma uses public-key certificates to authenticate Web applications to clients. Each Web application has an associated master public/private key pair. Using the private key, Web services sign manifests to prove their authenticity to clients. These signatures are included in the manifests using the XML-SIG standard [5]. Note that this certification scheme does not completely solve all trust issues. Neither the BOS nor a user has any reason to initially believe that a particular key pair speaks for the real-world entity that should be associated with the Web application. For this we rely on traditional PKI certification authorities.

Manifest signatures allow an application's manifest to evolve over time. A signature securely verifies that two manifests came from the same source. A Web service can replace an existing application manifest by sending a new manifest with the same name. Alternatively, the Web service can add a manifest for a Web application by sending a new manifest that has a *different* manifest name but the same application name.

3.3.2 Manifest policies

As previously mentioned, the manifest network policy describes the access rights of a browser instance by listing the Web sites with which it may communicate. An entry in the list contains a host name (or a regular expression against which candidate host names are matched) and an optional set of qualifiers, including a port, protocol, or URL on that host. The Web service specified by a manifest is defined as the union of all items in its network policy. To allow an application's Web service to be incrementally defined, a single Web application may consist of multiple manifests. Because network policies are simply lists, we can easily concatenate network policies within manifests without creating unexpected policy effects.

Web services can express any policy they choose. Nothing prevents one Web application from including a Web object in its manifest that also falls in a second application. In other words, Web applications can overlap. This reflects the nature of the Web: there are no technical restrictions on linking or embedding content without the approval of its publisher. However, a Web application can prevent its browser instances from fetching content from or sending information to other Web applications.

In addition to the network policy, each manifest specifies the code that should be run in the browser instance. If the Web service wishes to run a stock Web browser, it provides a list of permissible browsers and guest OSs. As a performance optimization, the BOS kernel stores a set of VM checkpoints of freshly booted stock browsers. If one of the browser checkpoints matches a permissible browser specified in the manifest, the BOS clones the checkpoint into a new VM sandbox. If not, the BOS relies on the Web service to supply a URL of a VM image to download and execute.

Alternatively, the Web service can mandate that a custom browser instance should run in the sandbox. In this case, the Web service must supply a URL and hash of the custom VM image to be downloaded. The VM image must contain a bootable x86 guest operating system and applications.

We will not discuss network policy creation in detail in this paper. However, we have built a Web crawler to aid in manifest construction and have written manifests for the top ten most popular Web sites (according to Alexa.com). In general, we found it fairly simple to construct manifests with the help of this tool.

3.4 The Window Manager

Tahoma's user interface is implemented by a window manager process running in $Domain0$. We designed the windowing mechanisms with both performance and safety in mind. For performance, the window manager offloads work, using functions available in the graphics processing unit (GPU) of modern video cards.For safety, our implementation ensures that browser instances cannot perform denial-of-service attacks by consuming excessive BOS resources: all graphics state is maintained by and charged to the browser instances themselves.

The Tahoma window manager provides a virtual screen abstraction to each browser instance. Within this virtual screen, the browser instance can create and position one or more rectangular *sprites,* as shown in Figure 6a. Each sprite consists of a grid of *tiles*. A tile, which is backed by a single 4KB machine page in the browser instance's virtual machine, contains 32 x 32 pixels, with 32 bits of color and alpha information per pixel.

Providing browser instances with the abstraction of multiple sprites is useful for several reasons. A Web browser typically exposes multiple windows to the user; each window can be represented by a sprite. In addition, layered user-interface elements, such as floating toolbars and pull-down menus, can be incorporated as additional sprites overlaid on the main window sprite. By using page-aligned tiles, the window manager can exploit the dirty-page tracking of the CPU memory management unit (MMU) to determine which tiles have been modified and need to be copied to the graphics card.

The Tahoma window manager superimposes the sprites of each browser instance onto the physical computer screen, as shown in Figure 6. Many different policies are possible. For example, the window manager could co-mingle the sprites of all browser instances in the main screen area, as shown in Figure 6b. Alternatively, it could preserve the notion of virtual screens, as shown in Figure 6c.

To simplify porting browsers to Tahoma, we modified the Qt multi-platform GUI library to interact with the window manager through its tiles and sprites abstractions. We preserved Qt's API; Qt-compatible applications can be relinked against our modified libQT to make them work with Tahoma's graphics subsystem.

