January 2001

Sub-Operating Systems: A New Approach to Application Security

Sotiris Ioannidis
University of Pennsylvania

Steven M. Bellovin
AT&T Labs

Recommended Citation

Sotiris Ioannidis and Steven M. Bellovin, "Sub-Operating Systems: A New Approach to Application Security", . January 2001.

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-01-06.

Sub-Operating Systems: A New Approach to Application Security

Abstract

In the current highly interconnected computing environments, users regularly use insecure software. Many popular applications, such as Netscape Navigator and Microsoft Word, are targeted by hostile applets or malicious documents, and might therefore compromise the integrity of the system. Current operating systems are unable to protect their users from these kinds of attacks, since the hostile software is running with the user's privileges and permissions. We introduce the notion of the SubOS, a process-specific protection mechanism. Under SubOS, any application that might deal with incoming, possibly malicious objects, behaves like an operating system. It views those objects the same way an operating system views users - it assigns *sub-user id's* - and restricts their accesses to the system resources.

Keywords

secure systems, capabilities, process-specific protection

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-01-06.

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

Abstract

In the current highly interconnected computing environments, users regularly use insecure software. Many popular applications, such as Netscape Navigator and Microsoft Word, are targeted by hostile applets or malicious documents, and might therefore compromise the integrity of the system. Current operating systems are unable to protect their users from this kind of attacks, since the hostile software is running with the user's privileges and permissions.

We introduce the notion of the SubOS, a process-specific protection mechanism. Under SubOS, any application that might deal with incoming, possibly malicious objects, behaves like an operating system. It views those objects the same way an operating system views users—it assigns *sub-user id's*—and restricts their accesses to the system resources.

Keywords: Secure systems, capabilities, process–specific protection.

1 Introduction

Many important applications, such as mailers, Web browsers, word processors, etc., have many of the characteristics of operating systems. In particular, they accept requests from a variety of mutually-suspicious sources, grant different permissions based on the source (or other attributes, such as a cryptographic token), and mediate access to assorted resources. But applications are poorly suited to this task. For example, they have to implement file access restrictions by matching file names against assorted patterns. History shows, however, that that approach is fraught with danger (i.e., CERT Advisories CA:98-04 and CA:97-03). Real operating systems, which bind permissions to the protected objects, rarely have many problems like that.

In this paper we introduce the notion of a sub-operating system (called SubOS hereafter). A SubOS is an application that might have to operate on untrusted objects. By the term *untrusted object* we refer to any incoming file to our system, such as a Word document received in the mail, a postscript file down–loaded from some ftp site, or a Java applet that a browser might load from a Web page. These applications use operating system protection mechanisms to implement their own.

More precisely, applications that "touch" possibly malicious objects, like the ones listed above, will no longer maintain the users privileges, but will rather get restricted access rights to the underlying resources. Figures 1 and 2 display the differences of a regular, and a SubOS enabled operating system.

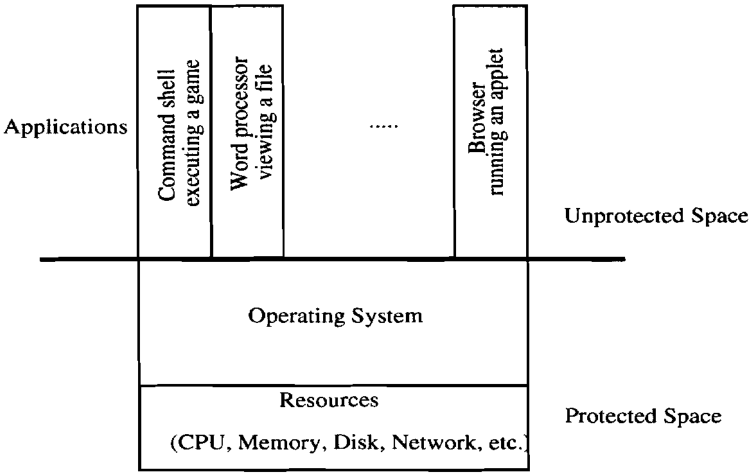
Applications
Command shell executing a game
Word processor viewing a file
.....
Browser running an applet
Unprotected Space
Operating System
Resources
(CPU, Memory, Disk, Network, etc.)
Protected Space

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

The paper is organized as follows. In Section 2 we discuss the motivation behind this work. In Section 3 we present the design and implementation details of a SubOS-capable OpenBSD [11] system. In Section 4 we discuss work that is related to SubOS, and finally we conclude in Section 5.

