A Flexible Containment Mechanism for Executing Untrusted Code

by

DAVID S. PETERSON

B.S. (North Carolina State University) 1996

THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Professor Matt Bishop (Chair)

Professor Raju Pandey

Professor S. Felix Wu

Committee in Charge

2003

A Flexible Containment Mechanism for Executing Untrusted Code

by

David S. Peterson

Abstract

A widely used technique for securing computer systems is to execute programs inside protection domains that enforce established security policies. These containers, often referred to as sandboxes, come in a variety of forms. Although current sandboxing techniques have individual strengths, they also have limitations that reduce the scope of their applicability. In this paper, we give a detailed analysis of the options available to designers of sandboxing mechanisms. As we discuss the tradeoffs of various design choices, we present a sandboxing facility that combines the strengths of a wide variety of design alternatives. Our design provides a set of simple yet powerful primitives that serve as a flexible, general-purpose framework for confining untrusted programs. As we present our work, we compare and contrast it with the work of others and give preliminary results.

Professor Matt Bishop
Thesis Committee Chair

Contents

List of Figures **v**

List of Tables **vi**

1 Introduction **1**

2 Design Alternatives **4**

2.1 Representation and Organization of Privileges . . . 8
2.2 Location of Enforcement Mechanisms . . . 14
2.2.1 Runtime Environment . . . 14
2.2.2 Sandboxed Program . . . 15
2.2.3 User Space . . . 16
2.2.4 OS Kernel . . . 17
2.3 Passive vs. Active Monitoring . . . 19
2.4 Scope of Application: Global vs. Local . . . 22
2.5 Mandatory vs. Discretionary . . . 24
2.6 Inspection and Manipulation of Sandboxes . . . 25
2.7 Static vs. Dynamic . . . 28
2.8 Generic vs. Specific . . . 29
2.9 Transient vs. Persistent . . . 31
2.10 Interaction with Other Security Mechanisms . . . 34

3 Specification of Privileges **37**

3.1 Interval Lists . . . 38
3.2 Sandbox Sets . . . 41
3.3 Signal, `ptrace()`, and IPC Components . . . 44
3.4 File System Component . . . 50
3.5 Network Component . . . 65

3.6 Device Component 65
3.7 System Management Component 66

4 Performance **67**

5 Related Work **71**

6 Conclusions and Future Work **79**

Bibliography **82**

A Availability **86**

List of Figures

2.1 Blocking mechanism 21
2.2 Nested sandboxes 22

3.1 Include operation 39
3.2 Exclude operation 40
3.3 Behavior of sandbox sets 42
3.4 Directory subtree 52
3.5 File system component 53
3.6 Two equivalent file system components 57
3.7 Partial view of file system component F 58
3.8 Normalizing a file system component C 60
3.9 Computing the intersection of components X and Y 63

4.1 Latency of `kill()` executed by sandboxed process 70

List of Tables

4.1 Performance impact of sandboxing mechanism 67

Chapter 1

Introduction

The standard UNIX security model provides a basic level of protection against system penetration. However, this model alone is insufficient for security-critical applications. The security of a standard UNIX system depends on many assumptions. File permissions must be set correctly on a number of programs and configuration files. Network-oriented services must be configured to deny access to sensitive resources. Furthermore, system programs must not contain security holes. To maintain security, one must constantly monitor sites such as CERT and SecurityFocus, install new patches, and hope that holes are patched before an attacker discovers them. Since potentially vulnerable system programs often execute with root privileges, attacks against them often

This research is supported in part by NSF grants CCR-00-82677 and CCR-99-88349.

lead to total system compromise. The typical UNIX system is therefore characterized by many potential weaknesses and is only as secure as its weakest point.

The limitations of the UNIX security model have created much interest in alternate paradigms. This has drawn attention to a wide variety of mechanisms. Examples are capabilities[9], access control lists (ACLs), domain and type enforcement (DTE)[6, 1], and sandboxing mechanisms. Sandboxes are attractive because they provide a centralized means of creating security policies tailored to individual programs and confining the programs so that the policies are enforced. They therefore provide great potential for simplifying system administration, preventing exploitation of security holes in system programs, and safely executing potentially malicious code. Their value as security tools increases as computing environments become more network-centered and execution of downloaded code becomes more common.

A number of methods have been proposed for confining untrusted programs. Although these techniques have individual strengths, they also have limitations that narrow the scope of their applicability. In this paper, we systematically explore the range of options available to designers of sandboxing mechanisms. As we discuss various design choices and their consequences, we present a sandboxing facility that combines the advantages of a number of alternatives. Our sandboxing mechanism is implemented as a system call API that serves as a general-purpose framework for confining untrusted programs. Our goal is to provide primitives that are simple yet powerful enough that system administrators, individual users, and application developers may use them to specify and enforce security policies that are custom-tailored to satisfy their diverse needs.

In the next chapter, we present the design of our sandboxing facility within the context of various design alternatives and the motivations behind them. Chapter 3 provides details of how privileges are represented in our design. In Chapter 4, we give preliminary performance results from a partially completed implementation within the Linux kernel. Chapter 5 contains an overview of related works and how they differ from our design. Finally, we present conclusions and discuss future work in Chapter 6.

Chapter 2

Design Alternatives

The design of a sandboxing mechanism may be viewed from a number of angles. We have identified the following issues:

1. Sandboxes may grant or deny various privileges to the programs that they contain. How are these privileges represented and organized?

2. Where are the mechanisms located that enforce sandbox-imposed restrictions?

3. Are restrictions enforced by passive or active entities[1]?

4. Are sandboxes global entities that enforce systemwide constraints or more localized entities that confine individual programs or perhaps groups of related programs? What criteria are used to group programs into sandboxes?

[1] Active entities are separate processes or threads that monitor the activities of sandboxed programs. Passive entities are variables or data structures maintained by the sandbox that are examined as part of the privilege checking steps that occur when a program attempts some action.

5. Do sandboxes enforce mandatory or discretionary access controls?
6. How are access privileges determined for inspection and manipulation of sandbox configurations?
7. Are sandboxes static or dynamic entities? In other words, are their configurations fixed or subject to change? If sandboxes are reconfigured in response to changing security policies, how do the changes propagate throughout a running system?
8. Are sandboxes generic entities for entire classes of programs, or are they narrowly customized for specific programs?
9. Are sandboxes transient or persistent entities? Do they function as lightweight, disposable containers, or do they maintain relatively static long-term associations with programs and other objects that they may contain?
10. How do sandboxes interact with other security mechanisms?

Before giving detailed consideration to each of these questions, we first give a brief introduction to our sandboxing facility and a few of its properties. This will clarify our subsequent discussion of the design space and where our mechanism stands in relation to each of the above issues. As the discussion progresses, we will present additional aspects of our design and the motivations behind them.

We have developed a kernel-based mechanism that provides a general-purpose system call API for confining untrusted programs. Processes may create their own sandboxes, launch arbitrary programs inside them, and dynamically reconfigure the sandboxes as programs execute inside. Unprivileged processes may safely create and configure sandboxes because our mechanism follows the principle of attenuation of privileges. Specifically, a sandbox can never grant privileges to a program beyond what the program would normally have if it were not executing inside the sandbox. Consider the following example of how our facility might typically be used:

1. A process creates a new sandbox by making an `sbxcreate()` system call. The newly created sandbox is assigned a numeric identifier that is conceptually similar to a filename. The creator receives a numeric handle that is essentially the same as a file descriptor. Initially, only the creator can access the sandbox.

2. The process configures the sandbox using additional system calls.

3. The process forks and the child inherits a copy of the parent's sandbox descriptor.

4. The child applies the sandbox to itself by making an `sbxapply()` system call. This can be done in one of two ways:

   (a) No options are specified when calling `sbxapply()`. On return, the sandbox is applied to the child. The apply operation automatically closes any sandbox descriptors held by the child. The child therefore gives up control of all sandboxes it formerly controlled, including the one that now contains it.

   (b) The `SBX_OPT_APPLY_ON_EXEC` option is passed to `sbxapply()`. The child then performs an `execve()` system call. If `execve()` succeeds, the sandbox is applied to the child and all of its sandbox descriptors are closed. On failure, the sandbox is not applied. Thus the child retains any privileges necessary for error handling.

5. The parent retains full control over the sandbox and may reconfigure it while the child executes inside. The parent may also launch additional programs inside the sandbox. Alternately, it may close its sandbox descriptor, giving up all access rights and eliminating itself as a potential point of attack. The sandbox is now unchangeable by any process, even those with root privileges. Although the child is trapped in the sandbox for the rest of its lifetime, outside processes can still suspend or terminate it. Sandboxes only impose restrictions on the processes they contain. They never place limits on what outside processes can do relative to processes executing within.

6. All of the child's descendants inherit its sandbox. A process may be sandboxed only by applying a sandbox to itself or inheriting its parent's sandbox.

7. There is no explicit destroy operation for sandboxes. The kernel manages their destruction through reference counting.

Now that our sandboxing facility has been introduced, we continue with a discussion of the design space that individually addresses each of the previously mentioned questions.

2.1 Representation and Organization of Privileges

The question of how to represent and organize sandbox-related privileges is open-ended. There are a multitude of potential options, and any attempt to thoroughly discuss every possibility is almost certain to leave out many alternatives. We therefore focus on two key issues: extensibility and expressiveness.

As computer systems evolve to serve new purposes, new features are added to operating systems. A sandboxing mechanism should therefore be easy to extend so that it may enforce security policies governing access to new types of system resources. With this requirement in mind, we have divided system functionality into several categories, each represented by a different component type. As new features are added to operating systems, our mechanism may be extended by creating additional component types. To facilitate their development, we have structured our implementation in a modular fashion. Our current design specifies the following seven types of components:

- *Device component:* Specifies access privileges for devices according to device number.
- *File system component:* Specifies access privileges for files according to directory path.
- *IPC component:* Specifies access privileges for IPC objects such as semaphores, message queues, and shared memory segments.

- *Network component:* Specifies ranges of IP addresses to which sandboxed processes may open connections. Also specifies ranges of ports from which incoming connections may be received.
- *ptrace() component:* Specifies which processes a sandboxed process may `ptrace()`.
- *Signal component:* Specifies processes to which a sandboxed process may send signals.
- *System management component:* Specifies privileges for administrative actions such as rebooting and setting system date/time.

A sandbox's creator specifies allowed privileges by creating components and attaching them to the sandbox. A component may be attached to several sandboxes simultaneously, but a given sandbox may be attached to at most one component of each type at any given instant[2]. This aspect of our design provides a convenient means of sharing related privileges across multiple protection domains and allowing changes in privileges to simultaneously affect multiple protection domains. Separating sets of privileges from their associated sandboxes also facilitates their customization and reuse.

The creator of a sandbox may change the set of attached components or adjust their settings while processes execute inside. When a component is first created, it initially denies all privileges that it governs. The creator must then specify explicitly

[2] Actually, a sandbox has two sets of attachment points for the various component types. The purpose of the second set of attachment points will be described later.

which privileges are allowed. If no component of a particular type is attached to a given sandbox, then all privileges associated with that component type are implicitly denied. Therefore, existing programs that use our mechanism will deny access to new areas of system functionality by default. Since privileges are denied by default, our design exhibits the principle of fail-safe defaults as described by Saltzer and Schroeder[10].

To permit flexible specification of fine-grained security policies, privileges must be specified in a highly expressive manner. With this goal in mind, we divide privileges into two categories: *binary privileges* and *quantitative privileges.* A binary privilege may be assigned one of two possible values: *allow* or *deny.* An example is the ability to read the contents of `/etc/passwd`. A quantitative privilege may be assigned numeric values such as 50 or 100. For example, the total memory allocated to a program might be restricted to a maximum of 4 megabytes.

Our current design only deals with binary privileges. Quantitative privileges address issues regarding denial of service. The addition of features that guard against these types of attacks is an area of future work. We intend to study solutions that others have developed[3, 13] and incorporate them into our design.

One issue for further consideration is the potential for using the sandboxing mechanism itself as a means of implementing a denial of service attack. This could be done by creating large numbers of sandboxes or components until all memory is exhausted. Sandbox-related objects occupy kernel memory, which may not be paged out under most UNIX implementations. The importance of guarding against abuse of the sandboxing mechanism is therefore elevated. However, this type of problem may be easily solved by restricting the numbers of sandbox-related objects that programs may create. Furthermore, existing UNIX systems are already plagued by similar problems[3]. User processes may easily overwhelm the system by creating large numbers of processes, files, and IPC objects, or allocating large amounts of memory and writing to individual pages, forcing page allocation. Even in its current form, our system is therefore unlikely to make matters much worse than they already are.