3.5 Browsers

The execution environment of a browser instance is based on a Xen virtual machine. Therefore, most applications will run on Tahoma with little or no modification. However, three kinds of modifications may be necessary: (1) linking to libQT to access the Tahoma graphics subsystem; (2) using a browser-call to access remote services, rather than accessing the network directly through a virtual device; and (3) using browser-calls for new functions, such as forking a new browser instance or interacting with the holding bin.

To date, we have implemented two Tahoma browsers: a port of the Konqueror Web browser [18] and a port of the MPlayer media player [20]. Konqueror is a fully fea-

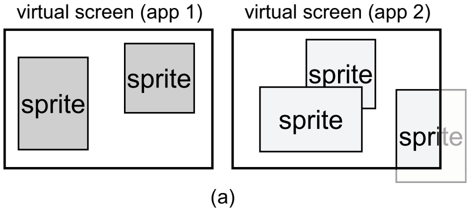

Figure 6. Tahoma window manager. (a) The Tahoma window manager aggregates the virtual screens of each browser instance on the physical screen. (b) In one policy, the individual sprites from each instance are all collapsed into a single drawing area. (c) In another policy, the window manager preserves the isolation of sprites for each instance in their own virtual screens.

tured Web browser that supports Java applets, JavaScript, SSL, DOM, and cascading style sheets. The MPlayer media player supports most popular video and audio codecs, either natively or by wrapping third party binary codec libraries. The MPlayer implementation is performance focused, optimizing the speed of video decoding via specialized hardware features, such as CPU SIMD extensions or GPU colorspace conversion and image scaling.

Our MPlayer port demonstrates the flexibility of the Tahoma architecture. By selecting MPlayer to run in their browser instances, Web services can provide users with streaming media applications, such as Internet radio or television, instead of a more conventional HTML browsing service. From the perspective of a user, an MPlayer browser instance will have a radically different look and feel than a conventional Web browser. From the perspective of Tahoma, MPlayer is simply a browser instance and is treated like any other browser instance.

3.6 Summary

We implemented the Tahoma browser operating system as a layer on top of the Xen virtual machine monitor. Our implementation benefits from the ability to create and control Xen virtual machines through the Xen $Domain0$ VM. This lets us program the major components of Tahoma – the BOS kernel, the network proxy, and the window manager – as user-mode $Domain0$ processes. Browser instances, which are encapsulated in Xen virtual machines, communicate with the BOS kernel through high-level browser-calls carried by virtual network links. In the following section, we evaluate the safety benefits and the performance overhead of our Tahoma architecture and its implementation.

4 Evaluation

Our Tahoma prototype consists of approximately 10K lines of Perl and C code. This section evaluates two aspects of the prototype: (1) its safety and effectiveness in containing threats, and (2) its performance. We have not yet optimized our implementation. Therefore, our performance results should be considered as an upper bound on the overhead intrinsic to our approach.

4.1 Safety and Effectiveness

A critical measure of Tahoma's value is whether it successfully prevents or contains threats that occur in practice. Isolation should provide significant safety benefits. However, Tahoma will not prevent all threats.

As an example, security vulnerabilities can arise due to Tahoma's dependence on external systems, such as DNS. Attackers that subvert DNS can subvert Tahoma's network filtering policies by changing legitimate bindings to point to IP addresses outside of the intended domain. Tahoma cannot defend itself from these attacks. Another example is a malicious browser instance, which could use a sharing interface provided by Tahoma to attack another browser instance or Web application. While Tahoma greatly reduces the number of sharing channels, these channels still exist. Consider a browser that contains a buffer-overflow vulnerability in its URL parsing code. A malicious browser instance could use the `fork` browser-call to pass an attack string to a second browser instance, potentially subverting it. Any channel that permits sharing between mutually distrusting Web applications is susceptible to attack.

