Building a Secure Web Browser*

Sotiris Ioannidis
sotiris@dsl.cis.upenn.edu
University of Pennsylvania

Steven M. Bellovin
smb@research.att.com
AT&T Labs Research

Abstract

Over the last several years, popular application such as Microsoft Internet Explorer and Netscape Navigator have become prime targets of attacks. These applications are targeted because their function is to process unauthenticated network data that often carry active content. The processing is done either by helper applications, or by the web browser itself. In both cases the software is often too complex to be bug free. To make matters worse, the underlying operating system can do very little to protect the users against such attacks since the software is running with the user's privileges.

We present the architecture of a secure browser, designed to handle attacks by incoming malicious objects. Our design is based on an operating system that offers process-specific protection mechanisms.

Keywords: Secure systems, web browser, process–specific protection.

1 Introduction

In the current highly interconnected computing environments, Web browsers are probably the most popular tool for receiving data over the internet. More often than not, the data come from unauthenticated sources that can potentially be malicious. Since the incoming data often carry active content that will be interpreted on the client machine, in many cases without the users knowledge, a number of attacks become possible.

To interpret active content web browsers often rely on helper applications, that become security critical since they operate on untrusted data. These applications which are often buggy [11], execute with the users privileges and can therefore compromise the security of the system. Furthermore the browsers also interpret code like JavaScript and VBScript [6], making the browser itself vulnerable [1].

In this paper we present the architecture of a secure web browser. Our system is designed to address the problems that plague the popular Web browsers by using support offered by the operating system. We built our prototype on SubOS [12]. SubOS is an operating system that offers process-specific protection mechanisms, which we will explain in Section 3.

The paper is organized as follows. In Section 2 we discuss the motivation behind this work. In Section 3 we give a brief background description of a SubOS-capable operating system. In Section 4 we present the architecture of our system. In Section 5 we discuss related work, and finally we conclude in Section 6.

2 Motivation

With the growth of the Internet, exchange of information over wide-area networks has become essential for users. Web browsers, like Netscape Navigator and Microsoft Internet explorer often automatically invoke helper application to handle the downloaded object. In some cases, like in Perl scripts, they will query the user about executing it. In others, like in Postscript files or Java applets [10, 15, 9], they will execute the content, possibly compromising the security of the system. The former approach puts a lot of burden on the user, who more often than not is not particularly security conscious. In

*This work was supported by DARPA under Contract F39502-99-1-0512-MOD P0001.

[1] There are a number of hostile JavaScript and VBScript sites on the Web, easily found using search engines

the latter case the user is bypassed altogether and system security becomes dependent on the correctness of the Postscript or Applet viewer.

It is also the case that seemingly inactive objects like Web pages are very much active and potentially dangerous. One example is JavaScript [6]programs which are executed within the security context of the page with which they were down–loaded, and they have restricted access to other resources within the browser. Security flaws exist in certain Web browsers that permit JavaScript programs to monitor a user's browser activities beyond the security context of the page with which the program was down-loaded (CERT Advisory CA:97.20). It is obvious that such behavior automatically compromises the user's privacy and security.

The lack of flexibility in modern operating systems is one of the main reasons security is compromised. The UNIX operating system, in particular, violates the principle of least privilege. The principle of least privilege states that a process should have access to the smallest number of objects necessary to accomplish a given task. UNIX only supports two privilege levels: "root" and "any user".

To overcome this shortcoming, UNIX, can grant temporary privileges, namely `setuid(2)` (set user id) and `setgid(2)` (set group id). These commands allow a program's user to gain the access rights of the program's owner. However, special care must be taken any time these primitives are used, and as experience has shown a lack of sufficient caution is often exploited [13].

Another technique used by UNIX is to change the apparent root of the file system using `chroot(2)`. This causes the root of a file system hierarchy visible to a process to be replaced by a subdirectory. One such application is the `ftpd(8)` daemon; it has full rights in a safe subdirectory, but it cannot access anything beyond that. This approach, however, is very limiting, and in the particular example commands such as `ls(1)` become unreachable and have to be replicated.

