Sub-Operating Systems: A New Approach to Application Security

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

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

Jonathan M. Smith
jms@dsl.cis.upenn.edu
University of Pennsylvania

Abstract

Users regularly exchange apparently innocuous data files using email and ftp. While the users view these data as passive, there are situations when they are interpreted as code by some system application. In that case the data become "active". Some examples of such data are Java, JavaScript and Microsoft Word attachments, each of which are executed within the security context of the user, allowing potentially arbitrary machine access. The structure of current operating systems and user applications makes solving this problem challenging.

We propose a new protection mechanism to address active content, which applies fine-grained access controls at the level of individual data objects. All data objects arriving from remote sources are tagged with a non-removable identifier. This identifier dictates its permissions and privileges rather than the file owner's user ID. Since users possess many objects, the system provides far more precise access control policies to be enforced, and at a far finer granularity than previous designs.

1 Introduction

Most classical work on computer security focuses on operating systems. One of the fundamental tasks of an operating system is resource allocation and control: making sure that different users have fair access to shared resources, such as disk space and CPU time, all the while ensuring that access restrictions are honored. A secure system is one where these controls are effective in the face of deliberate attempts at subversion.

The advent of ubiquitous networking has changed all this. While operating system security is still important, many of the new threats arise because of network activity. Often, these threats are data-driven, and within the confines of a single user's protection boundaries. For example, pieces of mobile code, run with the permissions of some user, may attempt to steal or destroy files belonging to that user.

Naturally, the applications accepting the mobile code attempt to guard against such things. But their record has at best been mixed. The problem is more acute because many more network objects can, in some sense, be considered "code", even if not intended as such.

Many of the security problems that have occurred have been due to problems in determining file permissions. Applications typically resort to pattern-matching, a dangerous and error-prone technique. For example, CERT Advisory CA-98.04 describes a problem on a Web server that didn't properly check the so-called "short name" when the actual name of the file did not fit into the legacy 8.3 format. Similarly, Advisory CA-2000-15 describes how a Java applet could open `file:` URLs, thus reading files from the local machine. In the first case, the application misunderstood the operating system's file name semantics; in the second, the check was omitted entirely.

Operating systems rarely have such problems. File permissions are associated with the *file* itself; any attempt to open the file will cause a permission check, regardless of how the file was accessed. OS security failures typically occur when a privileged program is tricked into opening a file; the kernel's access control mechanism is simple enough that it is rarely at fault.

In this paper, we describe an architecture, called SubOS, for fine grain control of data objects. In SubOS data objects are treated as users (sub-users), with their own privileges and permissions. Figures 1 and 2 demonstrate the differences between a regular and a SubOS-enabled operating system. On a regular operating system user applications execute with the permissions of the user and have access to the underlying system identical to that of the user Under SubOS, applications execute with the permissions of the data object (sub-user) they operate on. This allows for finer grain control, and therefore greater protection from malicious data objects.

The paper is organized as follows. In Section 2 we discuss the motivation behind this work. In Sections 3 and 4 we present the design and implementation details of a SubOS-capable OpenBSD [2] system, and two applications that benefit from such an architecture. In Section 5 we dis-

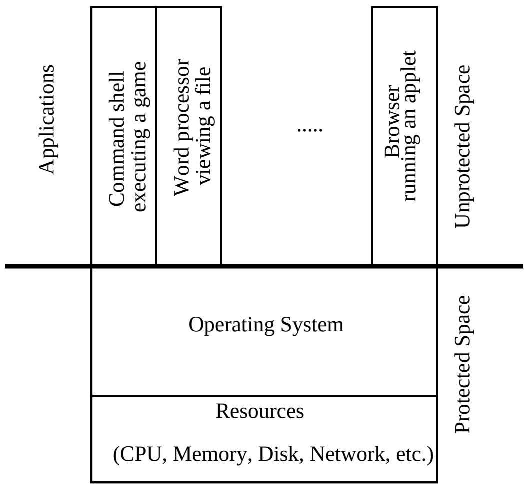
Applications
Command shell executing a game
Word processor viewing a file
.....
Browser running an applet
Unprotected Space
Operating System
Resources
(CPU, Memory, Disk, Network, etc.)
Protected Space