To quantitatively evaluate the effectiveness of Tahoma, we obtained a list of 109 security vulnerabilities discovered in current or previous versions of the widely used Mozilla open source browser. We analyzed the vulnerabilities and classified them into five different categories. The five vulnerability categories, along with the Tahoma features intended to defend against them, are:

- **Sandbox weakness:** Browsers use language and runtime mechanisms to sandbox scripts, applets, and other active Web content, but these language- and type-specific sandboxes are often flawed. In con-

class of vulnerability	examples	% contained
weak sandbox	Active content can replace a portion of the JavaScript runtime with its own scripts and gain access to trusted areas of Mozilla.	100% (55 of 55)
vulnerable sharing interface	By crafting an HTML upload form, attackers can select the name of a file to transfer, accessing any file on the user's machine.	86% (25 of 29)
improper labeling	By subverting DNS, an attacker can trick a browser into sending cached credentials to an IP address of the attacker's choosing.	33% (4 of 12)
interface spoofing	Web content can override Mozilla's user interface, allowing attackers to spoof interface elements or remote sites.	100% (11 of 11)
other	Though instructed by the user not to do so, Mozilla stores a password on disk.	0% (0 of 2)
total:		87% (95 of 109)

Table 2. Vulnerabilities. We show the percentage of Mozilla vulnerabilities that Tahoma contains or eliminates.

trast, Tahoma uses virtual machines as a language-independent sandbox for the entire browser instance.

- **Vulnerable sharing interface:** Browsers contain many programmatic interfaces (e.g., DOM access) and user interfaces (e.g., file upload dialog boxes) for sharing data across security domains. These interfaces can often be subverted. In Tahoma, we limit sharing across Web applications to a small set of browser-calls and holding bin manipulation interfaces.
- **Improper labeling:** Browsers assign Web objects to security domains using a complicated set of heuristics. Incorrectly labeling an object as belonging to a domain can enable attacks such as drive-by downloads. In Tahoma, Web services explicitly declare the scope of their Web application through manifests.
- **Interface spoofing:** Browsers are susceptible to spoofing attacks, in which a malicious site attempts to occlude or replicate browser UI elements or the "look and feel" of victim sites. In Tahoma, the Tahoma window manager decorates browser instances with labeled borders that cannot be accessed or occluded.
- **Other:** Some vulnerabilities could not easily be classified; this category is a "catch-all" for these.

We examined each of the 109 Mozilla vulnerabilities to determine whether Tahoma successfully contains or eliminates the threat within the affected browser instance, or whether the attacker can use the vulnerability to harm external resources or Web applications.

Table 2 shows the results of our analysis, broken down by our vulnerability categories. As examples, we list one specific attack that was seen for each category. The table shows that Tahoma successfully contains or eliminates 95 of the 109 listed Mozilla vulnerabilities (87%). Many of these vulnerabilities are browser implementation flaws that allow a remote attacker to inject code, extract files from the user's machine, or otherwise subvert the browser's security mechanisms. Although Tahoma does not directly fix these vulnerabilities, its isolated virtual machines *contain* the damage to a single browser instance and its application, preserving the integrity of the user's resources, the host operating system, and other browser instances.

A good example of a contained vulnerability is an attack on Mozilla's SSL certificate management functions. An attacker could deliver a malicious email certificate to Mozilla that masks a built-in certificate-authority certificate, permanently blocking SSL connections to valid sites. Under Tahoma, this attack would succeed on a susceptible browser instance, but it would be contained to that instance.

4.2 Performance

Our analysis of Mozilla vulnerabilities demonstrates that Tahoma can increase safety and security for Web browsing. However, there is typically a tradeoff between safety and performance. Given Tahoma's use of VMs for isolation, what is the cost of virtualization to the user and to the Web application?

To answer this question, we ran several benchmarks to quantify the performance of common Web-browsing operations and the overhead of Tahoma's browser virtualization. Our measurements were made on an Intel Pentium 4 processor with a 3.0GHz clock, 800MHz front-side bus, 1 GB of RAM, and an ATI Radeon 9600SE graphics card. Networking tests used an on-board Intel Pro/1000 NIC connected to an Asante FriendlyNet GX5-2400 Gigabit Ethernet switch. We booted Linux version 2.6.10 either directly on the CPU or in Xen virtual machines, as indicated for each experiment. For the Xen-hosted tests, the kernels included the necessary Xen modifications, built from the Xen 2.0 unstable branch with patches through March 7, 2005.