These mechanisms are inadequate to handle the complex security needs of todays applications. This forces a lot of access control and validity decisions to user–level software that runs with the full privileges of the invoking user. To overcome these shortcomings applications such as Web browsers become responsible for accepting requests, granting permissions and managing resources. All this is what is traditionally done by operating systems. Web browsers consequently, because of their complexity as well as the lack of flexibility in the underlying security mechanisms, possess a number of security holes. Examples of such problems are numerous, e.g. JavaScript, malicious Postscript documents, etc.

We wish to demonstrate how to build a secure browser, designed to handle attacks by incoming malicious objects, on top of an an operating system that offers process-specific protection mechanisms.

3 SubOS-enabled Operating Systems

SubOS is a process–specific protection mechanism, a more extensive discussion on SubOS can be found in [12]. Under SubOS any application (e.g. ghostscript, perl, etc.) that might operate on possibly malicious objects (e.g. postscript files, perl scripts, etc.) behaves like an operating system, restricting their accesses to system resources. We are going to call these applications SubOS processes, or sub-processes in the rest of this paper. Figures 1 and 2 demonstrate the difference between a regular and a SubOS-enabled operating system. The access rights for that object are determined by a sub-user id that is assigned to it when it is first accepted by the system. The sub-user id is a similar notion to the regular UNIX user id's. In UNIX the user id determines what resources the *user* is allowed to have access to, in SubOS the sub-user id determines what resources the *object* is allowed to have access to. The advantage of using sub-user id's is that we can identify individual objects with an immutable tag, which allows us to bind a set of access rights to them. This allows for finer grain per-object access control, as opposed to per-user access control.

The idea becomes clear if we look at the example shown in Figure 3. Let us assume that our untrusted object is a postscript file `foo.ps`. To that object we have associated a sub–user id, as we will discuss in Section 3.1. Foo.ps initially is an inactive object in the file system. While it remains inactive it poses no threat to the security of the system. However the moment `gs(1)` opens it, and starts executing its code, foo.ps becomes active, and automatically a possible danger to the system. To contain this threat, the applications that open untrusted ob-

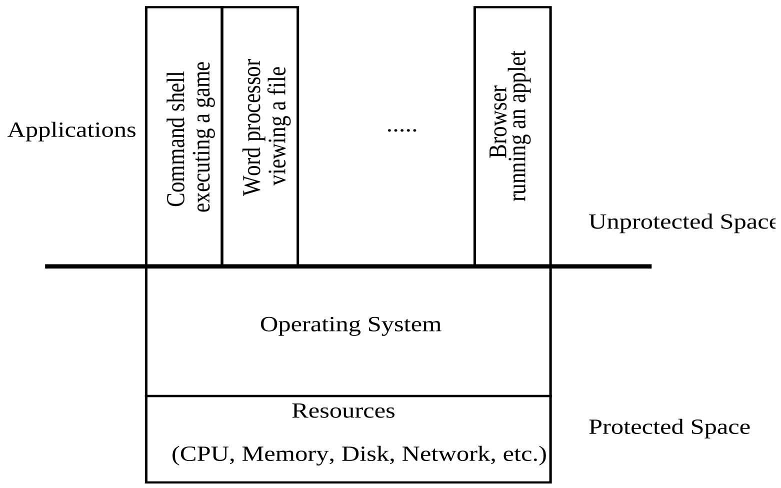

Figure 1: **User applications executing on an operating system maintain the user privileges, allowing them almost full access to the underlying operating system.**

jects, inherit the sub–user id of that objects, and are hereafter bound to the permissions and privileges dictated by that sub–user id.

There is a strong analogy here to the standard UNIX setuid mechanism. When a suitably-marked file is executed, the process acquires the access rights of the owner. With SubOS, suitably-marked *processes* acquire the access rights of the owner of the *files* that they open. In this case, of course, the new rights are never greater than those the process had before.