The two possible values of a binary privilege may be viewed as membership in or exclusion from a set of allowed operations. This insight suggests the following approach: Represent sets of privileges as first-class objects and provide primitives for manipulating them using set-theoretic transformations. Our components are designed to behave in exactly this manner. Specifically, given two components x and y of a given type, we provide the following operations:

- *Create union:* Create a new component z that represents the union of the privileges given by x and y.

- *Create intersection:* Create a new component z that represents the intersection of the privileges given by x and y.

- *Create complement:* Create a new component z that represents the complement of the privileges given by x.

- *Union with self:* Modify x so that it represents the union of y with its prior value.
- *Intersect with self:* Modify x so that it represents the intersection of y with its prior value.
- *Complement self:* Modify x so that it represents the complement of its prior value.

Our set-oriented approach to creating and manipulating privileges associated with protection domains represents a unique perspective. As an example application, consider an employee Bob who initially works in the personnel department of some company and then transfers to the finance department. Let B represent the privileges that Bob's sandbox initially allows. Let P represent the privileges required for Bob's personnel-related duties and let F represent the privileges required for Bob's finance-related duties. The transition between departments may then be accomplished by manipulating Bob's sandbox as follows:

$$B := (B \cap \overline{P}) \cup F$$

Suppose that Bob then starts working on a project that requires collaboration with another employee George. He therefore needs to access some of George's files. Let G represent George's files and let G_C represent a subset of George's files that are confidential and must not be shared with Bob. The necessary sharing may then be allowed by making the following change to Bob's sandbox:

$$B := B \cup (G \cap \overline{G_C})$$

As our discussion continues, we will mention other applications that may benefit from a set-oriented view of privileges. In general, the ability to manipulate components using set operations has several advantages:

- Set operations are very expressive. They allow components to be constructed that satisfy assertions relative to each other given by arbitrary set-theoretic expressions.
- Set theory is well-understood. Therefore, so are relationships among components.
- Set operations provide a means of manipulating privileges that is uniform across all component types. This exemplifies the principle of economy of mechanism presented by Saltzer and Schroeder[10] and is likely to simplify programs that use our sandboxing API.
- Set operations provide a means of answering questions such as "Which privileges are granted to user A or user B but denied to user C?" This information may be useful if we wish to know how much damage user C can inflict if he successfully bribes users A and B. In general, a convenient means of answering such questions allows one to easily understand implications of various sandbox configurations.
- By clarifying relationships between sandbox-associated privileges, set operations provide a means of verifying that security policies are correctly enforced.
- Providing users with simple yet powerful mechanisms often results in the development of new and useful applications. In this respect, our design follows the

UNIX philosophy of giving users building blocks that they can combine to produce customized solutions to a wide variety of problems.

We therefore believe that the inclusion of set-oriented primitives in our model is a prudent design decision.

2.2 Location of Enforcement Mechanisms

Sandboxing mechanisms may be implemented in any of the following locations:

- runtime environment
- sandboxed program
- user space[3]
- OS kernel

We will now consider each of these alternatives, focusing on their advantages and disadvantages.

2.2.1 Runtime Environment

In this arrangement, the sandboxed program executes within a specialized runtime environment that provides complete mediation between the program and underlying

[3]Here, we mean separate from the sandboxed program and any runtime environment in which it may be executing.

system resources. The runtime system can therefore prohibit actions that violate established security policies. A well-known example of this type of sandbox is the Java virtual machine[8]. This option is attractive because it allows security policies to be tailored to the runtime environment. For example, an object-oriented system could restrict access to individual method invocations. Furthermore, protection mechanisms may be very fine-grained. Pointer use may be completely eliminated, or pointer dereferences may be individually validated at runtime. However, this approach is only applicable to programs that execute within a particular runtime environment. It is therefore not suitable as a general-purpose mechanism.

2.2.2 Sandboxed Program

An alternate approach is to embed the sandboxing mechanism within the sandboxed program. Proof-carrying code[7] is an example of this technique. In this scheme, a binary executable contains a mathematically rigorous proof that it satisfies a given security policy. Before the program executes, a verifier checks the correctness of the proof. If the proof is incorrect or does not satisfy the security policy, then the program is denied the privilege to execute. It is also possible to instrument a binary executable with additional machine instructions that verify compliance with a security policy[4]. Both of these types of sandboxes have the advantage of being able to enforce fine-grained security policies at the level of individual machine instructions.

However, the need to modify binary executables makes these techniques inconvenient. A disadvantage of this approach is that application developers, not users, are responsible for defining the security policies. This places the burden of anticipating the security needs of individual users on application developers. It forces developers to adopt a "one size fits all" approach to security that is unlikely to be a good match for the requirements of a diverse set of users. Furthermore, users are unable to modify security policies to satisfy their changing needs. Another disadvantage is that sandboxes of this type are not generally applicable to all types of programs (such as shell scripts, for instance). They are therefore not suitable as general-purpose mechanisms.

2.2.3 User Space

Another option is to implement sandboxes as separate processes that execute in user space. This requires some type of OS-provided mechanism that allows one process to control the execution of another process. Several mechanisms of this variety[16, 2, 14] use the `/proc` process tracing facility of Solaris for system call interception. This type of design is advantageous because it may be easily deployed in existing systems. Binary executables do not require modification, and the mechanism may be applied to arbitrary types of programs such as shell scripts. A disadvantage is that the Solaris process tracing facility is not applicable to setuid programs. If setuid programs were traceable in this manner, an unprivileged user could perform arbitrary operations as root simply by tracing a setuid program and modifying parameters to system calls as they are invoked. This approach adds overhead, since it requires additional processes for monitoring. Furthermore, monitoring requires interprocess context switches, and the monitoring process must typically `fork()` each time the sandboxed process forks.

2.2.4 OS Kernel

The OS kernel is another potential place where sandboxing mechanisms may reside. This location allows placement of privilege checking hooks and other functionality at points deep within the kernel. It therefore provides essentially unlimited options for restricting access to system resources and fundamentally changing how the system as a whole behaves. Furthermore, the strict isolation of the kernel from user space entities is likely to make kernel-resident sandboxing mechanisms less vulnerable to attack. However, kernel modification requires access to source code unless the sandboxing mechanism is implemented as a loadable kernel module (LKM). Another disadvantage is that kernel code is difficult to write and debug, and must be fully trusted. Bugs or design flaws may create systemwide vulnerabilities or cause system crashes. Placing the implementation code for individual sandboxes within the kernel causes the set of all code that must be trusted with full kernel privileges to expand without bound as new sandboxes are defined. Furthermore, users and programs who define the sandboxes must be fully trusted. In contrast, our approach places support within the kernel for only a core set of primitives.

As programs create their own sandboxes, their application-specific code executes in user space without being given any special privileges.

We have chosen to implement our sandboxing mechanism within the OS kernel. The kernel-resident status of our implementation allows us to export a universally accessible system call API that may be applied to both privileged and unprivileged programs, regardless of what language they were written in. Our system call API is designed to be policy-neutral and highly flexible. It provides a minimal set of primitives that are designed to serve a wide variety of purposes. Thus, application-dependent aspects of sandbox manipulation are pushed into user space where they belong. The general-purpose nature of our design mitigates the disadvantages of kernel code being difficult to develop and debug. Furthermore, the increasing pervasiveness of open-source operating systems such as Linux and FreeBSD mitigates the disadvantage that access to kernel source code is required.

An area of future work is to modify our implementation so that it interoperates with the Linux security module framework[11] now being developed within the Linux community. This will allow users to add our security mechanism as a module to existing kernels, therefore eliminating the need to patch an existing kernel source tree and do a complete kernel rebuild. It will also eliminate the need to maintain our code as a patch that constantly requires updating as new kernels become available.

2.3 Passive vs. Active Monitoring

Sandbox-imposed restrictions may be enforced by passive data structures that are examined whenever a program attempts to perform some operation. For example, the kernel's implementation of the `open()` system call might be modified so that sandbox-related data structures are consulted before `open()` is allowed to proceed. We refer to this as passive monitoring. Alternately, restrictions may be enforced by separate processes or threads that monitor programs as they execute. We refer to this as active monitoring. An advantage of active monitoring is its flexibility. Monitoring processes are not restricted to making policy decisions based on relatively static data structures. Instead, they may implement security policies defined by complex state machines. This provides a means of changing the set of granted privileges in response to actions attempted by sandboxed programs, allowing for richer security policy definitions.

The disadvantage of active monitoring is the high overhead it requires. Monitoring processes must be created and individual privilege checks require interprocess context switches. Furthermore, most designs require the monitoring process to `fork()` each time a sandboxed process forks.

To address this design issue, we have developed a novel mechanism that allows monitoring to be purely passive, purely active, or anywhere in between. Thus, programs may benefit from the best aspects of both alternatives. We achieve these benefits through a mechanism that allows privileges to be determined interactively at runtime.

Specifically, a sandbox may be configured so that attempting certain actions will cause a sandboxed process to block instead of being immediately denied the privilege to perform the action. When a process blocks in this manner, an event is generated and placed in the event queue of the sandbox where the blocking occurred. A process that has ownership over the sandbox uses the `sbxwait()` system call to wait for and obtain events. An event may be examined to determine which process generated it and what action was attempted. The `sbxdecide()` system call is then used to unblock the process that triggered the event and decide whether to allow the attempted action.

Our design permits application of the blocking mechanism in a fine-grained manner. Figure 2.1 illustrates how this works. Each sandbox has two sets of attachment points for the various component types. Sandbox S has device components D_1 and D_2 attached at points d_1 and d_2. File system component F_1 is attached at point f_1. Process p controls sandbox S while q executes inside. When q attempts to access a device, the sandboxing mechanism first examines D_1. If D_1 allows the required privilege then the operation will succeed[4]. Otherwise, D_2 is examined. If D_2 allows the privilege, then q blocks and p decides whether to allow the operation. If D_2 denies the required privilege, then the operation will fail. If q attempts to access some file, the sandboxing mechanism examines component F_1. If F_1 allows the required privilege, then the operation is allowed. Otherwise, the operation is immediately denied, since no component is attached at point

[4] This assumes that file permission bits and other applicable security mechanisms also allow the operation.

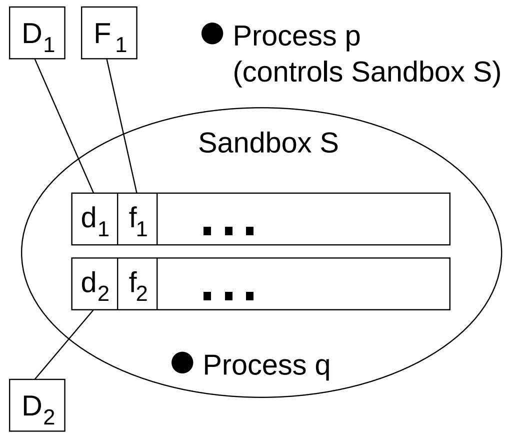

Figure 2.1: Blocking mechanism

f_2.

A potential use of this feature is intrusion detection. For example, a telnet daemon could place a user's login shell inside a sandbox and use the blocking feature to monitor aberrant behavior. If such behavior is detected, the system can make fine-grained adjustments to the set of actions that it monitors. In response to suspicious behavior, the system may tighten sandbox-imposed constraints, or perhaps perform other actions such as notifying a system administrator. The status of our design as a system call API facilitates interfacing with existing auditing and intrusion detection systems.