4.2.1 The cost of virtual machine creation

Although the optimization of virtual machine performance is well studied [4], virtualization still has a cost. In particular, the Tahoma implementation frequently creates (or forks) a virtual machine to execute a new browser instance. Forks occur whenever the user enters the URL of a new Web application. Therefore, we wished to measure the impact of VM fork overhead on Tahoma users.

operation		average latency
Tahoma fork( )	specialize a pre-forked browser instance	1.06 seconds
	clone a new VM, boot guest OS, launch browser program	9.26 seconds
native Konqueror open URL	load URL in running Konqueror	0.84 seconds
	warm-start Konqueror	1.32 seconds
	cold-start Konqueror	5.74 seconds

Figure 7. Browser fork latency. This table compares the time to fork a new browser instance for the Konqueror browser running on the Tahoma prototype, and for Konqueror running on native Linux.

Figure 7 shows the cost of forking a new Tahoma browser instance in a virtual machine compared to the cost of starting a new browser in native Linux. The top half of the table shows two different Tahoma cases. The first line shows the time to "specialize" a pre-forked browser instance. Because we expect forking of commodity browsers to be the common case, Tahoma maintains a pool of pre-forked guest operating systems with stock browsers (we use Konqueror on Linux for this test). When the BOS receives a fork browser-call, it checks whether a pre-forked version of the specified browser and guest OS is available. If so, then the BOS need only set up the appropriate network policy in a tinyproxy process and "specialize" the browser instance by mounting its data disk. The time to instantiate and specialize a pre-forked browser instance is 1.06 seconds.

If a compatible pre-forked instance cannot be found in the pool, then to service the fork, Tahoma must clone a new VM, boot its guest OS, and launch the browser. The cost for this full operation is 9.26 seconds.

For comparison, the bottom half of Figure 7 shows the latency of opening a Konqueror window on native Linux. We measured three cases: (1) the latency of opening a new window in a running Konqueror process, (2) the "warm-start" latency of launching Konqueror, assuming it has been previously launched, and (3) the "cold-start" latency of launching Konqueror on a cold file system buffer cache. Interestingly, the best case latency with Konqueror on native Linux, 0.84 seconds for an already executing browser, is only slightly (and imperceptibly) better than the time to launch a pre-forked Tahoma VM, while a warm-start of Konqueror is slightly worse than the pre-fork operation. The latency for a Konqueror cold start on native Linux is 5.7 seconds, 60% of the latency of a full VM clone and OS boot on Tahoma. Both the cold-start and full-clone latencies are relatively long; both could be reduced through optimization.

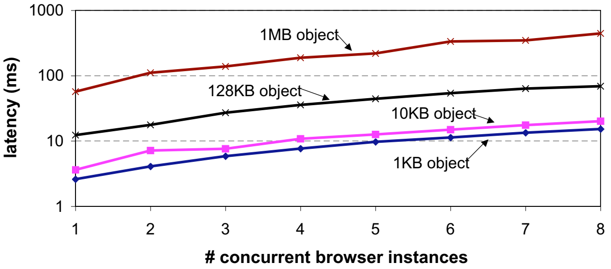

Figure 8. Web object download latency (LAN). this graph shows the latency of downloading Web objects of different size, with varying numbers of concurrent browser instances actively fetching the object, over a LAN.

4.2.2 Network performance

From the user's perspective, there are two key network-related performance metrics: the latency of fetching a Web page, and the achieved throughput when fetching a large object. On Tahoma, all browser network communications flow through the Xen $Domain0$ VM, where they are filtered by tinyproxy according to the network policy. We measured the latency and bandwidth overhead of this additional indirection and filtering.

To measure the Web-object fetch latency, we started several concurrent browser instances, each scripted to fetch a Web object repeatedly. We measured the average latency to fully retrieve the object from a dedicated server on the local network as a function of the number of concurrent browser instances for different object sizes. Figure 8 shows the results. For a single browser instance fetching a 10KB Web object, the measured fetch latency was 3.6 ms. With eight concurrent browser instances, the latency for the 10KB object grew to 20.1 ms. These results are encouraging, as this latency is well below the 625 ms response-time threshold for users to operate in an optimal "automatic response" mode [30]. For large Web pages (1MB), the single-instance latency was 57 ms; at eight concurrent instances, the latency grew to 444 ms.

As the Web page size grows, the user's perceived response time is dictated by the system's bottleneck bandwidth. We repeated our latency measurements on a wide-area network. In all cases, network round-trip time and bandwidth dominated the download latency: Tahoma was not a bottleneck and had no impact on perceived latency in a WAN setting.