The advantages of our approach become apparent if we consider the alternative methods of ensuring that a malicious object does not harm the system. Again using our postscript example we can execute foo.ps inside a safe interpreter that will limit its access to the underlying file system. There are however a number of examples on how relying on safe languages fails [11]. We could execute the postscript interpreter inside a sandbox using `chroot(2)`, but this will prohibit it from accessing font files that it might need. Finally we could read the postscript code and make sure that it does not include any malicious commands, but this is impractical.

Our method provides transparency to the user and increased security since every data object has its access rights bound to its identity, preventing it from harming the system.

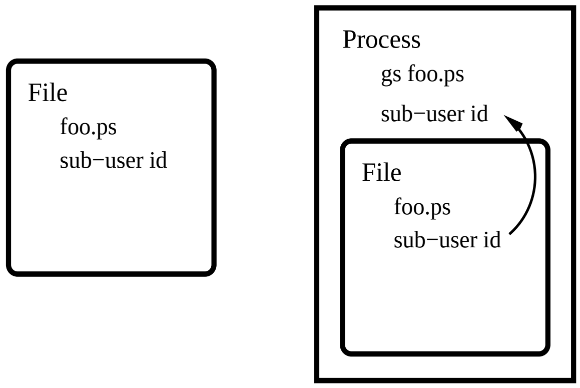

Figure 3: **In the left part of the Figure we see an object, in this case a postscript file foo.ps, with its associated sub–user id. The moment the ghostscript application opens file Foo.ps, it turns into a SubOS process and it inherits the sub–user id that was associated with the untrusted object. From now on, this process has the permissions and privileges associated with this sub–user id.**

3.1 Security Mechanism Enforcement

As we mentioned earlier in Section 3, every time the system accepts an incoming object it associates a sub-user id with it, depending on the credentials the object carries. The sub-user id is permanently saved in the `Inode` of the file that holds that object, which is now its immutable identity in the system and specifies what permissions it will have. It has essentially the same functionality as a UNIX user

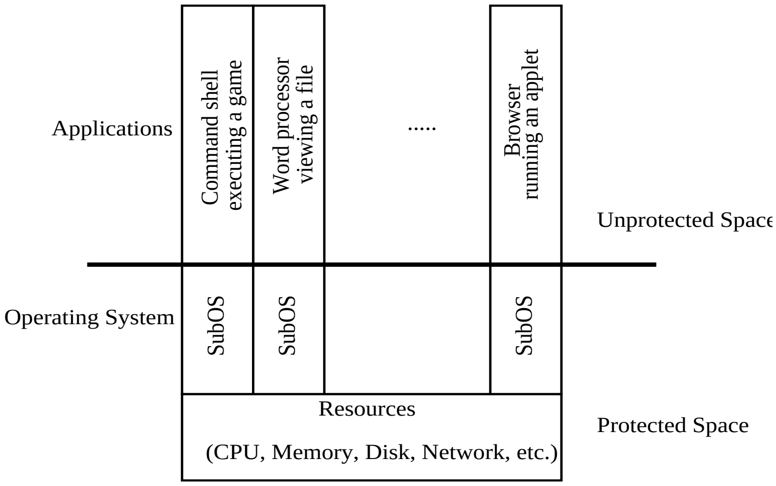

Figure 2: **Under SubOS enabled operating systems user applications that "touch" possibly malicious objects no longer maintain the user access rights, and only get restricted access to the underlying system.**

id. One can view this as the equivalent of a user logging in to the system.

Figure 4 shows the equivalence of the two mechanisms. In the top part of the figure we see the regular process of a user Bar logging in a UNIX system Foo and getting a user id. In the same way, objects that enter the system through ftp, mail, etc., "log in" and are assigned sub-user id's based on their (often cryptographically-verified) source.

4 The Browser Architecture

4.1 The Threat

The use of Java, JavaScript and VBScript in HTML pages is becoming ever more popular, furthermore HTML provides support for other scripting languages with the use of the `<SCRIPT>` tag [3]. Even though this functionality is primarily intended to enhance the capabilities of web pages and the "surfing experience" of the user, it is often used to attack unsuspecting hosts.