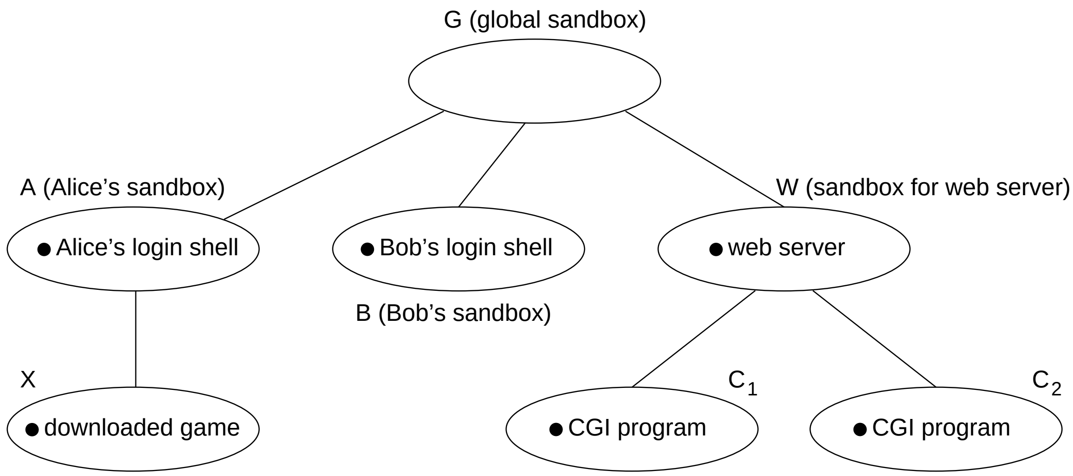

Figure 2.2: Nested sandboxes

2.4 Scope of Application: Global vs. Local

In principle, sandboxes may be used to confine individual users, groups of users, individual programs, or perhaps groups of programs that cooperate to serve common purposes. One might even imagine a global sandbox that enforces certain restrictions on all programs. These alternatives raise the question of where sandboxes should be deployed on the spectrum from global to local. Also, what criteria should be used for grouping programs into sandboxes?

We believe that there is no single best answer to these questions. Therefore our design allows system administrators, users, and application developers to create sandboxes that enforce security policies at any level of granularity. To permit simultaneous enforcement of access controls at multiple levels, our design provides the ability to create hierarchically nested sandboxes, as shown in Figure 2.2.

In this example, sandbox G is a global sandbox that contains all processes. G enforces global policies such as the restriction that no process should be able to modify system programs in locations such as `/bin` and `/usr/bin`. At system startup time, `/sbin/init` creates G and applies G to itself before it forks any child processes. To override the restrictions imposed by G, an administrator with physical access to the system console must reboot the system with a kernel in which sandboxing functionality has been disabled.

At a more localized level, programs such as telnet daemons, ftp daemons, and the standard login program may be modified to place restrictions on individual users. Sandboxes A and B restrict the login shells of users Alice and Bob in this manner.

Users may selectively delegate their privileges by creating sandboxes for individual applications. For instance, user Alice has downloaded a video game from an untrusted source. To protect against Trojan horses, she executes the program inside sandbox X.

Finally, an application program that is aware of the sandboxing mechanism may use it as a flexible means of dropping privileges when performing sensitive operations. The web server executing in sandbox W uses our mechanism in this manner by executing CGI programs in sandboxes C_1 and C_2.

If the blocking mechanism is used in combination with nested sandboxes, an attempted action by a sandboxed process may cause it to block sequentially at multiple levels. For instance, if the downloaded game in sandbox X attempts to open some file, the privilege checking operation performed at sandbox X may cause it to block. If a process in sandbox A decides to allow the action, then a privilege check will be performed at sandbox A. Depending on how A is configured, this may also cause the process to block, providing an opportunity for a process in sandbox G to allow or deny the action. The same behavior could also take place at sandbox G if it were configured appropriately, although this would require some process outside G to be responsible for monitoring G. In practice, we believe that sandboxes will rarely be nested at depths of more than three or four levels. Therefore the overhead required to perform privilege checks at multiple levels should be reasonably low.

2.5 Mandatory vs. Discretionary

Security policies may be enforced by either mandatory or discretionary access controls. Mandatory access controls are useful because they are based on systemwide rules beyond the control of individual users. They therefore provide a high degree of assurance that systemwide security policies are not violated. Discretionary access controls are useful because they allow individual users to define their own security policies. These two alternatives raise the question of whether sandboxes should be mandatory or discretionary in nature.

Our design provides both options. One means of providing mandatory access controls is to place `/sbin/init` in a sandbox at system startup time. Additionally, sandboxes may enforce mandatory access controls at the level of individual users. Since our mechanism follows the principle of attenuation of privileges, unprivileged users may employ it to create discretionary sandboxes.

As future work, we intend to add a mechanism that allows transitions between sandboxes when certain programs are executed. This would make sandboxes more similar to the domains provided by DTE[6, 1]. However, the use of components to define privileges granted to domains is a different approach from using types. Using our mechanism, a core set of components may be defined that serve the same purpose as types. Additional types can be derived using set-theoretic transformations. Permitting dynamic creation of types at runtime may also be useful. For instance, executing a certain program might cause creation of a new type that is a function of the user's previous type and possibly other variables.

2.6 Inspection and Manipulation of Sandboxes

An effective sandboxing mechanism must provide some means of guarding access to sandbox-related objects. In this discussion, the term *object* refers to a sandbox, component, or pool[5]. If anyone may reconfigure a sandbox, then the restrictions it im-

[5] Pools are collections of sandboxes. They will be described in more depth later.

poses are easily circumvented. Furthermore, one might create a sandbox that denies access to some resource whose existence must remain hidden. Allowing anyone to examine a sandbox configuration may therefore cause unacceptable leakage of information.

The question of how access to sandboxes should be governed is open-ended and depends on the details of the mechanism being considered. We have taken a conservative approach in which access is strictly limited. A descriptor with read privilege is required for examining the configuration of an object. Likewise, a descriptor with write privilege is required for calling `sbxwait()` on a sandbox or modifying an object. Descriptors may be obtained only as follows:

- The creator of an object receives a descriptor with both read and write privileges for the new object.
- When a process forks, the child inherits all of the parent's descriptors along with their associated privileges.
- If a process inside a sandbox creates an object, it may specify that a link is created for the new object. Other processes in the same sandbox may then use the `sbxopen()` system call to open descriptors for the new object. This is analogous to accessing files with the `open()` system call. Processes inside a given sandbox may therefore have shared access to child objects.
- There is only one circumstance in which processes not within the immediate bound-

aries of a given sandbox may open descriptors for its child objects. When creating a component, a process may label it as public. In this case, processes in descendant sandboxes may open descriptors for the component with read-only access.

Our design provides a system call for dropping read and write privileges associated with descriptors. An object that is linked may also be unlinked, or the read and write privileges associated with the link may be dropped individually. Thus, access privileges may be irreversibly dropped in order to eliminate potential points of attack. We may eventually consider extending our model to allow more flexible specification of privileges. One possibility is to define a new type of component that controls access to the sandboxes and components themselves. Although there is a certain elegance in this approach, it creates additional complexity that may be undesirable.

We made a deliberate effort to define our system call API in a manner that follows the design philosophy of existing UNIX system calls. Our use of descriptors, permission bits, and inheritance of sandbox-related state information by child processes provides a familiar, comfortable programming paradigm for veteran UNIX programmers. The ability to `fork()` a child process and then have the child perform an `sbxapply()` system call fits well with the common practice of forking a child which then performs an `execve()` system call. In these respects, our system call API follows the design philosophy of UNIX operating systems and blends well with existing UNIX system calls. However, our design should be relatively easy to adapt to other types of operating systems.

2.7 Static vs. Dynamic

Security policy enforcement mechanisms may be static or dynamic in nature. If the policy seldom changes, then a static mechanism is best because it excludes the possibility of unauthorized tampering. However, a dynamic mechanism may be preferable if the policy changes frequently. Our mechanism provides both options. Sandboxes and components are dynamic by default, but dropping write privileges causes them to become static.

When adjustments to security mechanisms are made, they should ideally have an immediate effect on all relevant aspects of system behavior. Our implementation of nested sandboxes was designed with this consideration in mind. Since privilege checks are done individually at each level, reconfiguration of a sandbox immediately affects all of its descendants.

File descriptors represent a similar area of concern. For instance, suppose that a process opens some file and its sandbox is then adjusted so that access to the file is denied. Under our current implementation, the process may continue to access the file through its previously opened file descriptor. Adding the ability to revoke privileges stored in file descriptors would be relatively easy. This may be done by attaching sandbox-related tags to file descriptors and performing additional privilege checks during `read()` and `write()` system calls. Although this option has little value for guarding confidentiality, it may still be useful as a damage control mechanism for protecting data integrity. We may therefore eventually implement this feature.

2.8 Generic vs. Specific

When specifying privileges for sandboxed programs, two alternative strategies are possible. One option is to grant privileges that are custom-tailored to individual programs. This approach is advantageous because it follows the principle of least privilege. Since each program is only allowed to perform actions that are necessary for proper functioning, the potential for abuse of privileges decreases. However, creating specialized policies for many applications is labor-intensive. It is also error-prone, since required privileges may be hard to predict in advance. Applications may therefore fail unexpectedly if their sandboxes constrain them too tightly.

To address these problems, one may create generic protection domains for groups of programs with similar behavior. A sandboxing mechanism known as MAPbox[2] employs this technique. Although this approach may simplify sandbox construction, appropriate behavior classes may be difficult to create. If privileges are defined too conservatively, then the scope of applicability of each behavior class becomes unacceptably narrow. However, loosely specified behavior classes stray from the principle of least privilege. The implementors of MAPbox found that this tension between loosely and tightly specified behavior classes creates substantial practical difficulties when defining sandboxes for various programs.

Some application-specific differences among programs within a behavior class may be handled by a technique that MAPbox refers to as parameterization. For instance, a group of network-oriented services may function in a similar manner but differ in the ports from which they receive incoming connections. In this case, their behavior class may take a port number as a parameter. Using our facility, behavior classes could potentially be represented as groups of components. Set operations could then be employed to create customized versions for individual programs in a manner somewhat similar to parameterization. This approach provides a level of convenience, expressive power, and clarity of relationships among protection domains that goes beyond the capabilities that MAPbox provides.

Alternately, our blocking mechanism may be used to create custom-tailored sandboxes for individual applications. For example, consider the following sequence of events:

1. A user executes a program inside a sandbox. The user has no way of knowing ahead of time what privileges it will require. Therefore the sandbox is made initially very restrictive.

2. When the program attempts to perform a denied action, it blocks and the user learns exactly what happened. The user can then decide to allow or deny the action. To allow all future operations of this type, the user may adjust the appropriate component.

3. When the sandboxed program terminates, the user may save the final sandbox configuration to be reused when executing the program in the future.

This technique makes sandbox construction less labor-intensive, since privileges may be granted interactively. Attempted actions that might otherwise cause a sandboxed program to fail may therefore be allowed at the time they are attempted. This eliminates the need to execute the program multiple times, making incremental changes to its sandbox after each execution. Furthermore, programs may be constrained very tightly without adverse consequences. Additional privileges may be granted at runtime as they are needed, providing users with a means of overcoming the difficulties discovered by the designers of MAPbox.

2.9 Transient vs. Persistent

Sandboxes may be implemented as lightweight, disposable containers or as persistent entities that maintain relatively static, long-term associations with files that they contain. Our current design only provides transient sandboxes. We chose this option because they require substantially less implementation effort than persistent sandboxes. Implementing sandboxes as persistent entities is likely to require modification of the internal structure of multiple types of existing file systems. This would greatly increase the level of invasiveness of the kernel changes required by our mechanism. It is likely to create code maintenance problems, cause incompatibility with file systems that users have already deployed, and require additional kernel modifications as new types of file systems are created. Furthermore, the implementors of the Linux security module API have decided to omit support for associating persistent security attributes with individual files[11]. Therefore, adding support for persistent sandboxes would prevent our system from being implemented as a Linux security module. For these reasons, we are unlikely to add support for persistent sandboxes.

WindowBox[5], a sandboxing system implemented within the Windows NT kernel, is a design in which sandboxes are persistent entities. It consists of a set of desktops that are completely separate from each other and from the rest of the system. Users may give some desktops more privileges than others. They may also place individual programs and other files within a given desktop. The association between a file and its desktop persists until the user either deletes the file or moves it to a different desktop. This feature is useful because a given program is automatically confined to its desktop whenever the user executes it. Therefore, the security policy associated with the desktop is consistently enforced. Associations between files and their desktops also provide an alternate means of defining privileges. Specifically, access may be granted because a file resides in the same desktop as the program attempting to open it.

It is important to note that the status of Windows as a closed, proprietary system largely eliminates the above-mentioned maintenance problems associated with persistent sandboxes. However, our design is more applicable to open-source operating systems, which are rapidly increasing in popularity.

A potential advantage of defining sandboxes as transient entities is that they may be efficiently discarded when no longer needed. Our design provides a feature that eliminates unnecessary overhead for creating and destroying sandboxes. With this option, a server may create pools of sandboxes for different types of client connections. The server does the following for each client connection:

1. The server forks a child process. The child inherits the parent's descriptors for the sandbox pools that the server created.