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 flavors 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

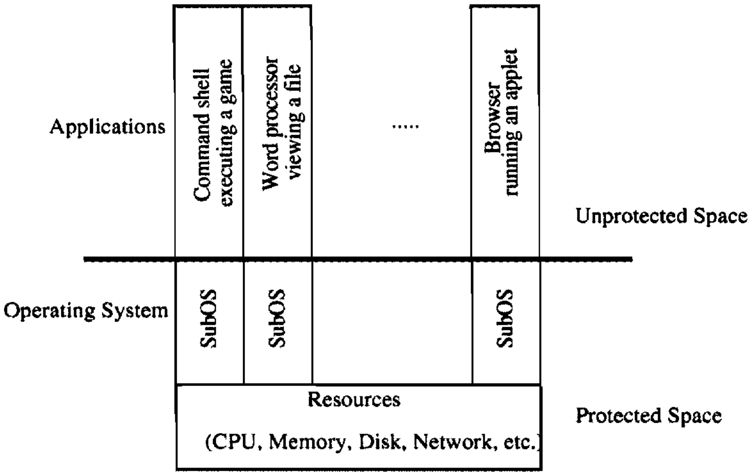
Applications
Command shell executing a game
Word processor viewing a file
.....
Browser running an applet
Unprotected Space
Operating System
SubOS
SubOS
SubOS
Resources
(CPU, Memory, Disk, Network, etc.)
Protected Space

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.

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 e-mail 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 Multipurpose 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) [10].

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 have to 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 [12].

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. 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 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

3.1 What is SubOS?

SubOS is a process–specific protection mechanism. 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. 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 3.3. 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 it's 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.

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 [10]. 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

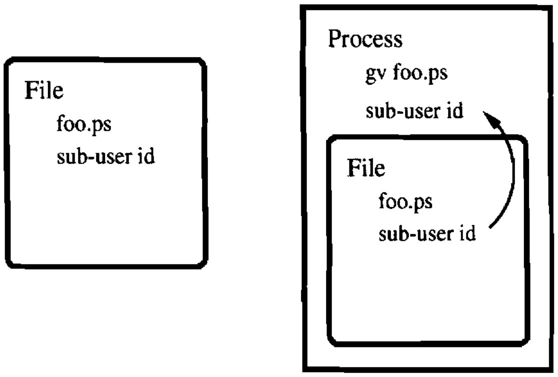

Figure 3: In the left part of the Figure we see an object, in this case a postscript file foo.ps, with its associated sub–user id. The moment the ghostscript application opens file Foo.ps, it turns into a SubOS process and it inherits the sub–user id that was associated with the untrusted object. From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

each object has its access rights bound to its identity, preventing it from harming the system.

3.2 Where should SubOS be Implemented?

The most important design decision is where we should add the SubOS functionality. The two possible answers are user level and kernel level; each has its advantages and disadvantages.

A user level approach would require each application of interest to be linked with a library that will provide the required security mechanisms. This has the advantage that it is operating system-independent, and so does not require any changes in the kernel code. However such a solution requires rewriting the application in such a way as to use the security mechanisms. Since there are a lot more applications than operating systems, this approach is not scalable and also more likely to have implementation errors.

An alternative user level implementation would be similar to that taken by [6]. Processes that might pose potential danger to the system have their system calls *traced*, using `ptrace(2)` or a similar facility. Using this approach, the application runs until it performs a system call. At this point, it is put to sleep and a tracing process wakes up. The tracing process determines which system call was attempted, along with the arguments of the call. It can then determine whether to allow or deny this system call based on policies set by either the user or the administrator.

For a kernel level approach, we would need access to the operating system source code. This restricts our prototype to open source operating systems like

BSD and Linux. [1] However there is no other constraint limiting us to UNIX like operating systems, and similar implementations are possible for operating systems like Microsoft Windows. The main advantage of this approach is that the additional security mechanisms will be large 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.

3.3 How does SubOS enforce its Security Mechanisms?

As we mentioned earlier, 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 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.