Even worse, the site or host is vulnerable even if the browser is behind the firewall and the document is a "secure" HTTPS-based document. JavaScript programs are executed within the security context of the page in which they were down-loaded, and

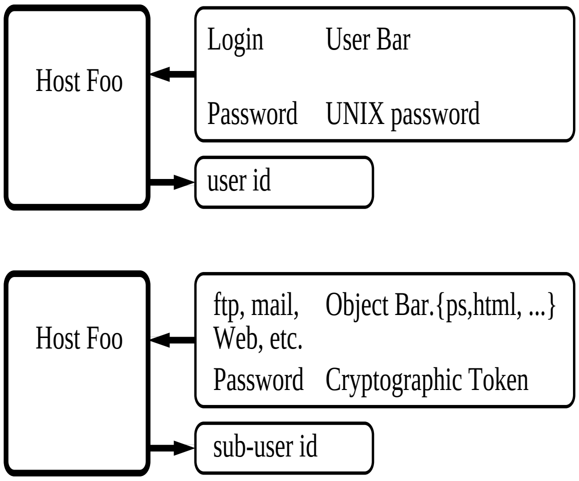

Figure 4: **In the top part of the Figure we see the regular process of a user Bar logging in a UNIX system Foo and getting a user id. In the same way objects that enter the system through ftp, mail, etc., "log in" using a cryptographic token, and are assigned sub-user id's.**

have restricted access to other resources within the browser. Some browsers running JavaScript may, in turn, have security flaws that allow the JavaScript program to monitor a user's browser more than what is considered safe or secure. In addition, it may be difficult or impossible for the browser user to determine if the program is transmitting information back to the web server. For instance, among other functions, JavaScript is able to monitor a user's browser activity by:

- Observing the URLs of visited documents as well as bookmarks.
- Observing the data filled into HTML forms (including passwords).
- Observing the values of cookies (that might hold critical information).

In Java the user may or may not be informed that an applet is being down-loaded into their browser. The real shock comes when a user inadvertently downloads a hostile applet. There are many different things hostile applets can do to wreak havoc on your system. Among a few of the most noteworthy are the following:

- Reveal information about your machine (e.g. details about passwords or structure of your system).
- Allocate resources to the point your machine "locks up" (i.e. denial of service attacks).
- Delete or alter files.
- Be just plain annoying (e.g. popping countless windows).

Hostile applets have also been known to have the capability to contact machines behind firewalls, send off a listing of a user's directories, track a user's actions through the web, generate machine code, make directories readable and writable, and send off email without intention [2].

4.2 Modular Approach

In our architecture we address the two security problems of Web browsers:

1. Helper applications running with the user's privileges.
2. Web pages that carry active content that is interpreted by the browser.

[2] There are a number of web sites that list hostile applets, JavaScript and VBScript, readily available for anyone interested in launching an attack.

To address these problems we will use the mechanisms provided by the SubOS-capable operating system, as well as a modular Web browser architecture. We divide the Web browser into three parts, according to its functionality. The first part is responsible for down-loading objects over the network, the second is responsible for displaying the content, and the last is a set of helper applications/interpreters used to process the content of the down-loaded objects. The design is presented in Figure 5

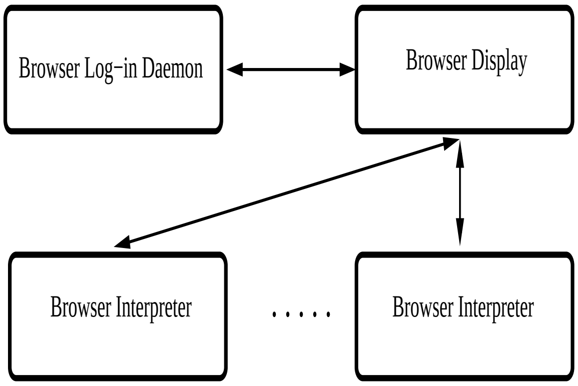

Figure 5: **The Web browser is comprised of three parts. The first part is responsible for down-loading objects from the net and assigning sub-user id's to them. The second provides the user interface of the browser. Finally the third is a set of processes that interpret the active code that is carried by the incoming objects.**

We decided against using an existing Web browser since that would require significant modification to its architecture. Down-loading and authentication of objects could be easily achieved by using a proxy, however execution of embedded code in HTML web pages would be a lot more challenging, since it would have to execute in a separate address space to maintain its security properties, as we discussed in Section 3.