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

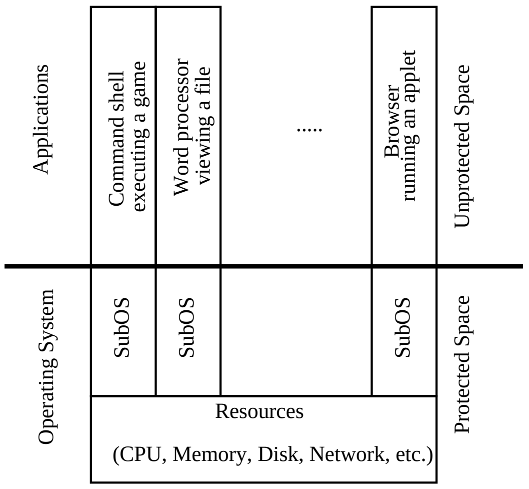
Applications
Command shell executing a game
Word processor viewing a file
.....
Browser running an applet
Unprotected Space
Operating System
SubOS
SubOS
SubOS
Resources
(CPU, Memory, Disk, Network, etc.)
Protected Space

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.

cuss work that is related to SubOS, and finally we conclude in Section 6.

2 Motivation

A number of trends in computing are fueling the need for a more flexible, yet stricter security model in operating systems.

2.1 Information Exchange

With the growth of the Internet, exchange of information over wide-area networks has become essential for both applications and users. Modern applications often fetch help files and other data over the World Wide Web. In extreme cases, like some versions of the BSD UNIX operating system, even whole operating systems install and upgrade themselves over the network. However, the most common case is electronic mail. Users regularly receive mail from unknown sources with a number of possibly malicious attachments. The attached documents use vulnerabilities in the helper applications that are invoked to process them, which in turn could compromise system security. The need for connectivity and exchange of information even at this most basic level is therefore a major threat to security.

It is also the case that seemingly inactive objects like Web pages or email messages are very much active and potentially dangerous. One example is JavaScript 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 downloaded (CERT Advisory CA:97.20). It is obvious that such behavior automatically compromises the user's privacy.

Another example is the use of Multi-purpose Internet Mail Extensions (MIME). The MIME format permits email to include enhanced text, graphics, and audio in a standardized and inter–operable manner. `Metamail(1)` is a package that implements MIME. Using a configurable `mailcap(4)` file, `metamail(1)` determines how to treat blocks of electronic mail text based on the content as described by email headers. A condition exists in `metamail(1)` in which there is insufficient variable checking in some support scripts. By carefully crafting appropriate message headers, a sender can cause the receiver of the message to execute an arbitrary command if the receiver processes the message using the `mailcap(4)` package (CERT Advisory CA:97.14) [1].

2.2 Application Complexity

But the problem is deeper than obvious forms of mobile code. Given the increasingly complex environment presented to many applications, we assert that these applications have many of the characteristics of operating systems, and should be implemented as such.

Even simple HTTP requests return a complex object, wherein the remote side tells the local browser what to do, up to and including a request to run certain applications. Print spoolers have to check file access permissions. Email can be delivered directly to programs. Web servers must run scripts, often via an interpreter, while denying direct access to the interpreter and perhaps ensuring that one script does not access or modify the private data of another script. All of these applications should worry about resource consumption. And these, of course, are the characteristics of operating systems. In fact, arbitrating access to various objects is more or less the definition of what an operating system does.

However, re–implementing an operating system with each new application would be extreme. Instead, our goal is to add sufficient functionality to an existing system so that applications can rely on the base operating system to carry out its own particular security policy. That security policy, in turn, can reflect its own particular needs and its degree of certainty as to the identity of users.

2.3 Inadequate Operating System Support

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 [18].