2. The child makes an `sbxapply()` system call, passing in a descriptor for the appropriate pool. If the pool is not empty, this causes a sandbox to be removed from the pool. Otherwise, a new sandbox is created and associated with the pool. The newly obtained sandbox is applied to the child, which then handles the client request.

3. When the child dies, the reference count on its sandbox drops to zero. Instead of being destroyed, the sandbox is returned to the pool for later reuse.

Creation of a sandbox pool requires specification of a maximum capacity. If the pool becomes full, additional sandboxes will be destroyed instead of being returned to it. A pool's creator may adjust its capacity value, find out how many sandboxes the pool contains at a given instant, or make adjustments to the current number of sandboxes in the pool.

2.10 Interaction with Other Security Mechanisms

Our facility is designed to be implemented within existing systems. It must therefore peacefully coexist with other security mechanisms. This consideration may be viewed from the following two perspectives:

1. Can other mechanisms override the denial of a privilege by a sandbox?

2. If a sandbox grants a given privilege, can other mechanisms override this decision?

The answer to the first question is "no." In particular, root has no special privileges that allow sandbox-imposed constraints to be bypassed. This property enhances the security of our mechanism. It also permits construction of sandboxes that confine root programs to a subset of the privileges that they normally have. The answer to the second question is "yes." This property allows sandboxes to coexist with other mechanisms without compromising their effectiveness.

Summary

In summary, we have implemented our sandboxing mechanism within the OS kernel. The status of our design as a system call API allows us to provide a universally accessible facility for confinement of untrusted code. Our design provides a minimal set of primitives that permit a wide scope of applicability. Application-dependent aspects of sandbox creation and manipulation therefore reside in user space while the core implementation benefits from the strict isolation of the kernel from user space entities. The location of our mechanism within the OS kernel allows us to exercise essentially unlimited control over how the system as a whole behaves. Root programs may be confined to only a subset of the privileges normally assigned to the superuser while unprivileged programs may safely employ our mechanism without requiring any special privileges.

Components group privileges into related categories, allowing for easy extensibility as new features are added to operating systems. The components allow flexible sharing of privileges among multiple protection domains. They provide a means of representing sets of privileges as first-class objects and deriving new protection domains from existing components using set-theoretic primitives. By default, components may be manipulated dynamically at runtime while programs execute within the sandboxes that they are attached to. However, components may be turned into static entities by dropping their associated write privileges. This is useful for enforcing security policies that rarely change since it essentially eliminates the possibility of unauthorized tampering.

Our blocking mechanism is useful for specifying security policies in which the set of allowed privileges changes in response to actions attempted by sandboxed programs. In this manner, policies specified by complex state machines as opposed to relatively static data structures may be implemented. The blocking mechanism may be used as a means of tightly constraining the actions of untrusted programs when required privileges are not known in advance. Furthermore, it serves as a useful tool for development of auditing and intrusion detection systems.

The ability to create hierarchically nested sandboxes provides a means of simultaneously enforce security policies at multiple levels of granularity. Since privilege checks are performed sequentially at each level, nested sandboxes may be defined and reconfigured independently by their creators. Changes made to the configuration of a given sandbox immediately affect all sandboxes nested beneath it.

We have made a deliberate effort to design our system call API in a manner that is consistent with the design philosophy of UNIX operating systems. Familiar concepts such as descriptors, permission bits, and inheritance of sandbox-related state by child processes fit well with the standard UNIX model and provide veteran UNIX programmers with a familiar programming paradigm.

Chapter 3

Specification of Privileges

We now present the details of how privileges are represented in our design. Although the various component types have individual differences, several common elements are shared among them. One shared feature is support for the set operations of intersection, union, and complement. Additionally, the components employ the following two common mechanisms:

- *Interval lists* allow specification of intervals of values over a fixed range. For instance, we could use an interval list to represent all integers between 10 and 100, the value 250, and all integers between 400 and 500. The components use this data structure in several places.

- *Sandbox sets* specify privileges that allow sandboxed processes to perform actions relative to other processes. The ability to send signals is an example of this type of privilege.

These two shared building blocks simplify the implementation of the components that use them. They also facilitate the construction of new component types. Next, we give a more detailed presentation of their design. This is followed by descriptions of how the individual component types are constructed.

Once sandbox sets have been introduced, the descriptions of signal, `ptrace()`, and IPC components are relatively straightforward since these component types are implemented as sandbox sets. However, there are some details concerning process groups that influence the semantics of our system call API and the implementation of signal-related privilege checks. We give a detailed presentation of these details in our description of signal components. Once interval lists have been introduced, the descriptions of network and device components become trivial and are therefore very short. Our description of system management components is also brief due to the simplicity of their design. However, the design of file system components is rather complex and requires considerable explanation. A large part of this chapter is therefore devoted to their design and behavior.

3.1 Interval Lists

Interval lists provide a convenient way of specifying and manipulating sets of unsigned integers. They support the following operations:

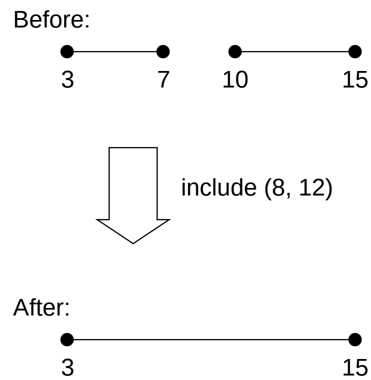

Figure 3.1: Include operation

- *Include:* Figure 3.1 illustrates the include operation. In this example, an interval list initially specifies the intervals $\{(3,7),(10,15)\}$. The interval $(8,12)$ is then included. This produces the interval list $\{(3,15)\}$. Notice that this result is obtained rather than $\{(3,7),(8,15)\}$ or $\{(3,7),(8,12),(10,15)\}$. Interval lists always merge intervals together so that no two intervals are overlapping or immediately adjacent to each other. This yields the simplest possible representation.

- *Exclude:* Figure 3.2 illustrates the exclude operation. In this example, we start with the interval list $\{(5,7),(9,9),(11,15)\}$. The interval $(6,12)$ is then excluded. This produces the interval list $\{(5,5),(13,15)\}$.

- *Intersection:* This operation takes two interval lists as operands and produces a new interval list representing the intersection of the sets of integers they specify. The intervals contained in the result are all nonoverlapping and separated by at

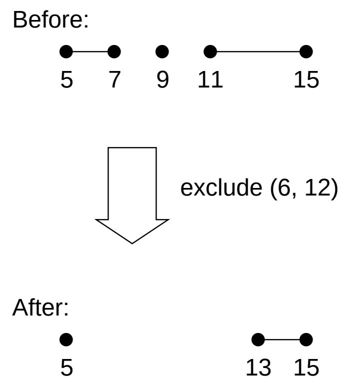

Figure 3.2: Exclude operation

least one integer value.

- *Union:* This operation is similar to intersection, except that the union is computed.

- *Complement:* This operation takes an interval list and produces its complement. For instance, the complement of $\{(5, 10)\}$ is $\{(0, 4), (11, \text{UINT\_MAX})\}$.

- *Query point:* This operation takes an integer as a parameter and returns a Boolean value indicating whether any interval in the list contains it.

We will also provide a mechanism for iterating through an interval list and examining its contents, although this has not yet been implemented.

3.2 Sandbox Sets

Some privileges govern what a process may do relative to other processes. For example, we may wish to allow a sandboxed process to send signals to some processes but not others. One way of accomplishing this is to specify privileges individually for every existing process. However, this is clearly not practical. Therefore processes must be grouped together in some manner. Our design employs sandboxes as the basic unit of organization for assigning privileges relative to processes. For example, signal components specify sets of sandboxes containing processes that may be signaled. We chose sandboxes as the unit of grouping because this is the simplest option. Introducing some other abstraction would create additional complexity without any clear benefits.

Figure 3.3 illustrates how sandbox sets operate. Signal components C_1, C_2, and C_3 are attached to sandboxes S_1, S_2, and S_5 respectively. C_1 allows S_1 to signal processes in S_9, C_2 allows S_2 to signal processes in S_3 and S_9, and C_3 allows S_5 to signal processes in S_6 and S_7. Process p created and initialized S_1, S_9, and C_1. The following rules govern the behavior of components implemented using sandbox sets:

- A process in a given sandbox is always allowed to access other processes in its own sandbox or any descendant sandboxes. For example, a process in S_2 may signal any process in S_2, S_5, S_6, or S_7 regardless of how C_2 is configured.

- If a component grants access to a given sandbox, then access is also granted to all of

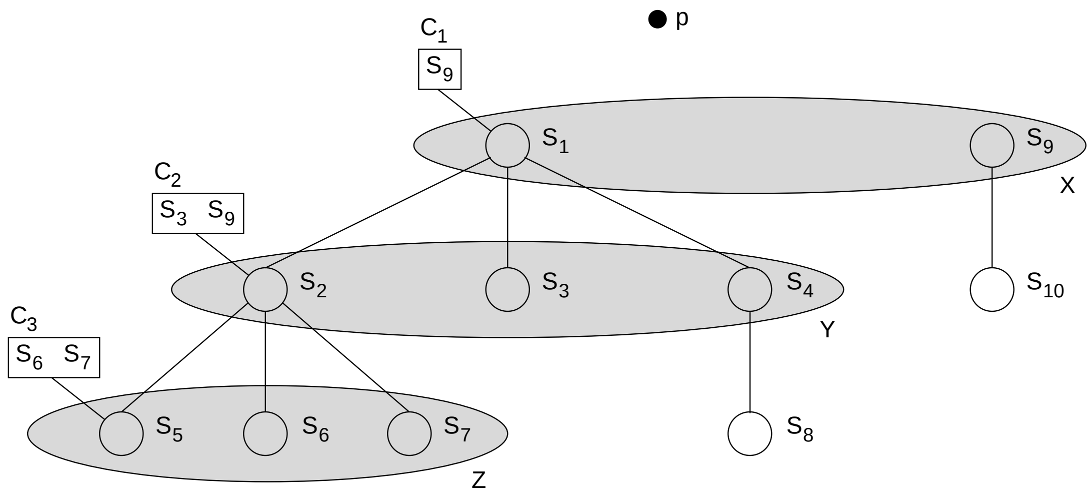

Figure 3.3: Behavior of sandbox sets

the sandbox's descendants. For instance, processes in S_1 can signal processes in S_{10} since C_1 grants access to S_9 and S_{10} is a descendant of S_9. The motivation for this behavior may be understood by considering the viewpoint of process p. Clearly, p is aware of the existence of S_9. However, p can not in general be expected to keep track of actions, such as creating child sandboxes, that may be performed by processes in S_9. All p cares about is that processes in S_1 are granted access to all processes that S_9 governs. Thus this rule allows processes to manipulate components without needing to be aware of details that are outside their scope of concern.

- A process in a given sandbox may delegate to child sandboxes any access rights to other sandboxes that it possesses. For example, S_1 has adjusted C_2 so that its privilege for signaling processes in S_9 is passed down to S_2. Similarly, S_2 may adjust C_3 so that processes in S_5 can signal processes in S_9. However, S_2 may not adjust C_3 so that processes in S_5 are granted access to S_1, S_2, or S_4. This is because S_2 does not have access to S_1, S_2, or S_4. In general, any sandboxes in shaded area X could potentially appear in C_1. However, C_1 can not specify S_{10} directly because S_{10} is outside C_1's scope of concern. Likewise, any sandboxes in shaded areas X or Y but not Z could potentially appear in C_2.

- All processes that are not in any sandbox are grouped together as if they are all inside a common sandbox that imposes no restrictions. This can be thought of as the "null sandbox", and may be specified in a sandbox set just like any other sandbox.

- It is possible to compute the complement of a sandbox set. For instance, the complement of the set given by C_3 would be a set that grants access to all sandboxes (including the null sandbox) except S_6 and S_7. Likewise, intersections and unions of sandbox sets may be computed.

Sandbox sets are implemented internally using a global matrix. Columns represent sandboxes and rows represent components that are implemented as sandbox sets. Adjusting a component C so that it grants access to a sandbox S is accomplished by adding an entry to the matrix at position (C, S). When a component is destroyed, its corresponding row is removed from the matrix. Likewise, destruction of a sandbox results in the removal of its associated column. This ensures that components do not refer to sandboxes that no longer exist.