4.2.1 Browser Log-in Daemon

Every object that is down-loaded by our browser log-in daemon is assigned a sub-user id, which is bound to some permissions, and is then stored in the file system. Assignment of sub-user id's is similar to the log in mechanism of UNIX. Objects that carry certificates are given more permissions than unauthenticated objects. For example an authenticated object might get access to `/home/user_foobar`, network access and unlimited resources, whereas an unauthenticated objects might only get access to `/tmp` with no access to the network and limited CPU time and memory allocation.

In the current implementation we use the URL address is used to select the sub-user id that will be assigned to the down-loaded object. This approach of course is not really secure, ideally we should use some sort of cryptographic token (e.g. a certificate) that is carried along with the down-loaded object.

4.2.2 Browser Display Daemon

The display daemon is responsible for providing the user interface of the our Web browser. It can make requests to the log-in daemon to down-load files, it is responsible for spawning interpreters to handle the incoming objects, and display HTML.

4.2.3 Browser Interpreter Daemon

The final part of our web browser is the set of interpreter daemons. These processes have dual functionality; they interpret HTML along with any possible active content embedded in the web page, and they execute the helper applications that handle incoming objects such as Perl, Postscript, etc.

Objects that are normally handled by helper applications are also assigned sub-user id's by the log-in daemon, the same way as ordinary web pages. When they are interpreted they are bound to the permissions of that sub-user id. This way users don't need to be queried about every arbitrary object they down-load of the net and also don't have to worry about executing possibly malicious code on their machine.

When the interpreter daemon encounters active code embedded in a web page (by encountering an `<APPLET>` or `<SCRIPT>` tag) it spawns a process to interpret the Java, JavaScript [1], or Perl code. The new process inherits the permissions of the parent process so the active code can never escape it's sandbox.

5 Related Work

Web browser security is topic that has received a great deal of attention since its so crucial in todays highly interconnected computing. However there have not been any satisfactory solutions so far. The primary proposed solution is secure interpreters for JavaScript, VBScript, Java, etc. [14, 15, 17, 10, 9]. Such solutions fail because their complexity. The more complex the implementation, the more likely it is to have bugs. Furthermore they don't address the issue of other helper applications like Perl or Tcl. When they are invoked, the user is queried, and this puts a lot of burden to the user.

Another language related technique used for ensuring security is code verification. This approach uses *proof-carrying code* [16] to demonstrate the security properties of the object. This means that the object needs to carry with it a formal proof of its properties; this proof can be used by the system that accepts it to ensure that it is not malicious. Code verification is very limiting since it is hard to create such proofs. Furthermore, it does not scale well; imagine creating a formal proof for every Web page.

A different approach relies on the notion of system call interception, as used by systems such as TRON [5], MAPbox [4], Software Wrappers [7] and Janus [8]. TRON and Software Wrappers enforce capabilities by using system call wrappers compiled into the operating system kernel. The syscall table is modified to route control to the appropriate TRON wrapper for each system call. The wrappers are responsible for ensuring that the process that invoked the system call has the necessary permissions. The Janus and MAPbox systems implement a user-level system call interception mechanism. It is aimed at confining helper applications (such as those launched by Web browsers) so that they are restricted in their use of system calls. To accomplish this they use `ptrace(2)` and the `/proc` file system, which allows their tracer to register a call-back that is executed whenever the tracee issues a system call. These systems are the most related to our work; however, our system differs in a major point. We view every object as a separate user, each with its own sub-user id and access rights to the system resources. This sub-user id is attached to every incoming object when it is accepted by the system, and stays with it throughout it's life, making it impossible for malicious objects to escape.