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 today's 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. Applications such as mailers, Web browsers, word processors, *etc.*, become responsible for accepting requests, granting permissions and managing resources. All this is what is traditionally done by operating systems. These applications, 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, including macros in Microsoft Word, JavaScript, malicious Postscript and PDF documents, *etc.*

We wish to offer users flexible security mechanisms that restrict access to system resources to the absolute minimum necessary.

3 The SubOS Architecture

SubOS is a process–specific protection mechanism. Under SubOS any application (e.g. ghostscript, Perl, *etc.*) that operates on possibly malicious objects (e.g. Postscript files, Perl scripts, *etc.*) inherits the identity of that object. Any further system accesses that application makes are restricted by *that* identity, instead of the *user identity*. We will call these applications SubOS processes, or sub-processes in the rest of this paper. 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 3.1.1. 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 objects, 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(2)` 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

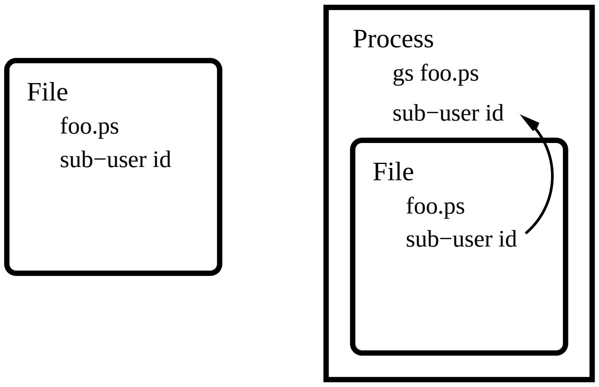

Figure 3. **In the left part of the Figure we see an object, in this case a postscript file `foo.ps`, with its associated sub–user id. The moment the ghostscript application opens file Foo.ps, it turns into a SubOS process and it inherits the sub–user id that was associated with the untrusted object.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

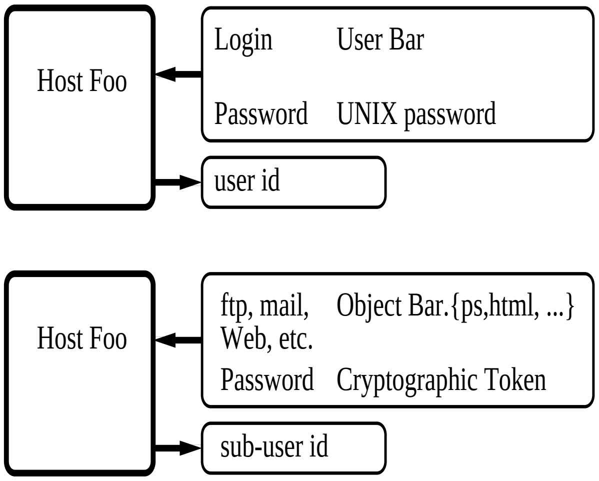

Figure 4. **In the top part of the Figure we see the regular process of a user Bar logging in to a UNIX system Foo and getting a user id. In the same way objects that enter the system through ftp, mail, *etc.*, "log in" using a cryptographic token, and are assigned sub-user id's.**

relying on safe languages fails [1]. 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.

3.1 Implementation

For our development platform we decided to use the OpenBSD operating system [2]. OpenBSD provides an attractive platform for developing security applications because of the well-integrated security features and libraries (an IPsec stack, SSL, KeyNote, *etc.*). However, there is nothing inherent in the SubOS architecture that limits us to UNIX like operating systems, so similar implementations are possible for operating systems like Microsoft Windows. The main advantage of our kernel implementation is that the additional security mechanisms will be largely transparent to the applications. Specifically, although the applications may need to be aware of the SubOS structure, they will not need to worry about access control or program containment.

3.1.1 Data Object Identifiers

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 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 to 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, as we will see in Sections 3.1.4 and 3.1.5.

Enhancing applications to utilize the functionality of the SubOS system require either making minor modification to the application code, or interposing a proxy that assigns sub-user id's to objects arriving via proxied services, *e.g.* ftp, http, and mail.