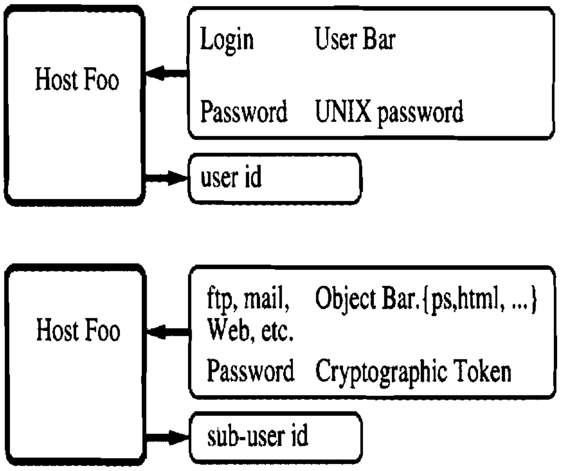

Figure 4: In the top part of the Figure we see the regular process of a user Bar logging in a UNIX system Foo and getting a user id. In the same way objects that enter the system through ftp, mail, etc., "log in" using a cryptographic token, and are assigned sub-user id's.

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

[1] Sometimes, operating systems are structured to permit easy additions to the kernel, even without source code availability.

mh(1) to implement a *login*–like mechanism. Depending on the source of the message—ideally, this should be cryptographically verified—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.

Similar implementations are possible for other applications, like ftp(1), or Web browsers.

3.4 What should SubOS Protect?

The SubOS mechanisms should 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.

3.4.1 Process-related Controls

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.

3.4.2 Controlling Network Connections

The way SubOS processes protect against unauthorized network use is by filtering the network related system calls, using a firewall-like mechanism. To do that it uses a list of *firewall entries* as shown in Figure 5.

By default a SubOS process is not allowed to create network connections. If we want to allow specific SubOS process to use the network we need to add a

```
struct  FWE     {
        int     subp_pid;
        int     host;
        int     port;
};
```

Figure 5: Firewall entry.

firewall entry in the kernel specifying the sub–user id, the host it is allowed to connect to, and the port.

Implementing policies similar to Java's—that a host can connect back to the host the applet was originally loaded from—requires more bookkeeping at the application level. Specifically, some database mapping sub–user ids to network policies must be maintained. While policies are always necessary, the actual permission bits are maintained in the file system for file accesses.

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 opean 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.

When a SubOS process enters the kernel on a network–related system call, the firewall entry list is traversed and if the right permissions are found the system call is allowed to continue; otherwise we return with an error indicator.

3.4.3 Controlling File System Accesses

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, which the user or the administrator is responsible for maintaining (Figure 6). This is very much like `chroot(2)` but more like pruning the directory tree of the file system than setting a new root. In the example in Figure 6 both sub–processes are allowed to execute commands `cp(1)` and `ls(1)`, which are typical utilities in `/bin`, and both have full access rights to `/tmp`. However each one has it's own private subdirectory under `/home/foobar/.netscape`.

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 uses those permission bits instead of the normal bits set for user, group or other.

For example, looking at Figure 6, the `inode` for `/home/foobar/.netscape` will have an ACL with two entries, for sub–user id's 1024 and 1025 and execute

```
Subp_pid    Path                            Permissions
# allow execute access to the commonly used commands ls and cp
1024,1025   /                               execute
1024,1025   /bin                            execute
1024,1025   /bin/ls                         execute
1024,1025   /bin/cp                         execute
# allow full access rights to the temporary directory /tmp
1024,1025   /tmp                            read write execute
# give each sub-process full access rights
# to it's own subdirectory
1024,1025   /home                           execute
1024,1025   /home/foobar                    execute
1024,1025   /home/foobar/.netscape          execute
1024        /home/foobar/.netscape/sub1     read write execute
1025        /home/foobar/.netscape/sub2     read write execute
```

Figure 6: Example permissions file. This file holds the permissions that SubOS processes with sub–user id's 1024 and 1025 have, in the file hierarchy.

permissions for both. However the `inode` for `/home/foobar/.netscape/sub1` will only have an entry for sub–user id 1024. If the sub-process with id 1025 tries to access `/home/foobar/.netscape/sub1`, the kernel will first identify that the access is being made by a sub–process, and then follow the ACL on the `inode` of `/home/foobar/.netscape/sub1` to find whether or not it should permit the operation. In this case of course it will fail, since 1025 has no permissions for that file, and the default behavior if no permissions are specified is *deny*.