Our implementation includes a kernel thread that is responsible for disposing of matrix rows and columns associated with expired sandboxes and components. This allows removal of rows and columns to be batched together for efficiency purposes and decreases the frequency with which the global matrix lock must be acquired and released. Furthermore, the kernel thread may sleep in an uninterruptible manner when attempting to acquire the lock. This eliminates the problem of failed lock acquisition due to interruption by a signal. Our implementation therefore adheres to the well-known design principle that operations involving destruction of objects or release of resources should be failsafe. Since user processes are not required to sleep in an uninterruptible manner, difficulties associated with killing processes under heavy lock contention are avoided.

3.3 Signal, `ptrace()`, and IPC Components

Signal components specify processes to which a sandboxed process may send signals. Likewise, `ptrace()` components specify which processes a sandboxed process may `ptrace()`. Both of these component types are implemented as sandbox sets. IPC components specify which IPC objects[1] a sandboxed process may access. If a process

[1] semaphores, message queues, and shared memory segments

executing in sandbox S creates an IPC object X, then S is viewed as owning X. Suppose that S has a parent sandbox T, and S is subsequently destroyed while X still exists. In this case, ownership of X is transferred to T. If S has no parent, then ownership of X is transferred to the null sandbox when S is destroyed. Given this notion of ownership, sandbox sets may be used to implement IPC components. For instance, suppose that the components shown in Figure 3.3 are IPC components. Then C_1 allows processes in S_1 to access IPC objects owned by S_9 or S_{10}, since S_{10} is a descendant of S_9.

Associating ownership of IPC objects with the sandboxes that contain their creators makes sense because of the manner in which IPC objects are typically used by server processes and their clients. A server daemon will typically create one or more IPC objects and make them known to clients who wish to initiate communication. Therefore, granting access to a given IPC object may be viewed conceptually as granting access to the server that created it. This fits well with our treatment of signals and `ptrace()`, which also represent interactions among processes.

For applications in which previously created IPC objects are accessed by mutually cooperative programs, the objects may be created by a program that plays a supervisory role. This program may choose which sandboxes to place the objects within and configure the sandboxes of the cooperating entities appropriately.

We encountered one difficulty when implementing the privilege checks associated with sending signals. This involves the use of the `kill()` system call to signal all processes in a process group. Our implementation is influenced by two design alternatives:

A. Allow a given process group to simultaneously contain processes executing within separate sandboxes.

B. Enforce the requirement that at any given instant, all processes within a process group must belong to the same sandbox.

Option A implies that when a sandboxed process attempts to signal all members of a process group, multiple privilege checks must be performed as signals are sent to individual processes. This could potentially add much overhead for large process groups. The overhead is magnified if attempts to send signals to individual processes cause invocation of the blocking mechanism. The possibility of a `kill()` system call only partially succeeding complicates its semantics. It raises the issue of what value should be returned to the user in this case. It also raises the question of how the caller should detect which processes in the group were prevented from being signaled.

Option B allows the following implementation of signal-related privilege checks:

1. Search the process table for a task within the given process group. Hashing may be used here to make this an O(1) operation.

2. Once such a process is found, see which sandbox it belongs to. Then perform a privilege check to determine whether the currently executing process is allowed to

signal processes in this sandbox. If the privilege check fails, then return a value indicating that the `kill()` system call was denied.

3. Assuming that the above privilege check succeeds, acquire the tasklist lock that protects the Linux process table. Then search the process table for tasks that belong to the given process group. Hashing can be used here to speed the search. When the first such task is found, see if it belongs to the same process group as the process identified in step 1. If so, send signals to this process and all other processes in the group. Then release the tasklist lock and return a value indicating success. Otherwise release the tasklist lock and return a value indicating that the operation should be reattempted.

Option B has the advantages of being potentially more efficient than option A and not substantially altering the semantics of `kill()`. However, B requires reworking the code that implements process groups to enforce the requirement that B imposes. We chose option B because we believe that this is a small price to pay for the benefits it provides over option A. Furthermore, our addition of a hash table that organizes processes according to process group is a better choice than the default Linux implementation, which steps through the entire process table when signaling a process group. Under our implementation, we step directly to the hash bucket corresponding to a process group and find its associated bucket item. The bucket item maintains a linked list of all processes in the group, which are then signaled sequentially.

The requirement imposed by option B above affects the semantics of the `sbxapply()` system call. To ensure that two members of a given process group never simultaneously inhabit different sandboxes, two alternatives are provided for invoking `sbxapply()`.

The default semantics are as follows: At the instant when `sbxapply()` is invoked, the caller must not be a process group leader. Otherwise the system call will fail. This implies that the process group ID (PGID) of the caller must be different from its process ID (PID) and that no process group currently exists with a PGID equal to the PID of the caller. On invocation of `sbxapply()`, the PGID of the caller is set to its PID. The caller therefore becomes the leader of a new process group and the invariant specified by option B above is preserved. This is the exact same type of behavior employed by the `setsid()` system call to ensure that no process group may simultaneously contain members that belong to different sessions.

If the `SBX_OPT_APPLY_ON_EXEC` option is specified, then the kernel verifies at the time of invocation that the caller is not a process group leader. Assuming that this condition holds, the kernel then places the PGID value equal to the PID of the caller in a reserved state. Any attempt to set the PGID of the caller to its PID will then fail. This ensures that no new process group may be created with a PGID equal to the PID of the caller. When the caller then performs an `execve()` system call, the PGID value equal to the PID of the caller is unreserved and the caller becomes the leader of a new process group.

The approach described above has one disadvantage. Suppose that one wishes to implement a command line utility for executing shell commands inside their own sandboxes. Assuming that the name of the command is `sbx`, it might be used as follows to execute the program `myprog` inside a sandbox specified by `mysbx`:

```
$ sbx -s mysbx myprog
```

The `sbx` program might be implemented as follows:

1. A new sandbox is created and configured appropriately.

2. The `sbxapply()` system call is invoked with the `SBX_OPT_APPLY_ON_EXEC` option specified.

3. An `execve()` of `myprog` is performed. This causes the `sbx` program to be replaced by `myprog`, which then executes within the sandbox specified by `mysbx`.

There is a problem with this implementation strategy. When the shell forks a process for the `sbx` program, it makes this process the leader of a new process group. The call to `sbxapply()` by the `sbx` program will therefore fail. To work around this problem, the `sbx` program could `fork()` a child and have the child execute `myprog` inside a sandbox. Since the child is not a process group leader, its call to `sbxapply()` will succeed. However, this workaround is not totally adequate. If the `sbx` program exits immediately after forking its child, then the shell will `wait()` for the `sbx` program and then immediately give the user a new shell prompt instead of first waiting for `myprog` to finish. To prevent this behavior, the `sbx` program could `wait()` for `myprog` before exiting. However, this is somewhat inelegant since it requires keeping an extra process alive while `myprog` executes. Killing this extra process prematurely will cause the shell to give the user another shell prompt before `myprog` finishes.

To overcome these difficulties, alternate semantics are provided for the `sbxapply()` system call. If the `SBX_OPT_PRESERVE_PGID` option is specified, then the system call simply checks at the time of invocation that the caller is the only member of its process group. If this check fails, then the system call returns an error. If the `SBX_OPT_APPLY_ON_EXEC` option is specified in combination with `SBX_OPT_PRESERVE_PGID`, then the check is performed when `execve()` is invoked. If the check fails, then `execve()` returns an error.

3.4 File System Component

File system components specify file-related privileges. They are represented as trees of directory paths with labels that specify privileges at each node. The following types of privileges are defined:

- *r:* For a normal file, this privilege allows the file to be opened for reading. For a directory, it allows the directory contents to be listed.
- *w:* For a normal file, this privilege allows the file to be opened for writing. For a directory, it allows files in the directory to be created, unlinked, or renamed.

- *x:* For a normal file, this privilege allows the file to be executed. For a directory, this privilege has no meaning.
- *p:* For both normal files and directories, this privilege allows permission-related settings to be changed. Specifically, it allows use of `chmod()`, `chown()`, and `chgrp()`.
- *t:* For both normal files and directories, this privilege allows changing access and modification times using `utime()`.
- *s:* For a directory, this privilege allows opening files in the directory, accessing subdirectories, and moving into the directory using `chdir()`. For a normal file, this privilege has no meaning.

For each of these privileges, a set of three labels is attached to each node. Figure 3.4 illustrates the meanings of the labels. Set S consists of the entire subtree rooted at directory n_1. Set T consists of n_1 and all of its children. Set U consists only of n_1. Given these definitions, the three labels attached to n_1 for a given privilege are defined as follows:

- *self:* This label represents set U (consisting of only n_1).
- *children:* This label represents the set of nodes defined by $T - U$ (n_2 and n_3 in the figure).
- *grandchild subtrees:* This label represents the set of nodes defined by $S - T$ (n_4, n_5, n_6, and all of their descendants).

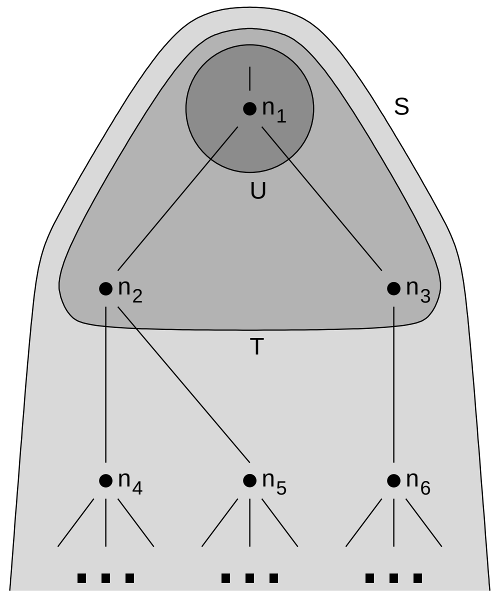

Figure 3.4: Directory subtree

Each label may be assigned one of three values: *allow*, *deny*, or *unspecified*. Labels are ordered according to two simple precedence rules. Labels with higher precedence override the settings of labels with lower precedence. The rules are as follows:

- A label at a node has higher precedence than labels at any of its ancestors.
- There is no ordering among the three labels at a node. This is because the labels represent disjoint sets of nodes.

A label of *unspecified* on a node imposes no particular setting on it or its descendants. Settings are instead determined by labels of higher precedence. A file system component consisting of an empty tree denies all file-related privileges.

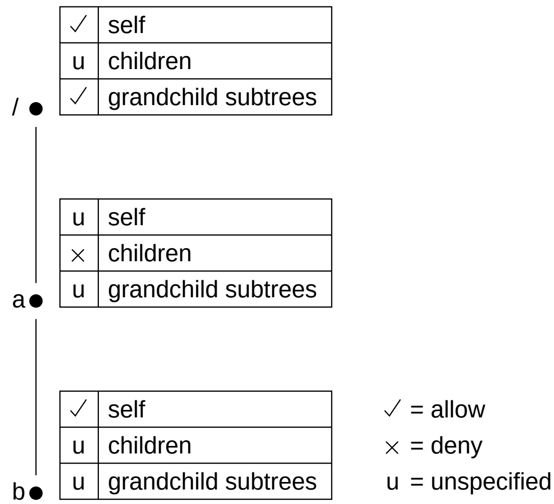

Figure 3.5: File system component

Figure 3.5 illustrates a file system component. It shows labels only for the *w* privilege. Labels for the other privileges have been omitted for simplicity. Given the above rules, this file system component is interpreted as follows:

- Write access to the root directory is allowed, since its *self* label has a value of *allow*.
- Write access is denied for all files in the root directory except `/a`. Since the *children* label of the root directory is *unspecified*, it takes on the default value of *deny* that denies all file-related privileges for an empty tree.
- Write access is also denied to `/a`. Since its *self* label and the root directory's *children* label are both unspecified, it takes on the default value of *deny* that denies all file-related privileges for an empty tree.

- For all files in `/a` except `/a/b`, write access is denied. This is due to the setting of the *children* label for `/a`.
- Write access is allowed for the file `/a/b`, since its *self* label has a value of *allow*.
- Write access is allowed for all descendants of `/a/b`. This is because the *grandchild subtrees* label of the root directory is not overridden by any labels with higher precedence that affect descendants of `/a/b`.