We compared the throughput of a long-lived TCP connection under Tahoma and native Linux. For this experiment, we initiated a TCP connection from the client to a server running on the local LAN and measured the sustained throughput from the server. Table 3 shows our results. Surprisingly, Tahoma's raw throughput exceeds that

1081-6011/06 $20.00 © 2006 IEEE

scenario		TCP throughput
Tahoma	direct from domain 0	911 Mb/s
	domain 1, routed through domain 0	638 Mb/s
	domain 1, proxied through domain 0	637 Mb/s
native Linux	direct	840 Mb/s
	through a local proxy	556 Mb/s

Table 3. Long-lived TCP throughput. This table compares the TCP bandwidth achieved in Tahoma and on native Linux.

display system	unit of execution	# sustained MPlayers
Tahoma graphics	VM	12
(networked) X11	VM	1
(shared-memory) X11	process	20

Table 4. Graphics throughput. This table compares the maximum number of sustainable MPlayer instances under different scenarios.

of native Linux: $Domain0$ achieves 911 Mb/s bandwidth from the server, compared to 840 Mb/s on native Linux. High-bandwidth TCP connections are notoriously sensitive to small parameter changes (such as send and receive buffer sizes) or timing discrepancies. Accordingly, it is difficult to fully account for the performance differences, though we attribute some of them to Xen performance optimizations, such as interrupt batching.

From Table 3, we can isolate the costs of indirection and proxying in Tahoma. Routing communications through $Domain0$ from another VM reduces throughput by 30%, to 638 Mb/s. The additional cost of the tinyproxy filtering is almost negligible at that point. From these numbers, we conclude that Tahoma's throughput, even when filtered through a proxy in the $Domain0$ VM, is high enough to support the vast majority of Web browsing workloads.

4.2.3 User interface performance

To measure the performance of the Tahoma window manager, we ran a variable number of virtual machines, each containing an MPlayer browser instance, which we consider a "worst case" test. Each MPlayer application rendered a 512x304 pixel DIVX/AVI video at 25 frames per second. We increased the number of browser instances until MPlayer reported that it could no longer sustain this frame rate. We ran this benchmark under two Tahoma configurations: (1) each MPlayer running as a browser instance under Tahoma using Tahoma's window manager, and (2) each MPlayer running as a browser instance under Tahoma, but using X11 to render to a $Domain0$ Xserver. We also ran an experiment on native Linux, where each MPlayer ran as a Linux process using shared-memory X11 to render to the local Xserver.

Table 4 shows our results. Tahoma's window manager can sustain 12 MPlayer instances simultaneously, achieving an order of magnitude better performance than X11 across VMs. Native Linux with shared-memory X11 improves on Tahoma by 70% (20 sustained instances), but it lacks Tahoma's isolation benefits. Tahoma's ability to support 12 simultaneous video players indicates that multiplexing windows from multiple VMs should not pose a visible performance problem for Tahoma users.

To measure Tahoma's input performance, we recorded the delay between the time a user presses a key and the time the corresponding character is rendered by a Konqueror browser instance. To do this, we instrumented Xen to timestamp physical keyboard interrupts and instrumented Konqueror to timestamp character rendering events. In the simple case of a single Konqueror browser instance, the input echo time was under 1 ms. In an attempt to increase window management interference, we measured the same input event with 10 MPlayer browser instances running 10 video streams concurrently. When competing with the ten MPlayer instances, the Konqueror echo time remained below 12 ms, still imperceptibly small to the user.

4.3 Summary

This section examined the safety, effectiveness, and performance of the Tahoma implementation. We used a list of 109 security vulnerabilities in the Mozilla browser to evaluate Tahoma's effectiveness at containing threats. Our analysis shows that Tahoma can contain or eliminate 87% of the vulnerabilities discovered in Mozilla. Next, we ran benchmarks to quantify the performance cost of Tahoma's VM-based isolation mechanism. Our benchmarks demonstrate that despite virtualization, indirection, and proxying, Tahoma can achieve the latency, throughput, and responsiveness characteristics demanded by the vast majority of browsing applications.

5 Related Work

Tahoma is a composite of architectural elements that isolate Web applications and provide users with a safer experience. Several of Tahoma's architectural components have been explored in various forms in the past, as has the general topic of improving Web security. We discuss this related work below.