3.1.2 Sandboxing

The most basic operation supported by SubOS is the inheritance of the sub-process id from an inactive file system object to a running process. To accomplish this we extended the `open(2)` system call. When it is used on objects that contain sub–user id's, it copies the sub–user id to the `proc` structure of that process (Figure 3). At that point the process becomes a SubOS process bound to that sub–user id.

It is crucial that a sub-process can never "escape" its sub-process status. To enforce this, whenever a sub-process forks and execs, the identity is inherited by the child process. To achieve this we extended the `fork(2)` and `exec(2)` system calls to have created processes inherit that status. Furthermore we modified the `creat(2)` system call, so that any files created by sub–processes have the sub–user id of the creator assigned in their `Inode`. Finally sub-processes are not allowed to execute setuid programs, to enforce this we block the setuid related (`setuid(2), seteuid(2), setgid(2), setegid(2)`) system calls in the kernel.

It is not clear that that is the right choice. However, UNIX has traditionally had trouble when setuid programs invoked other setuid programs. To give just one historical example, in the days when the `mkdir(2)` call was implemented by executing a setuid—root program, subsystems that were themselves setuid had trouble creating directories.

3.1.3 Resource Protection

The SubOS mechanisms must protect the various resources of the users computer from viruses, Trojan Horses, worms, etc. In order to do so, it should monitor the creation of network connection, accesses to the file system, execution time of processes and allocation of physical memory, that might result from malicious code in untrusted objects.

By default a SubOS process is not allowed to create network connections. We accomplish this by filtering network related system calls. It is however possible to set up policies that will allow certain sub–processes to access the network by setting up the hosts they are allowed to connect to, the port, and the protocol to be used.

A practical implementation would require considerable attention to policies, including wild cards for port numbers, network masks for the host, etc. It might also be desirable to include certain known-safe local host/port combinations. For example, we may wish to permit open access to a local DNS proxy, for safe name resolution. On the other hand, wide-open access to a real name server might permit the controlled process to map local domains, which may be undesirable.

In order for the SubOS to restrict file system accesses we introduce the notion of a *view*. The view refers to the permissions a sub-process has to parts of the directory tree. Sub-processes don't use the permission bits that are normally used by processes (user, group, other). Rather, they have their own permissions that are defined in a configuration file, maintained by the user or administrator. This is very much like `chroot(2)` but more like pruning the directory tree of the file system than setting a new root.

The extended permission bits are added in lists in the `inodes` of the files specified in the configuration file. Every time the kernel identifies a file system access originating from a sub–process, it traverses the list in the corresponding `inode` in order to locate the permissions that apply to the sub-user id of that sub-process. It then uses those permission bits, instead of the normal bits set for user, group or other, to determine whether to allow the access or not. If there are no permissions set for that sub-process the request is denied by default.

Finally execution time as well as memory allocation should also be monitored. This way, malicious objects (such as Java applets that run under a Web browser) will not hamper the smooth operation of the system. Our current working prototype lacks the appropriate controls for this type of enforcement, we are however in the process of implementing the necessary controls for the next version of our system. There are a number of things that need to be considered. Most importantly access to the `setpriority(2)` or `setrlimit(2)` system calls should be restricted, prohibiting sub-processes from executing at a higher priority than the parent process and limiting the amount of system resources they can allocate. Additionally, there must be a bound to the number of times a sub-process is allowed to *fork* in order to prevent possible process pollution in the system. Finally we need to add a form of accounting to monitor the amount of resources all the spawned sub-processes consume.

3.1.4 Sub-Users

In order for a SubOS to be effective, different sub-user ids must be assigned to different protection domains. Just how this is done depends on the the application, on how the file arrived on the local system, and on any credentials it carries.

For emailed files, the sender's identity is used to select the sub-user id, naturally, such mail should be digitally signed. Mail from a previously-unknown user, or mail that cannot be assigned with enough confidence to a particular sender, receives a new sub-user id.

For Web browsers, finer-grained protection is desirable. Each site visited is assigned its own sub-user id, thus preventing one site from interfering with another's content. This could, for example, have prevented the "Frame Spoof" bug in Internet Explorer (MS98-020) [1].