Before file-related privilege checks are performed, names of files are converted to absolute pathnames that contain no symbolic links. Therefore symbolic links do not affect the behavior of file system components. However, the file system component must do extra privilege checking when a sandboxed process attempts to create a hard link. Before allowing this type of operation to proceed, the file system component computes the file-related privileges that the link would have if it existed. If these privileges exceed the privileges of the pathname being linked to, then the operation is denied. This prevents a sandboxed process from gaining unauthorized access to files simply by creating links to them in directories with more permissive settings. It can be shown that the set of all possible file system components is closed under the operations of union, intersection, and complement. However, we omit the proof for the sake of brevity.

When specifying security policies governing file access, two alternative strategies are possible. One option is to specify files according to directory path. Alternately, files may be specified by device number and inode.

An advantage of using pathnames is ease of use. Users are accustomed to thinking of files in terms of their pathnames as opposed to device/inode pairs. Furthermore, a directory hierarchy provides a convenient means of grouping related files in a simple and concise manner. For instance, a security policy stating that write access should be denied to all files in the subtree rooted at `/usr/src` is much easier to understand than a policy that lists every file in the subtree by device and inode. Furthermore, using device numbers and inodes requires the policy specification to be updated each time a file is added to or deleted from the given subtree.

A disadvantage of using pathnames to specify file-related security policies is that a given file may have multiple pathnames if it is referenced by more than one hard link. In addition, the pathname associated with a file may change over time if the file is renamed, or if a new link is created and the previous link is removed. File-related security policies given in terms of pathnames may therefore be difficult to properly specify, since proper access rights for a given file must be correctly assigned to each of its pathnames. From this point of view, specifying files by device and inode is simpler because the above-mentioned aliasing problem is avoided.

This option also has the advantage of preventing the following type of attack: Suppose an attacker wishes to gain unauthorized access to a file f in directory d_1. Also, suppose that the file system component C governing the attacker's actions allows creation of a hard link to f in directory d_2. This implies that the privileges associated with the new link in d_2 do not exceed those associated with the original link in d_1. However, the attacker may have reason to believe that at some future time, the creator of component C will reconfigure C so that additional privileges are granted for directory d_2. If C is eventually modified in this manner, the attacker may then gain unauthorized access to f.

When designing the file system component, we decided to specify privileges according to pathname because we believe that the great majority of potential users are likely to find pathnames easier to work with than device/inode pairs. However, we realize that this design choice may not be adequate for all users. Therefore, it may be worthwhile to eventually provide both options. This might be done by replacing the file system component with two separate components: a pathname component and an inode component. Privilege checks would then be performed against both components and file accesses would be granted only for operations allowed by both components.

An alternate means of guarding against the above-mentioned type of attack involving link creation is to modify the design of our existing file system component so that a new privilege, l, is added to the existing privileges r, w, x, p, t, and s. The l privilege would allow creation of a new hard link for the file corresponding to the given directory path. As with the other privileges, the blocking mechanism could then be used for interactive monitoring of attempts to create hard links.

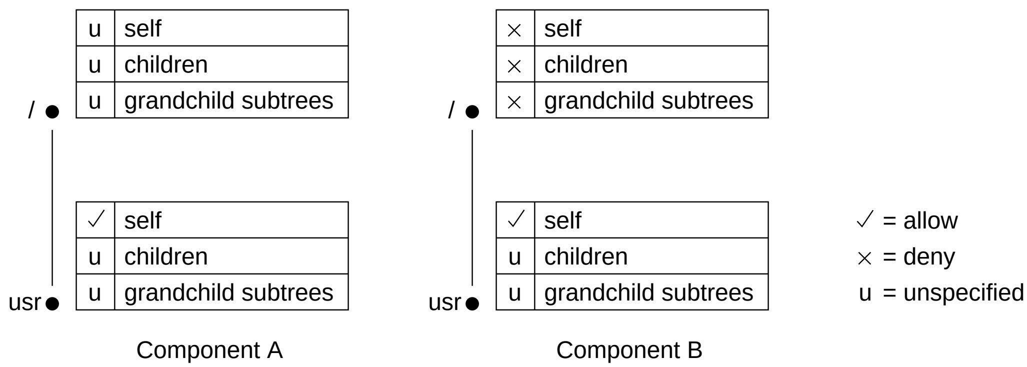

Figure 3.6: Two equivalent file system components

One observation regarding the behavior of our file system component is that two distinct file system components may have non-identical configurations but still assign the exact same set of privileges to the directory paths in the file system namespace. Figure 3.6 illustrates this possibility. Since an empty file system component denies all file access by default, it is unnecessary for component B to explicitly set its labels for the root directory to *deny*. As component A illustrates, setting these three labels to *unspecified* is sufficient since an empty file system component implicitly denies all file access.

In general, the potential for non-identical file system components to specify identical privileges with respect to the directory hierarchy stems from the existence of the *unspecified* label. Any of the labels attached to the root directory of component A may be changed to *deny* while maintaining equivalence with component B. Likewise, any

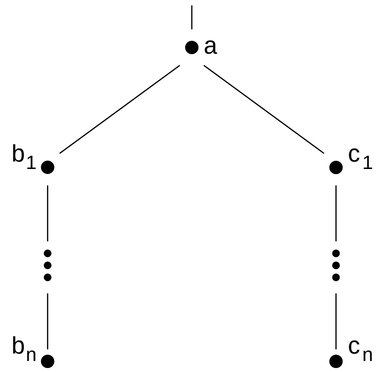

Figure 3.7: Partial view of file system component F

of the corresponding labels of component B may be set to *unspecified* while maintaining equivalence with A. An alternate means of altering B while maintaining equivalence with A is to append a leaf to the node representing `/usr` and set all of its labels to *unspecified.*

As discussed above, the option of setting a label to *unspecified* introduces a certain amount of complexity. However, we have provided this option because omitting it would make file system components much more cumbersome to manipulate, as illustrated by the partial view of file system component F in Figure 3.7. Suppose that the labels attached to the nodes of F are as follows:

- All three labels attached to node a are set to *deny.*
- All three labels attached to each of b_1, ..., b_{n-1} and c_1, ..., c_{n-1} are set to *unspecified.*
- The *children* and *grandchild subtrees* labels of b_n and c_n are all set to *unspecified.*

However, the *self* label of b_n is set to *allow* and the *self* label of c_n is set to *deny*. In summary, access is denied to all files in the subdirectory rooted at a except for b_n. The *self* labels of b_n and c_n are set explicitly to *allow* and *deny*, respectively, because the creator of component F wishes to maintain the invariant that access to b_n is allowed and access to c_n is denied regardless of how the labels of their ancestors are manipulated. Suppose we wish to change F so that for all nodes in the subtree rooted at a except b_n and c_n, access is granted. However, access privileges for nodes b_n and c_n are to remain unchanged. To implement this change, all one needs to do is set all three labels of a to *allow*. However, if the *unspecified* setting did not exist, all labels for nodes b_1, ..., b_{n-1} and c_1, ..., c_{n-1} would also need to be changed from *deny* to *allow*. As this example illustrates, the *unspecified* label makes file system components substantially easier to work with. We therefore believe that the *unspecified* setting is a worthwhile option to include despite the fact that it adds a certain degree of complexity.

The potential for equivalence of file system components with different label configurations raises the following question: Given two separate file system components A and B, how does one determine whether A and B are equivalent? This question may be answered using set-theoretic primitives as follows:

1. Create a component C representing the privileges given by $A \cap \overline{B}$.

2. Create a component D representing the privileges given by $B \cap \overline{A}$.

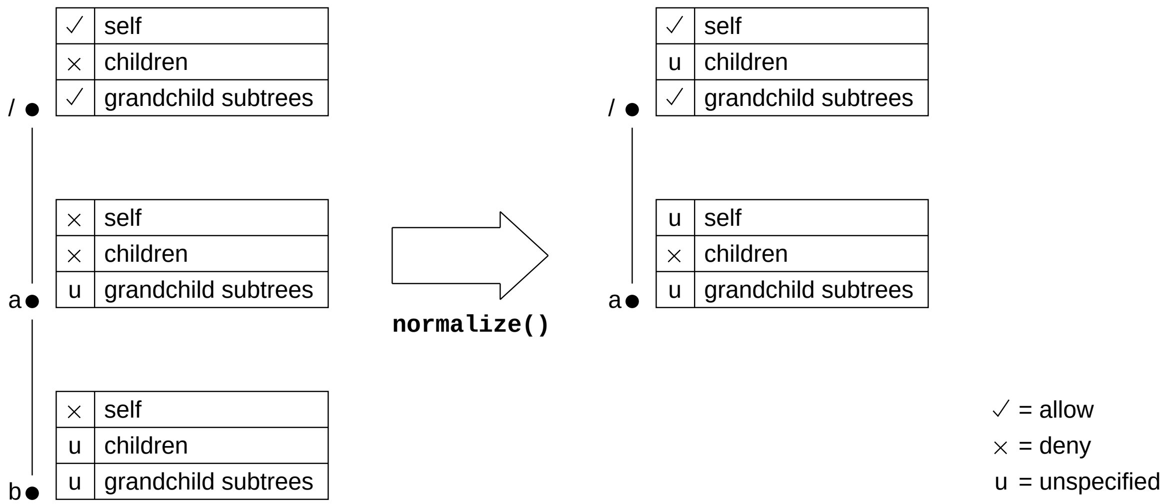

Figure 3.8: Normalizing a file system component C

Then A and B are equivalent iff C and D both represent the empty set.

Alternately, we may eventually implement a new system call `normalize()` for file system components. The effect of `normalize()` is to place a given file system component[2] into canonical form. Once two separate file system components are both in canonical form, they may be compared directly. In other words, they are equivalent iff their structures are exactly identical. Before defining canonical form, we first present two preliminary definitions:

> A node of a file system component is *minimal* if none of its labels can be set to *unspecified* without altering the set of privileges granted by the component.

> A node of a file system component is *extraneous* if it is a leaf and all three of its labels are set to *unspecified*.

[2]or subtree of a file system component

Given these two definitions, canonical form is defined as follows:

> A file system component is in *canonical form* if all of its nodes are minimal and none of its nodes are extraneous.

For example, consider the file system component C shown in Figure 3.8. C may be normalized as follows:

1. The root node of C is not minimal, since its *children* label may be changed from *deny* to *unspecified* without altering the set of allowed privileges. Therefore, it is minimized by setting its *children* label to *unspecified.*

2. The node corresponding to `/a` is not minimal, since its *self* label may be changed from *deny* to *unspecified* without altering the set of allowed privileges. Therefore, it is minimized by setting its *self* label to *unspecified.*

3. The node corresponding to `/a/b` is not minimal, since its *self* label may be changed from *deny* to *unspecified* without altering the set of allowed privileges. Therefore, it is minimized by setting its *self* label to *unspecified.*

4. The node corresponding to `/a/b` is now extraneous, since it is a leaf and all three of its labels are set to *unspecified.* Therefore, this node is eliminated. The final result is the component on the right side of Figure 3.8, which is in canonical form.

The *unspecified* label setting affects the semantics of the set-theoretic operations for file system components, as shown in Figure 3.9. When computing the intersection,

Z, of components X and Y, it is possible to set the *self* label of the node corresponding to `/usr` to either *deny* or *unspecified* without affecting the correctness of the result. Setting the label to *unspecified* would provide a simpler result, since this would make the node corresponding to `/usr` extraneous. This node could then be removed, resulting in an extraneous root node, which may then also be removed. The final result would consist of an empty tree, which is in canonical form. However, the design of our file system component specifies a different type of behavior. To facilitate description of this behavior, we define the following terms:

> When computing the intersection or union of two file system components A and B, the *dominant value* of a given privilege p with respect to A and B is the value of the privilege with respect to the result tree C. Here, a privilege may only evaluate to *allow* or *deny*. There is no notion of *unspecified*. For instance, in Figure 3.9, the dominant value of the *self* privilege for `/usr` with respect to trees X and Y is *deny*. If we were instead computing the union, the dominant value would be *allow*. Likewise, the dominant value of the *self* privilege for `/home` with respect to X and Y is *deny*. This is because both trees implicitly deny the given privilege[3] and the intersection of *deny* and *deny* is *deny*.
>
> A value for a given privilege p with respect to directory path q and file system component A may be either *explicit* or *implicit*. The value is explicit if there is a node representing q in A whose label for privilege p is set to either *allow* or *deny*. If the value for privilege p is not explicit, then it is *implicit*. The value of privilege p with respect to node A is implicit when its corresponding label is set to *unspecified* or there is no corresponding label[4].