5.1 Web Security Vulnerabilities

Vulnerabilities can exist in both client-side browsers and in the Web services with which they communicate. In browsers, scripting languages, such as JavaScript and VBScript, are a major source of security flaws. While individual flaws can be addressed, the underlying security framework of browser scripting is itself considered unsafe [2], suggesting that flaws arising from active content will be an ongoing problem.

Java applet security is a well-studied topic [8]. Java's current stack-based security model [33] is significantly stronger its original model. However, Java applets have recently taken a secondary role to on the Web other forms of active content, such as Flash elements, ActiveX components, and JavaScript. Tahoma uses VMs to provide a language-independent safe execution environment for browser instances. Even if a browser has security vulnerabilities, Tahoma contains those flaws within the VM sandbox.

Web services are prone to attack from buffer overruns, SQL injection attacks, and faulty access control policies. Improving Web service security is an active research topic [25, 13] beyond the scope of this paper.

5.2 Sandboxes

Multiple approaches for containing code within sandboxes [21] have been explored, including OS system call interposition [6, 11], fine-grained capability-based systems [27], intra-process domains [7], and virtual machine monitors [4, 34] or hypervisors [19]. In addition to exploring mechanisms, researchers have explored appropriate policies and usage models. MAPbox [1] defines a set of canonical application class labels (such as compiler, network client, or server) and appropriate sandboxes for them and relies on the user to classify programs according to those labels. WindowBox [3] provides users with durable, isolated Windows desktops, each associated with different roles or security levels (e.g., work, home, or play). Tahoma uses the Xen VMM to implement virtual machine sandboxes, each containing one Web browser instance. No sharing is permitted between browser instances except through Tahoma's narrow browser system call API.

GreenBorder [12] provides a sandboxed environment within Windows for the Internet Explorer and Outlook applications. GreenBorder works by virtualizing access to Windows resources such as the file system or the registry, and redirecting modifications of these resources to virtualized copies. By permitting users to "flush" these changes, any harmful side-effects from malicious content can be cleaned. GreenBorder visually separate applications running within the sandbox from trusted applications, and provides some degree of auditing and reporting. Tahoma also provides a sandboxed environment for Web browsers, but Tahoma additionally isolates Web *applications* from each other. As well, Tahoma permits Web services to customize the browsers used to access them, and to control which remote sites their browser instances can access.

5.3 Safely Executing Downloaded Applications

Tahoma shares the popular vision of making executable content available on the Internet for users to download and run safely. Jaeger et al. [15] describe a distributed system for authenticating and executing content from remote principals. They provide a rich policy structure for assigning access rights to local resources; in contrast, Tahoma uses the shared-nothing abstraction of VMs to isolate downloaded browser instances from each other and from the host OS. Web browsers support the safe execution of Java applets [8]. Applets have similarities to Tahoma browser instances, though browser instances can be written in any language, as their execution environment is a hardware VM.

The Collective project [22] encapsulates collections of applications within VMware [29] virtual machines and ships these compute "appliances" over the network to users. Tahoma is similar in that browser instances are encapsulated within VMs and downloaded to users. However, Tahoma's browser operating system mediates the access of browser instances to local host resources and remote Web services.

SubOS assigns "sub-user IDs" to downloaded objects. Processes that access these objects inherit the restricted access rights of the associated sub-user IDs [14]. Using this mechanism, this project re-factored a Web browser so that isolation is delegated to SubOS rather than the browser itself. This work has similar principles to ours, though Tahoma uses stronger VM sandboxes, isolates browsers and Web objects based on the Web applications to which they belong, and provides a trusted window manager and other user tools.

5.4 Trust and User Interfaces

Untrusted executable content can attack users by spoofing the user interface of trusted executables, fooling the user into providing sensitive information. Researchers have explored the potential for Web content to act as a trojan horse, spoofing local OS interfaces or remote services [31, 9]. Modern "phishing" attacks are an increasingly prevalent form of this attack.

The compartmented mode workstation (CMW) specification [35] provides requirements for enforcing mandatory access control in a multi-level or compartmented security system, and describes how to label data and windows with sensitivity labels. The trusted paths work by Ye et al. [36] attempts to label Web browser windows in an unforgeable way by clearly separating graphical content provided by remote servers from status information provided by the local browser. To prevent remote servers from spoofing status information, the trusted paths architecture uses synchronized random dynamic boundaries, in which the color of window borders is randomly shifted but synchronized with a reference window. The EROS [28] trusted window system (EWS) solves the complementary problem of denying untrusted applications the authority to disrupt trusted UI paths.