3.1.5 Accessing Multiple Objects

So far we have assumed that sub-processes will operate on only one object at a time. However it is possible for a sub-process to open multiple objects, each with its own sub-user id. When a sub-process opens another object containing a sub-user id it also inherits that new id, and the new permissions would depend on a combination of the individual permission as dictated by the system policy.

This is easily accomplished in the case of CPU and memory allocation; the new sub-process will have the minimum of the two for allocated memory and CPU time. In the case of network and file system access, any request is denied unless it is allowed by the permissions of *all* inherited sub-user id's.

4 SubOS Applications

To demonstrate the functionality of our SubOS enabled operating system we identified two applications that are most commonly targeted by hostile objects. Mailers and Web browsers are often attacked by a number of malicious attachments, and would therefore benefit from our security architecture.

4.1 A Secure Mailer

To test the functionality of our current prototype we modified a mailer, `mh(1)`, to take advantage of the SubOS architecture. To do this we extended `mh(1)` to implement a *login*–like mechanism. Depending on the source of the message—ideally, this should be cryptographically verified, though we have not yet implemented that portion—`mh(1)` will attach a sub–user id to that file when it saves it. `Mh(1)` assigns sub–user id's using a file, similar to the UNIX `/etc/passwd`, that matches e–mail addresses to id's.

Any helper application that is invoked—often automatically—to process the message, will inherit the sub-user identity. If the message contains active, malicious code, its effects will be contained by the limited permissions assigned to that sub-user id, protecting the rest of the system.

4.2 A Secure Browser

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

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

To address the problems we will use the mechanism provided by the SubOS-capable operating system, as well as a modular Web browser architecture. The implementation is sketched here; a more complete description is in [16]. We divided the Web browser into three parts, according to its functionality. The first part is responsible for downloading 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 downloaded objects. The design is presented in Figure 4.2

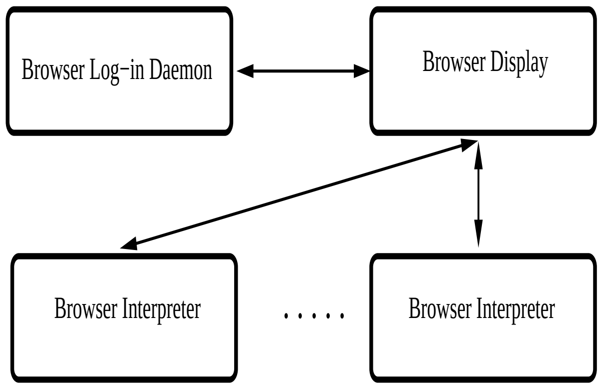

Figure 5. The Web browser is comprised of three parts. The first part is responsible for downloading objects from the net and assigning sub-user id's to them. The second provides the user interface of the browser. Finally the third is a set of processes that interprets the active code that is carried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

4.2.1 Secure Browser Components

Every object that is downloaded by the browser log-in daemon is assigned a sub-user id, which is bound to some permissions, and is then stored in the file system, similarly to Section 4.1. Objects that carry certificates, such as pages downloaded from Web sites that use `https`, may be given more permissions than are unauthenticated objects. For example an authenticated object might get access to `/home/foobar`, network access and unlimited resources, whereas an unauthenticated object might only be granted access to `/tmp` with no access to the network and limited memory and cpu time allocation. The display process provides the user interface of the our Web browser. It can make requests to the log-in daemon to download files; it is also responsible for spawning interpreters to handle the incoming objects, and display HTML.

Any active code is executed within the context of the interpreters. They handle inline scripts like JavaScript, as well as other types of active code, such as Postscript and Perl. Since the objects they interpret are bound by their sub-user id, which was assigned to them when they first entered the system, they cannot cause any damage.

5 Related Work

The area of operating system security is a field that has received a great deal of attention, and has been researched extensively. However, the ever-increasing demand and need for communication and openness has put new strains on operating systems. Communication environments like the Internet require us to solve a whole new set of problems that researchers have just recently started to address. In this section we focus our attention to work that is directly related to ours.