3.4.4 Controlling CPU Consumption

Sub-process execution time is monitored so that malicious code does not hamper the smooth operation of the system. Sub-processes have no access to the `setpriority(2)` or `setrlimit(2)` system calls, prohibiting them from executing at a higher priority than the parent process and limiting the amount of cpu time they are allocated. Furthermore every time a sub-process *forks*, its allocated cpu time (RLIMIT_CPU) is divided by two, ensuring that it cannot execute forever. There are a number of more elaborate cpu scheduling techniques, but they are beyond the scope of this work.

3.4.5 Controlling Memory Allocation

As with cpu time allocation, the amount of resident memory data of subprocesses is also controlled. This is done by using the RLIMIT_RSS field of the `rlimit` structure in the kernel. We use the same approach as above, reducing the amount of permissible resident memory by half every time a sub-process forks. Since `setrlimit(2)` calls are not permited, we protect against malicious code that attacks the memory subsystem.

3.5 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 and on how the file has arrived on the local system.

For e-mailed files, the senders identity is used to select the sub-user id. That is, if email arrives twice from the same user, any content will be executed using the same 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).

3.6 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. We are currently in the process of implementing support for this case, and we will describe our design. When a sub-process opens another object containing a sub-user id it inherits that id, and the new permissions are those of the intersection of the individual permissions.

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 sub-user id's.

4 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 [15, 17, 23, 8, 7], fault isolation [21] and code verification [18], to operating system-specific permission mechanisms [16] and system call interception [2, 5, 6, 1].

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 [3, 4]. However they offer no protection for the user against programs owned by the user, which may contain errors, Trojan Horses, or viruses.

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.

A different approach relies on the notion of system call interception, and its used by systems like TRON [2], MAPbox [1], Software Wrappers [5] and Janus [6]. 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.

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, [14, 15, 20, 13, 9] the most common example being Java [17]. 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 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* [18] 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.

All the above mechanism deal with the security issues. There is, however, the Quality of Service (QoS) [19, 22] aspect which we do not address directly in this paper. Many attacks will take up system resources but might not harm data. The most common issue is CPU scheduling. There is a lot of research that addresses CPU allocation to ensure fair CPU access for all running processes. Our SubOS uses the simple methods described in Section 3.4 to ensure fairness and a smooth running system. Incorporating any state of the art QoS mechanisms is possible, it is however beyond the scope of this work.

5 Conclusions

We have designed and implemented a process-specific mechanism to contain untrusted objects. 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 user. 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 mechanism and how it strengthens UNIX security.

References

[1] 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.

[2] 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.

[3] Helen Custer. *Inside Windows NT*. Microsoft Press, 1993.

[4] Helen Custer. *Inside the Windows NT File System*. Microsoft Press, 1994.

[5] 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.

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

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

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

[9] 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.

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

[11] http://www.openbsd.org.

[12] 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.

[13] X. Leroy. Le système caml special light: modules et compilation efficace en caml. Research report 2721, INRIA, November 1995.

[14] J. Y. Levy, J. K. Ousterhout, and B. B. Welch. The Safe-Tcl Security Model. In *Proc. of the 1998 USENIX Annual Technical Conference*, June 1998.

[15] 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.

[16] David Mazieres and M. Frans Kasshoek. Secure Applications Need Flexible Operating Systems. In *The 6th Workshop on Hot Topics in Operating Systems*, May 1997.

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

[18] 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.

[19] Jason Nieh and Monica S. Lam. The Design, Implementation and Evaluation of SMART: A Scheduler for Mult imedia Applications. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, October 1997.

[20] J. Tardo and L. Valente. Mobile Agent Security and Telescript. In *Forty-First IEEE Computer Society Conference (COMPCON)*, 1996.

[21] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software–Based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 203–216, December 1993.

[22] Carl A. Waldspurger and William E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proceedings of the First USENIX Symposium on Operating System Design and Implementation*, November 1994.

[23] 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.