Tahoma uses many of the mechanisms and window labeling techniques from this body of work, but adapts them to a virtual machine environment. Like EWS, Tahoma browser instances use shared memory to convey pixel information to the trusted BOS window manager. Tahoma's windowing environment was also partially inspired by the desktop operating environment winder server (DoPE) [10].

6 Conclusions

Over the last decade, the Web has evolved from a repository of interconnected, static content to a delivery system for complex, distributed applications and active content. As a result, modern browsers now serve as de facto operating systems that must manage dynamic and potentially malicious applications. Unfortunately, browsers have not adapted to their new role, leaving the user vulnerable to many potential threats.

This paper presented the architecture and implementation of Tahoma, a new Web browsing system intended to improve safety and security for Web users. In the Tahoma architecture, each Web application is isolated within its own virtual machine sandbox, removing the need to trust Web browsers and the services they access. Virtual machine sandboxes contain the damage that can be caused by malicious or vulnerable browsers. Consequently, Tahoma protects other applications, resources, and the user's host OS from these dangers.

We introduced a new trusted software layer in Tahoma, the *browser operating system* (BOS), which manages Web applications and their virtual machine sandboxes. To limit damage from hijacked browsers, the BOS restricts the set of sites with which each Web application can communicate. The BOS gives users increased visibility and control over downloaded Web applications. We implemented Tahoma on the Xen virtual machine monitor. Our security evaluation shows that Tahoma can prevent or contain 87% of the vulnerabilities that have been identified in the widely used Mozilla browser. Our performance evaluation demonstrates that users need not sacrifice performance for the benefits of stronger isolation and safety.

7 Acknowledgments

The authors are grateful for the insightful feedback provided by the anonymous reviewers, and for helpful discussions with Brian Bershad, Ed Lazowska, and Andrew Schwerin. This work was supported in part by the National Science Foundation under grants CNS-0430477, CCR-0326546, and ANI-0132817, by an Alfred P. Sloan Foundation Fellowship, by the Wissner-Slivka Chair, by the Torode Family Endowed Career Development Professorship, and by gifts from Intel Corporation and Nortel Networks.

References

[1] A. Acharya and M. Raje. MAPbox: Using parameterized behavior classes to confine untrusted applications. In *Proceedings of the Ninth USENIX Security Symposium*, Denver, CO, August 2000.

[2] V. Anupam and A. Mayer. Security of web browser scripting languages: Vulnerabilities, attacks, and remedies. In *Proceedings of the Seventh USENIX Security Symposium*, San Antonio, TX, January 1998.

[3] D. Balfanz and D. R. Simon. Windowbox: A simple security model for the connected desktop. In *Proceedings of the Fourth USENIX Windows Systems Symposium*, Seattle, WA, August 2000.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th Symposium on Operating System Principles (SOSP 2003)*, Bolton Landing, NY, October 2003.

[5] M. Bartel, J. Boyer, B. Fox, M. LaMacchia, and E. Simon. XML-signature syntax and processing. W3C recommendation, published at `http://www.w3.org/TR/xmldsig-core/`, February 2002.

[6] A. Berman, V. Bourassa, and E. Selberg. TRON: Process-specific file protection for the UNIX operating system. In *Proceedings of the 1995 Winter USENIX Conference*, New Orleans, LA, January 1995.

[7] C. Cowan, S. Beattie, G. Kroach-Hartman, C. Pu, P. Wagle, and V. Gligor. Subdomain: Parsimonious server security. In *Proceedings of the 14th USENIX Systems Administration Conference (LISA 2000)*, New Orleans, LA, December 2000.

[8] D. Dean, E. W. Felten, D. S. Wallach, and D. Balfanz. Java security: Web browsers and beyond. Chapter 7 of "Internet besieged: Countering cyberspace scofflaws". ACM Press, New York, NY, 1997.

[9] E. W. Felten, D. Balfanz, D. Dean, and D. S. Wallach. Web spoofing: an Internet con game. In *Proceedings of the 20th National Information Systems Security Conference*, Baltimore, MD, October 1996.

[10] N. Feske and H. Härtig. DOpE – a window server for real-time and embedded systems. In *Proceedings of the 24th IEEE International Real-Time Systems Symposium*, Cancun, Mexico, December 2003.