There are several methods for intrusion prevention in operating systems, ranging from type-safe languages [20, 22, 30, 14, 13], fault isolation [28] and code verification [25], to operating system-specific permission mechanisms [21, 26], system call interposition [12, 4, 3, 5] and system call interception [6, 10, 11, 7, 29, 24].

Capabilities and access control lists are the most common mechanisms operating systems use for access control. Such mechanisms expand the UNIX security model and are implemented in several popular operating systems, such as Solaris and Windows NT [8, 9]. However they offer no protection for the user against programs owned by the user, which may contain errors, Trojan Horses, or viruses.

The Flask system [26] extends the idea of capabilities and access cotrol lists by the more generic notion of the *security policy*. The Flask micro kernel system relies on a security server for policy decisions and on an object server for enforcement. Every object in the system has an associated security identifier very similar to our notion of a sub-user id. Requests coming from objects are bound by the permissions associated with their security identifier. However Flask does not address the threat SubOS is trying to protect against, namely passive objects becoming active and then executing with the permissions of the running process. As a minor issue, we have demonstrated that our prototype can be easily implemented as part of a widely used, commodity operating system, as opposed to an experimental micro kernel.

The traditional Orange Book-style systems offer protection against violation of security levels by malicious programs. But there is no barrier to attacks on files at the current security level, nor to attacks at that security level over the network. For example, a Top Secret worm can would still be able to spread, though it would only be able to infect other Top Secret-rated systems.

Reeds and McIlroy's unique implementation of the Orange Book's security policies [23] bears a strong conceptual resemblance to the SubOS scheme. Rather than assigning a process or a file fixed access rights or labels, these "float" in response to the program's execution. A process that opens a file marked Top Secret acquires a Top Secret label; any files that it writes are also marked Top Secret. Permissions are thus data-driven, as in SubOS.

A different approach relies on the notion of call interposition. Systems like [12, 4, 3, 5] operate at user level and confine applications by filtering access to system calls. To accomplish this they rely on `ptrace(2)`, the `/proc` file system, and special shared libraries. Another category of systems [6, 10, 11, 7, 29, 24], goes a step further. They intercept system calls inside the kernel, and use policy engines to decide whether to permit the call or not. 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.

In [17] the authors identify the dangers of active content and the need to contain it. They authenticate incoming objects and grant them access rights. These access rights identify which interpreters are allowed to operate on the objects. Furthermore these interpreters are also "sanitized" so that they don't include any unsafe calls. Our system offers much finer access control, enforced by the operating system kernel.

The methods that we mentioned so far rely on the operating system to provide with some sort of mechanism to enforce security. There are, however, approaches that rely on safe languages, [20, 27, 19, 15] the most common example being Java [22, 13]. In Java applets, all accesses to unsafe operations must be approved by the security manager. The default restrictions prevent accesses to the disk and network connections to computers other than the server the applet was down-loaded from. Our system is not only restricted to a limited set of type safe languages. We can secure any process running on the system that has touched some untrusted object.

Code verification is another technique for ensuring security. This approach uses *proof-carrying code* [25] to demonstrate the secur ity 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.

6 Conclusions

We have designed and implemented an object-specific protection mechanism to contain untrusted data. We restrict the environment that such objects can operate in, and the resources they can access, by extending the UNIX security model to assign sub-user id's to them and then treating them like regular users. The implementation is part of the kernel of the operating system, since that is the only natural and secure place for security mechanisms to enforce policies. SubOS is a working prototype implemented as part of the OpenBSD operating system. Finally, we have shown how SubOS relates to other security mechanisms and how it strengthens operating system security.

Acknowledgements

This work was supported by AT&T, by the DoD University Research Initiative (URI) program administered by the Office of Naval Research under Grant N00014-01-1-0795, and DARPA under Contracts F39502-99-1-0512-MOD P0001 and F30602-01-2-0537.

References

[1] CERT Advisories. http://www.cert.org/advisories/.

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

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

[4] A. Alexandrov, P. Kmiec, and K. Schauser. Consh: A confined execution environment for internet computations, December 1998.