6 Conclusions

We have presented the architecture of a secure web browser, that protects against malicious incoming objects. We have implemented a first version of our prototype on a SubOS-capable OpenBSD 2.8 [2] operating system using Perl.

There are several advantages in our modular architecture versus the monolithic architecture of popular Web browsers, such as Netscape Navigator and Microsoft Internet Explorer. Our design adds a stage of authentication before any incoming object is processed. The burden of access control is moved from the browser and its helper applications, to the operating system, allowing for a simpler and therefore more secure design. Finally the user is not involved in the processing of incoming objects, and therefore cannot be tricked into executing hostile code. Presently however, our architecture requires that the operating system provides a data centric protection mechanism, that associates permissions and privileges to data objects. This limits us to our experimental SubOS-enabled OpenBSD operating system.

There are still some things that remain to be added to our prototype in order to offer more complete functionality:

- We currently don't support frames. Frames require special handling since each frame consists of an HTML document with possibly individual security properties. In future versions of our browser we will add this functionality to the browser display daemon.

- Only a subset of HTML was implemented so there are a number of tags that need to be added, along with their possible variables.

- We want to expand the `<SCRIPT>` tag to deal with additional embedded scripting languages other than JavaScript and Perl.

- Finally we need to have some kind of secure authentication mechanism for the browser log-in daemon. The possible solutions we are considering are either an additional tag that carries a certificate in the down-loaded web page, or a certificate attached to the HTTP request.

7 Acknowledgments

We would like to thank Jonathan M. Smith for his useful comments and guidance throughout the course of this work. We also like to thank the FREENIX 2001 anonymous reviewers and our "shepherd" Ken Coar for their comments and suggestions on improving this paper.

References

[1] NJS JavaScript Interpreter. http://www.bbassett.net/njs/.

[2] The OpenBSD Operating System. http://www.openbsd.org/.

[3] World Wide Web Consortium. http://www.w3.org/.

[4] Anurag Acharya and Mandar Raje. Mapbox: Using parameterized behavior classes to confine applications. In *Proceedings of the 2000 USENIX Security Symposium*, pages 1–17, Denver, CO, August 2000.

[5] Andrew Berman, Virgil Bourassa, and Erik Selberg. TRON: Process-Specific File Protection for the UNIX Operating System. In *USENIX 1995 Technical Conference*, New Orleans, Louisiana, January 1995.

[6] David Flanagan. *JavaScript The Definitive Guide*. O'Reilly, 1998.

[7] Tim Fraser, Lee Badger, and Mark Feldman. Hardening COTS Software with Generic Software Wrappers. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 1999.

[8] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A Secure Environment for Untrusted Helper Applications. In *USENIX 1996 Technical Conference*, 1996.

[9] Li Gong. *Inside Java 2 Platform Security*. Addison-Wesley, 1999.

[10] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison Wesley, Reading, 1996.

[11] http://www.cert.org/advisories/.

[12] Sotiris Ioannidis and Steven M. Bellovin. Sub-Operating Systems: A New Approach to Application Security. Technical Report MS-CIS-01-06, University of Pennsylvania, February 2000.

[13] R. Kaplan. SUID and SGID Based Attacks on UNIX: a Look at One Form of then Use and Abuse of Privileges. *Computer Security Journal*, 9(1):73–7, 1993.

[14] Jacob Y. Levy, Laurent Demailly, John K. Ousterhout, and Brent B. Welch. The Safe-Tcl Security Model. In *USENIX 1998 Annual Technical Conference*, New Orleans, Louisiana, June 1998.

[15] Gary McGraw and Edward W. Felten. *Java Security: hostile applets, holes and antidotes*. Wiley, New York, NY, 1997.

[16] G. C. Necula and P. Lee. Safe, Untrusted Agents using Proof-Carrying Code. In *Lecture Notes in Computer Science Special Issue on Mobile Agents*, October 1997.

[17] Dan S. Wallach, Dirk Balfanz, Drew Dean, and Edward W. Felten. Extensible Security Architectures for Java. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, October 1997.