[11] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications. In *Proceedings of the Sixth USENIX Security Symposium*, San Jose, CA, July 1996.

[12] Green Border Technologies. GreenBorder desktop DMZ solutions. http://www.greenborder.com, November 2005.

[13] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D. T. Lee, and S.-Y. Kuo. Securing Web application code by static analysis and runtime protection. In *Proceedings of the Thirteenth International World Wide Web Conference (WWW 2004)*, New York, NY, May 2004.

[14] S. Ioannidis and S. M. Bellovin. Building a secure Web browser. In *Proceedings of the FREENIX track of the 2001 USENIX Annual Technical Conference*, Boston, MA, June 2001.

[15] T. Jaeger, A. D. Rubin, and A. Prakash. Building systems that flexibly control downloaded executable content. In *Proceedings of the Sixth USENIX Security Symposium*, San Jose, CA, July 1996.

[16] R. J. Kaes and S. Young. The tinyproxy lightweight HTTP proxy. http://tinyproxy.sourceforge.net, August 2004.

[17] P.-H. Kamp and R. N. Watson. Jails: Confining the omnipotent root. In *Proceedings of the Second International System Administration and Networking Conference (SANE)*, Maastricht, The Netherlands, May 2000.

[18] Konqueror. http://www.konqueror.org.

[19] T. Mitchem, R. Lu, and R. O'Brien. Using kernel hypervisors to secure applications. In *Proceedings of the 13th Annual Computer Security Applications Conference (ACSAC '97)*, San Diego, CA, December 1997.

[20] MPlayer. http://www.MPlayerHQ.hu.

[21] D. S. Peterson, M. Bishop, and R. Pandey. A flexible containment mechanism for executing untrusted code. In *Proceedings of the Eleventh USENIX Security Symposium*, San Francisco, CA, August 2002.

[22] C. Sapuntzakis and M. S. Lam. Virtual appliances in the Collective: A road to hassle-free computing. In *Proceedings of the Ninth Workshop on Hot Topics in Operating Systems (HotOS IX)*, Lihue, HI, May 2003.

[23] S. Saroiu, S. D. Gribble, and H. M. Levy. Measurement and analysis of spyware in a university environment. In *Proceedings of the First Symposium on Networked Systems Design and Implementation (NSDI '04)*, San Francisco, CA, March 2004.

[24] B. R. Schatz. NCSA Mosaic and the World Wide Web: Global hypermedia protocols for the Internet. *Science*, 265:841–1004, August 1994.

[25] D. Scott and R. Sharp. Abstracting application-level Web security. In *Proceedings of the Eleventh International World Wide Web Conference (WWW 2004)*, Honolulu, HI, May 2002.

[26] Secunia. Microsoft Internet Explorer window injection vulnerability. http://secunia.com/advisories/13251/, December 2004.

[27] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: a fast capability system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP'99)*, Kiawah Island, SC, December 1999.

[28] J. S. Shaprio, J. Vanderburgh, E. Northup, and D. Chizmadia. Design of the EROS trusted window system. In *Proceedings of the Thirteeenth USENIX Security Symposium*, San Diego, CA, August 2004.

[29] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *Proceedings of the 2001 Annual USENIX Technical Conference*, Boston, MA, June 2001.

[30] S. L. Teal and A. I. Rudnicky. A performance model of system delay and user strategy selection. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, Monterey, CA, May 1992.

[31] J. Tygar and A. Whitten. WWW electronic commerce and Java trojan horses. In *The Second USENIX Workshop on Electronic Commerce Proceedings*, Oakland, CA, November 1996.

[32] VMware, Inc. Browser appliance virtual machine. http://www.vmware.com/vmtn/vm/browserapp.html, October 2005.

[33] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. Extensible security architectures for Java. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP '97)*, Saint Malo, France, October 1997.

[34] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, MA, December 2002.

[35] J. P. Woodward. Security requirements for system high and compartmented mode workstations. Technical Report MTR 9992, The MITRE Corporation, 1987.

[36] Z. Ye and S. Smith. Trusted paths for browsers. In *Proceedings of the Eleventh USENIX Security Symposium*, San Francisco, CA, August 2002.

[37] ZDNet UK. Firefox phishing vulnerability discovered. http://news.zdnet.co.uk/internet/security/0,39020375,39183106,00.htm.