[5] R. Balzer and N. Goldman. Mediating connectors: A non-bypassable process wrapping technology. In *Proceeding of the 19th IEEE International Conference on Distributed Computing Systems*, June 1999.

[6] A. Berman, V. Bourassa, and E. Selberg. TRON: Process-Specific File Protection for the UNIX Operating System. In *Proceedings of the USENIX 1995 Technical Conference*, New Orleans, Louisiana, January 1995.

[7] C. Cowan, S. Beattie, C. Pu, P. Wagle, and V. Gligor. SubDomain: Parsimonious Security for Server Appliances. In *Proceedings of the 14th USENIX System Administration Conference (LISA 2000)*, Mar. 2000.

[8] H. Custer. *Inside Windows NT*. Microsoft Press, 1993.

[9] H. Custer. *Inside the Windows NT File System*. Microsoft Press, 1994.

[10] T. Fraser, L. Badger, and M. Feldman. Hardening COTS Software with Generic Software Wrappers. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 1999.

[11] D. P. Ghormley, D. Petrou, S. H. Rodrigues, and T. E. Anderson. SLIC: An Extensibility System for Commodity Operating Systems. In *Proceedings of the 1998 USENIX Annual Technical Conference*, pages 39–52, June 1998.

[12] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A Secure Environment for Untrusted Helper Applications. In *Procedings of the 1996 USENIX Annual Technical Conference*, 1996.

[13] L. Gong. *Inside Java 2 Platform Security*. Addison-Wesley, 1999.

[14] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, Reading, 1996.

[15] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. PLAN: A Programming Language for Active Networks. Technical Report MS-CIS-98-25, Department of Computer and Information Science, University of Pennsylvania, February 1998.

[16] S. Ioannidis and S. M. Bellovin. Building a Secure Browser. In *Proceedings of the Annual USENIX Technical Conference, Freenix Track*, June 2001.

[17] T. Jaeger, A. D. Rubin, and A. Prakash. Building systems that flexibly control downloaded executable content. In *Proceedings of the 1996 USENIX Security Symposium*, pages 131–148, San Jose, Ca., 1996.

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

[19] X. Leroy. Le système Caml Special Light: modules et compilation efficace en Caml. Research report 2721, INRIA, November 1995.

[20] J. Y. Levy, L. Demailly, J. K. Ousterhout, and B. B. Welch. The Safe-Tcl Security Model. In *USENIX 1998 Annual Technical Conference*, New Orleans, Louisiana, June 1998.

[21] D. Mazieres and M. F. Kaashoek. Secure Applications Need Flexible Operating Systems. In *The 6th Workshop on Hot Topics in Operating Systems*, May 1997.

[22] G. McGraw and E. W. Felten. *Java Security: hostile applets, holes and antidotes*. Wiley, New York, NY, 1997.

[23] M. D. McIlroy and J. A. Reeds. Multilevel security in the unix tradition. *Software Practice and Experience*, 22(8):673–694, 1992.

[24] T. Mitchem, R. Lu, and R. O'Brien. Using Kernel Hypervisors to Secure Applications. In *Proceedings of the Annual Computer Security Applications Conference*, Dec. 1997.

[25] 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.

[26] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Anderson, and J. Lepreau. The flask security architecture: System support for diverse security policies. In *Proceedings of the 2000 USENIX Security Symposium*, pages 123–139, Denver, CO, August 2000.

[27] J. Tardo and L. Valente. Mobile Agent Security and Telescript. In *Proceedings of the 41st IEEE Computer Society Conference (COMPCON)*, pages 58–63, February 1996.

[28] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient Software–Based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 203–216, December 1993.

[29] K. M. Walker, D. F. Stern, L. Badger, K. A. Oosendorp, M. J. Petkac, and D. L. Sherman. Confining root programs with domain and type enforcement. In *Proceedings of the 1996 USENIX Security Symposium*, pages 21–36, July 1996.

[30] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. Extensible Security Architectures for Java. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, October 1997.