Given these definitions, we now describe how labels are assigned in the result tree. In summary, explicit labels are preferred over the implicit setting of *unspecified*. More

[3]since `/home` appears in neither tree and the *self* privilege is therefore denied implicitly due to the settings of its ancestors (only the root node in this case)

[4]In other words, A contains no node corresponding to path q

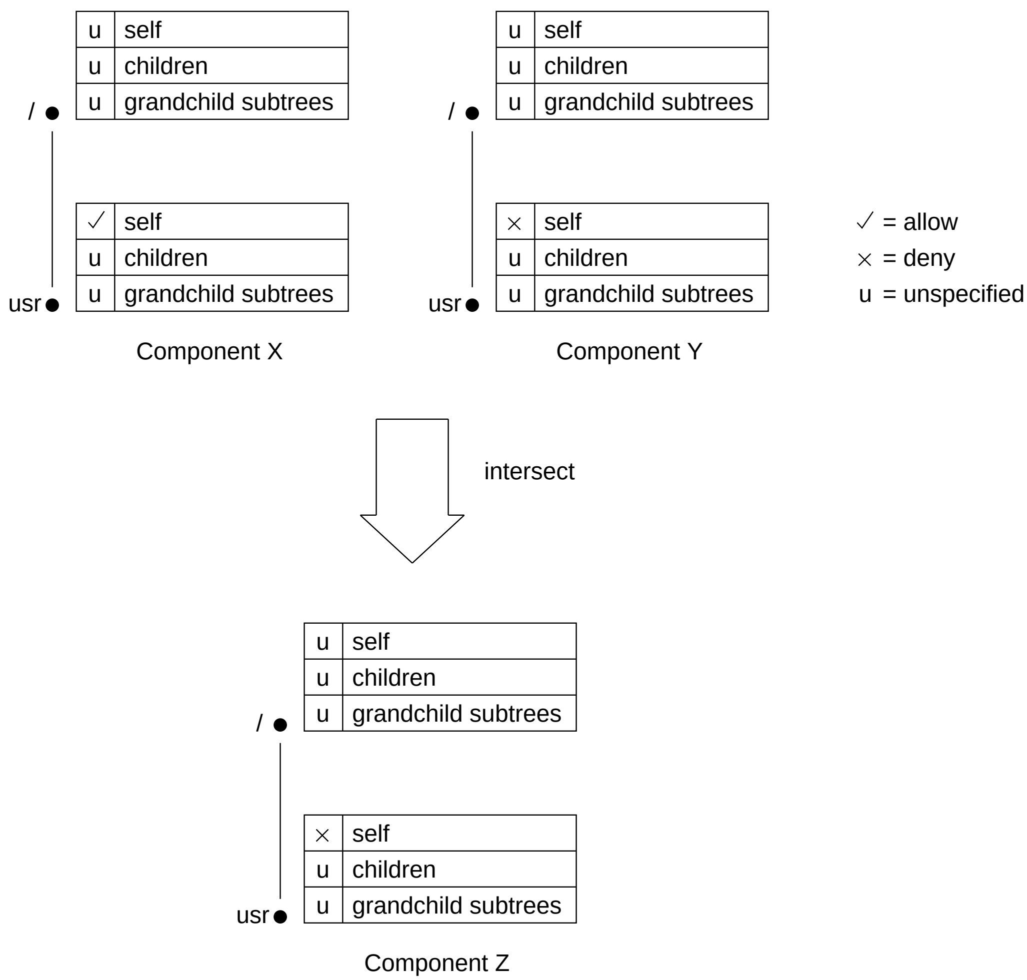

Figure 3.9: Computing the intersection of components X and Y

formally, suppose that we are computing the intersection or union, C, of two file system components A and B. The value for a label l corresponding to privilege p (*self*, *children*, or *grandchild subtrees*) associated with path q is computed in the following manner. Let v represent the dominant value of the privilege corresponding to l with respect to trees A and B. There are three cases to consider:

- *Case 1:* The value of privilege p with respect to tree A equals the dominant value v. However, the value of privilege p with respect to tree B does not equal the dominant value v. In this case, l is set explicitly to v if p is explicit in A. If p is implicit in A then l is set as follows: If either an explicit setting of v[5] or a setting of *unspecified* for l will produce the correct set of privileges for result tree C then the *unspecified* setting is chosen for l. Otherwise, the explicit setting v is chosen.

- *Case 2:* The value of privilege p with respect to tree B equals the dominant value v. However, the value of privilege p with respect to tree A does not equal the dominant value v. This is symmetric to case 1, and is handled in a symmetric manner.

- *Case 3:* The value of privilege p with respect to both trees A and B equals the dominant value v. In this case, l is set explicitly to v if p is explicit in either A or B. If p is implicit in both trees then l is set as follows: If either an explicit setting of v or a setting of *unspecified* for l will produce the correct set of privileges for

[5] either *allow* or *deny*

result tree C then the *unspecified* setting is chosen for l. Otherwise, the explicit setting v is chosen.

To motivate this behavior, consider Figure 3.9. Here, the *self* label for `/usr` is set explicitly to *deny* since *deny* is the dominant value for `/usr` with respect to X and Y, and the *self* label for `/usr` is set explicitly to *deny* in Y. The motivation for this choice is the conservative assumption that the *self* label for `/usr` is set explicitly in Y for a good reason. In other words, the creator of Y wishes to preserve the invariant that access to `/usr` should always be denied regardless of how the root directory is configured. Setting the *self* label for `/usr` to *deny* in result tree Z preserves this invariant. Therefore, the creator of Z may reconfigure its root node without inadvertently granting access to `/usr`.

3.5 Network Component

A network component consists of two interval lists that specify IP addresses that sandboxed processes may open connections to and ports that sandboxed processes may receive incoming connections from.

3.6 Device Component

A device component consists of three interval lists that specify `read()`, `write()`, and `ioctl()` privileges for various device numbers.

3.7 System Management Component

In its current implementation, the system management component is simply a set of Boolean flags that govern administrative actions such as rebooting and setting system date/time. The set of operations currently governed by this component type is not comprehensive, and will eventually be extended.

Chapter 4

Performance

In order to be practical, a security mechanism must not require an unreasonable amount of performance overhead. To demonstrate the feasibility of our design, we have therefore performed several microbenchmarks.

Aside from the addition of privilege checking hooks at various points in the

	`fork()`	`execve()`	`exit()`	`wait()`
total latency (μsec.)	169	375	145	—
overhead (μsec.)	6.8	1.2	5.9	11.2
overhead (% of total)	4.0	0.3	4.1	—

Table 4.1: Performance impact of sandboxing mechanism

code, our major changes to the standard Linux kernel involve modification of `fork()`, `execve()`, `exit()`, and `wait()`. We have therefore measured the amount of overhead that our mechanism adds to each of these system calls. All experiments were performed on a uniprocessor 266 MHz Pentium II PC with 96 Mb of memory. The Linux kernel we used is an SMP build of version 2.4.1. Each value in Table 4.1 represents the mean value from 10000 separate system call invocations. As shown, our modifications typically add several microseconds to each call.

During a `fork()`, sandbox-related state information must be copied from the parent process to the child. On `execve()`, a check is performed to see if a sandbox must be applied due to a previous invocation of `sbxapply()` with the `SBX_OPT_APPLY_ON_EXEC` option specified. The values in Table 4.1 reflect the typical case in which no sandbox is applied. We measured separately the latency of an `sbxapply()` system call (without `SBX_OPT_APPLY_ON_EXEC` specified) and found that value to be 56 microseconds.

During an `exit()` system call, our implementation closes any open descriptors for sandboxes and components. It then releases the reference to any sandbox the process may be executing within and does a partial cleanup of the sandbox if the reference count drops to 0. Additional cleanup of sandbox-related state is performed during `wait()` when the zombie process is collected. At this time, the expired sandbox is queued so that a kernel thread may perform the final cleanup. The values for `exit()` and `wait()` in Table 4.1 represent the case in which this cleanup activity occurs for a single expired sandbox. The purpose of the kernel thread is to remove the sandbox from the global matrix described in Section 3.2 and free the memory that it occupies. The thread is awakened periodically when the number of expired objects on its queue reaches a certain threshold. It then deletes all of them in a single operation. We measured the time required to delete 1024 expired sandboxes, and found that this operation takes 2829 microseconds (2.8 μsec. per sandbox). This represents the mean for 10 separate invocations of the kernel thread. Adding the per-sandbox value to the overhead values in Table 4.1 for `exit()` and `wait()` provides a rough idea of the total overhead required for destroying a sandbox.

Additionally, we measured the latency of the `kill()` system call when executed by a sandboxed process. The results are shown in Figure 4.1. For this experiment, we configured the sandbox of the sending process p so that its sandbox allows sending signals to the receiving process q, which has been placed within a separate sandbox. The values represent latencies when p is placed in sandboxes nested at various depths. For instance, the value 3 on the horizontal axis represents the case in which the sandbox enclosing p has a parent and a grandparent. Therefore, privilege checks occur at three separate levels. The value 0 on the horizontal axis indicates the case in which p is not inside a sandbox and therefore no privilege checks occur. As the graph shows, a single privilege check incurs approximately 5 microseconds of overhead. When sandboxes are nested, additional privilege checks incur approximately 1 microsecond each.

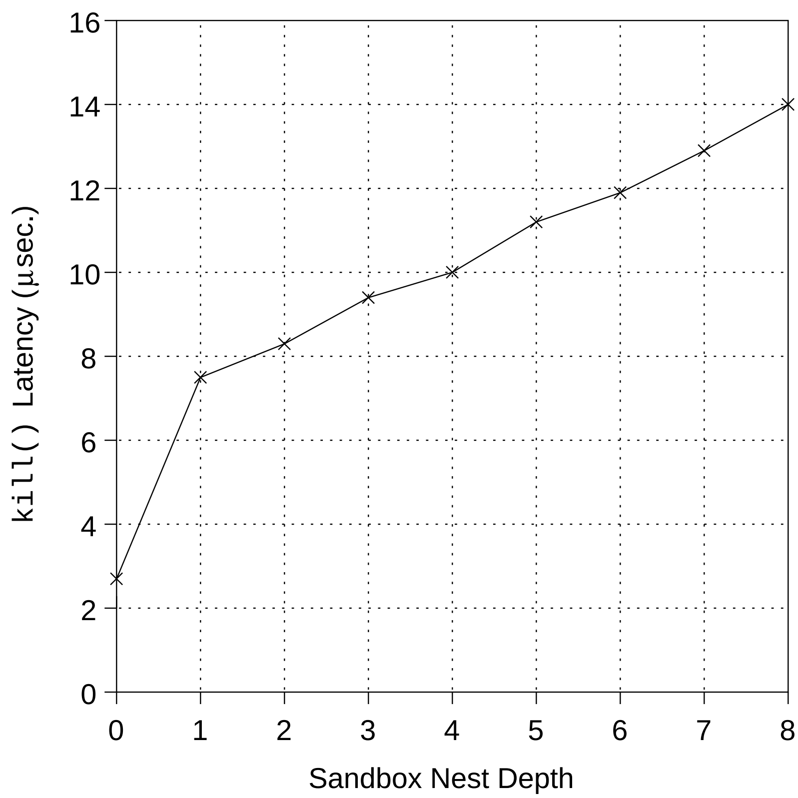

Figure 4.1: Latency of `kill()` executed by sandboxed process

Chapter 5

Related Work

Access control lists (ACLs) are a commonly used mechanism for enhancing system security. They associate detailed access rights with objects such as files. The main difference between sandboxes and ACLs is that they take opposite points of view. ACLs associate privileges with objects while sandboxes associate privileges with subjects. The centralized location of the controls on sandboxes makes the correctness of their settings easy to verify. Sandboxes impose strict upper bounds on privileges without depending on assumptions such as settings of file permissions throughout the system. They permit easy creation of customized protection domains without having to change settings on a wide variety of system objects. However, our sandboxing mechanism is designed to complement alternatives such as ACLs rather than replacing them. Sandboxes may be used in combination with other mechanisms to implement policies not easily enforceable using any single mechanism by itself.

Capabilities[9] are another alternative to sandboxes. A capability has two primary characteristics:

- A subject that holds a capability is granted access to the privilege it specifies.
- A subject that lacks a capability is denied access to the privilege it specifies.

A sandbox exhibits the second property but not the first one. This aspect of sandboxes allows their controls to be safely manipulated by untrusted users. The centralized location of the controls on a sandbox makes sandbox-granted privileges easy to track and revoke. In contrast, complete revocation of a capability C held by process P requires revocation from both P and all processes to which C has been delegated by P. The ability to create nested sandboxes provides a mechanism for delegation of privileges in a manner somewhat similar to delegation of capabilities. Opening files represents an interesting area of interaction between sandboxes and capabilities, since a file descriptor may be viewed as a capability for accessing a file. In our current design, a sandbox cannot revoke a previously granted file access privilege once the sandboxed process has obtained a file descriptor. However, this limitation may be removed by attaching sandbox-related tags to file descriptors and performing additional privilege checks during `read()` and `write()` system calls. Although this requires extra overhead, the creator of a sandbox could be given the option of disabling the feature to increase performance.

Domain and type enforcement (DTE)[6, 1] is a useful tool for implementing mandatory access controls. This technique groups subjects into domains and objects into types. Rules are provided that specify which domains are granted access to which types. In addition, the system may be configured so that execution of certain programs causes transitions between domains. A major difference between DTE and our sandboxing mechanism is that DTE is geared toward implementing systemwide mandatory access controls. A trusted security administrator defines the domains and types, along with the rules governing their interactions. In contrast, sandboxes are lightweight entities that may be created, configured, and destroyed by untrusted users. Our implementation allows them to enforce either mandatory or discretionary access controls. We plan on extending their functionality by allowing transitions between sandboxes when certain programs are executed.

A variety of sandboxing techniques have been previously implemented. One approach is to build protection mechanisms into programming languages such as Java[8]. Since this option ties the sandbox to a particular language or runtime environment, it is not suitable as a general-purpose mechanism. However, it is still useful as a special-purpose technique, since security policies may be tailored to the language or runtime environment.

Alternately, the sandbox may be embedded within the sandboxed program. Proof-carrying code[7] is one example of this type of approach. Another option is to instrument an existing binary with additional machine instructions that verify compliance with a security policy as a program executes[4]. However, these alternatives are inconvenient because they require modification of binaries. Furthermore, they are not useful as general-purpose techniques since they do not apply to all types of programs (such as shell scripts, for instance). As discussed previously, an additional disadvantage of these techniques is that the burden of defining security policies is placed on application developers. Therefore, developers are forced to anticipate the needs of a potentially diverse user base, adopting a "one size fits all" approach to security that is unlikely to be a good match for the requirements of particular individuals.

A sandboxing system known as Janus[16], along with two similar mechanisms[2, 14], employs user-space monitoring processes for interception of system calls made by sandboxed programs. The monitoring processes use the `/proc` process tracing facility of Solaris for system call interception. This approach limits the scope of applicability of these techniques, since it may not be used with setuid programs. It also has substantial overhead because the monitoring agent is a separate process and interprocess context switches are therefore required for monitoring. Furthermore, the monitoring process must `fork()` each time the sandboxed process forks. The fact that the monitoring agent runs in user space may also create vulnerabilities. Other advantages of our design over these solutions include the ability to nest sandboxes hierarchically and the flexibility provided by our set-theoretic approach to privilege specification.

To overcome the limitations of user-space mechanisms, sandboxes may be implemented as loadable kernel modules[12, 17]. Placing sandboxes inside the kernel may enhance their security by providing increased isolation from potentially malicious entities. Since kernel-based sandboxes may be implemented as passive entities, context switching overhead is not required for privilege checking. A disadvantage of this approach is that creating a new sandbox requires loading a kernel module. The module must be fully trusted, and a trusted user must perform the module loading operation. As new sandboxes are defined, the set of all code that must be trusted with full kernel privileges expands without bound. In contrast, our design only places a core set of primitives within the kernel. Application-dependent aspects of sandbox creation and manipulation are therefore pushed into user space where they belong.

A design known as ChakraVyuha (CV)[15] implements a kernel-based sandboxing mechanism. In this system, sandboxes for individual applications are defined using a domain-specific language. Sandbox definitions are stored in a secure location somewhere in the file system. When a given program is executed, its sandbox definition is passed to a kernel-resident enforcer. This entity enforces restrictions by matching system call parameters against the sandbox definition. Therefore, problems associated with implementing sandboxes as loadable kernel modules are avoided.

One difference between ChakraVyuha and our design is the level at which its external interfaces are specified. To confine a program with ChakraVyuha, it must first be installed using a specialized installer program. The installer generates a configuration file that specifies a default sandbox for the new program. If users wish to create customized sandboxes, they must do so using configuration files that follow a specific format. Our external interface is at a much lower level. We export a general-purpose system call API that application programs may use for their own purposes. This approach widens the scope of applicability of our design.

A second advantage of our model is the ability to dynamically reconfigure sandboxes at runtime. With ChakraVyuha, users may customize sandboxes, but the sandboxes are fixed once the sandboxed programs start executing. Other advantages of our model include nested sandboxes and our treatment of privilege sets as first class objects that may be manipulated using set-theoretic primitives.

Another solution, known as WindowBox[5], implements a sandboxing mechanism within the Windows NT kernel. The emphasis here is on providing an easy to use mechanism that is simple enough for unsophisticated users. The design consists of a set of desktops that are completely separated from each other and from the rest of the system. Users can give some desktops more privileges than others. As a user's level of trust increases, a program may be gradually moved to more privileged desktops. However, the desktops are relatively static entities. They are not designed to function as lightweight containers for individual programs. As discussed previously, our implementation of sandboxes as lightweight, transient entities widens the scope of applicability of our design. This is due to reduced invasiveness of kernel modifications and elimination of incompatibility with existing file systems. In contrast, a design similar to WindowBox would have serious code maintenance and compatibility problems if it were ported to an open-source operating system such as Linux.

Finally, a sandboxing mechanism somewhat similar to ours has been added to the ULTRIX operating system[18]. This mechanism, known as TRON, is similar to our design in some ways but more limited in scope, since it only deals with file-related privileges. Like our sandboxing mechanism, TRON allows creation of sandboxes by untrusted users. However, it does not provide a blocking mechanism for interactive privilege determination at runtime.

TRON does allow nesting of sandboxes, although this feature behaves differently from our design. When sandboxes are nested, our mechanism performs privilege checks at each level individually. However, TRON verifies at creation time that a nested sandbox contains a subset of its parent's privileges. It then checks privileges against only the innermost sandbox. Although TRON's approach reduces performance overhead, we chose our method for two reasons. First of all, our design allows changes in a sandbox configuration to affect all sandboxes nested below it. This behavior is necessary for interactive manipulation of sandboxes to function properly when sandboxes are nested. Secondly, our design allows a sandboxed process to create a nested sandbox without any awareness of how its own sandbox is configured. The child sandbox is not cluttered with restrictions imposed by its parent and therefore maintains a precise representation of the policies its creator wishes to enforce. Furthermore, restrictions imposed by the parent sandbox may be kept secret from its inhabitants.

The method that TRON employs for specifying access controls is less expressive than our file system component. When privileges are assigned to a directory, they automatically extend to all files it contains. It is not possible to grant privileges only for the directory without extending them to all of its files. However, a *subtree* option does exist that is equivalent to the union of *self*, *children*, and *grandchild subtrees* in our file system component. One feature that TRON omits is the ability to specifically deny access to files. It is therefore not powerful enough to support composition of privilege sets through union, intersection, and complement operations.

Chapter 6

Conclusions and Future Work

In summary, we have presented a general-purpose system call API for confinement of untrusted programs. We have described our design within the context of a systematic exploration of the design space for confinement mechanisms. Our approach is distinguished by its flexibility and provision of a relatively simple set of primitives that permit a wide scope of applicability. Preliminary performance results are encouraging, although we still need to perform more extensive testing.

As discussed previously, we intend to add a mechanism for causing transitions between sandboxes as a result of executing certain programs. This will provide a means of giving trusted programs additional privileges in a safer, more fine-grained manner than the `setuid()` mechanism provided by existing UNIX systems.

Additionally, we may create a new type of component that governs file access according to device number and inode. This would complement our existing file system component, providing users with an alternate means of defining file-related security policies.

Guarding against denial of service attacks is another area for future improvement. Here, we intend to study solutions that others have developed[3, 13] and incorporate them into our design.

The Linux community has recently been developing an infrastructure for integrating security modules into the kernel[11]. This provides users with the ability to integrate various types of security-related functionality into the kernel without having to patch a source tree and do a complete kernel rebuild. We intend to adapt our implementation so that it interoperates with the Linux Security Modules framework. This will widen our potential user base and simplify maintenance of our code as new kernels become available.

Another potential area for future work is the addition of a new type of component that governs access to the sandboxes and components themselves. Although this approach has a certain elegance, it requires careful consideration since it may create an undesirable amount of complexity.

We also intend to add a `normalize()` system call for file system components, allow revocation of open file descriptors, and extend the system management component so that it governs a more comprehensive set of privileges. Another useful feature will be new system calls for creation and manipulation of iterators for stepping through components and examining their contents.

Bibliography

[1] Security-enhanced linux. *http://www.nsa.gov/selinux/*.

[2] Acharya, A., and Raje, M. MAPbox: Using parameterized behavior classes to confine applications. In *Proceedings of the 9th USENIX Security Symposium*, Aug. 2000.

[3] Chang, F., Itzkovitz, A., and Karamcheti, V. User-level resource-constrained sandboxing. In *Proceedings of the 4th USENIX Windows Systems Symposium*, Aug. 2000.

[4] Small, C. A tool for constructing safe extensible C++ systems. In *Proceedings of the Third USENIX Conference on Object-Oriented Technologies and Systems*, Jun. 1997.

[5] Balfanz, D., and Simon, D. WindowBox: A simple security model for the connected desktop. In *Proceedings of the 4th USENIX Windows Systems Symposium*, Aug. 2000.

[6] Walker, K., Sterne, D., Badger, L., Petkac, M., Shermann, D., and Oostendorp, K. Confining root programs with domain and type enforcement (DTE). In *Proceedings of the Sixth USENIX Security Symposium*, Jul. 1996.

[7] Necula, G., and Lee, P. Safe kernel extensions without run-time checking. In *Proceedings of the USENIX 2nd Symposium on Operating Systems Design and Implementation*, Oct. 1996.

[8] Gong, L., Mueller, M., Prafullchandra, H., and Schemers, R. Going beyond the sandbox: An overview of the new security architecture in the java development kit 1.2. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Dec. 1997.

[9] Dennis, J., and VanHorn, E. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9:143–155, Mar. 1966.

[10] Saltzer, J., and Schroeder, M. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, Sep. 1975.

[11] Wright, C., Cowan, C., Smalley, S., Morris, J., and Kroah-Hartman, G. Linux security modules: General security support for the linux kernel. In *Proceedings of the 11th USENIX Security Symposium*, Aug. 2002.

[12] Fraser, T., Badger, L., and Feldman, M. Hardening COTS software with generic

software wrappers. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, May 1999.

[13] Lal, M., and Pandey, R. A scheduling scheme for controlling allocation of cpu resources for mobile programs. *Autonomous Agents and Multi-Agent Systems*, 5(1):7–43, Mar. 2002.

[14] Alexandrov, A., Kmiec, P., and Schauser, K. Consh: Confined execution environment for internet computations. *http://www.cs.ucsb.edu/~berto/papers/99-usenix-consh.ps.*

[15] Dan, A., Mohindra, A., Ramaswami, R., and Sitaram, D. Chakravyuha (CV): A sandbox operating system environment for controlled execution of alien code. Technical Report 20742, IBM T.J. Watson Research Center, 1997.

[16] Goldberg, I., Wagner, D., Thomas, R., and Brewer, E. A secure environment for untrusted helper applications (confining the wily hacker). In *Proceedings of the Sixth USENIX Security Symposium*, Jul. 1996.

[17] Mitchem, T., Lu, R., and O'Brien, R. Using kernel hypervisors to secure applications. In *Proceedings, 13th Annual Computer Security Applications Conference*, Dec. 1997.

[18] Berman, A., Bourassa, V., and Selberg, E. TRON: Process-specific file protection for

the UNIX operating system. In *Proceedings of the 1995 Winter USENIX Conference*, Jan. 1995.

Appendix A

Availability

At the time of this writing, we are still finishing the implementation of the sandboxing API. The latest version of the code may be obtained from `http://seclab.cs.ucdavis.edu/projects/sandbox.html`. As our work progresses, we will make updates available at this location.