Quarantining Untrusted Entities: Dynamic Sandboxing using LEAP

Manigandan Radhakrishnan
University of Illinois at Chicago
mani@rites.uic.edu

Jon A. Solworth*
University of Illinois at Chicago
solworth@rites.uic.edu

Abstract

Jails, Sandboxes and other isolation mechanisms limit the damage from untrusted programs by reducing a process's privileges to the minimum. Sandboxing is designed to thwart such threats as (1) a program created by an attacker or (2) an input crafted to exploit a security vulnerability in a program. Examples of the later include input containing interpreted code or machine language to be injected via a buffer overflow.

Traditionally, sandboxes are created by an invoking process. This is effective for (1) but only partially so for (2). For example, when a file is downloaded by a browser or processed as a mail attachment, the invoking process can sandbox it. However, sandboxing protections can be circumvented when the file is copied outside the sandbox. The problem is that traditional sandboxes do not provide complete mediation.

We introduce dynamic sandboxes*, and show how even when data is saved and/or copied, sandboxing protections are not lost. In addition, and in contrast to traditional sandbox implementations, dynamic sandboxes are implemented using general purpose access controls. Not only does this provide a more flexible sandbox mechanism, and enable complete mediation, but these same primitives can be used to build other (non-sandbox) authorization policies.*

1. Introduction

Traditional operating systems have evolved towards large, flat address spaces and widely accessible resources because of their convenience and simplicity. Widely used (discretionary) access controls have similarly been flat—allowing or denying operations based *solely* on the user on whose behalf the process executes. While this removes barriers to implementing applications and provides flexibility, it also increases the danger when, all too often, an application is attacked.

Even if applications could be written without security flaws, this flat access control model is problematic as applications come from many sources, not all of which can be trusted. Unfortunately, in this environment the least trusted programs run with the same permissions as the most trusted; the most frivolous programs can interfere with the most sensitive applications.

Sandboxes and Jails limit which resources are visible to a process. Resources which are not visible can neither be observed nor operated upon, and hence are protected from the actions of the sandboxed processes. Thus sandboxing increases both confidentiality and integrity.

One of the purposes of sandboxing is to reduce the privileges provided to an executable, and thus to implement least privilege. If an executable has security holes, then sandboxing limits the damage resulting from an attack.

But it is not always appropriate to accord a fixed set of privileges to an executable. One basis for varying the permission is the user who owns the process, which is easily handled by existing sandbox mechanisms. Another basis for varying the privileges is the data on which the process executes. If this data comes from an untrusted source—such as an email attachment or a web download—it may be necessary to curtail the processes' privileges.

Of particular concern are code injection attacks. Consider a data file which is read by an executable. If that data file contains either

- interpreted code (and the executable contains a suitable interpreter) or
- a buffer overflow attack (and the executable is susceptible),

then the execution may be completely determine by the "data" file. As we shall see, sandboxing mechanisms attempt, but do not always succeed, to attenuate privileges in this case.

Sandboxes are typically entered by a helper application (such as a PDF viewer), and may be defined solely by the helper class or by the program (e.g., browser) which

*This work was supported in part by the National Science Foundation under Grants No. 0627586 and 0551660. Any opinions, findings and conclusions or recommendations expressed in this paper are those of the author and do not necessarily reflect the views of the National Science Foundation.

launches the helper. The mechanism for invoking the sandbox depends on how the executable is launched. By launching the sandboxed application when invoked form a mail user agent or browser, unvetted files are caught at the source.

But the mechanism for invoking traditional sandboxes is incomplete. Thus there are cases where either the sandbox is not invoked, or the privileges not appropriately attenuated. The reason for this incompleteness is that sandbox protections are uni-directional[1]. That is, while the sandboxed process is prevented from seeing outside its sandbox, processes outside the sandbox can see into the sandbox. Hence, files stored in the sandbox can be copied outside the sandbox by an external process and thus escape sandbox protections. Indeed, it is desirable to do so, as users regularly incorporate outside objects (e.g., papers, news stories, and other publicly available material) into the things they create, for example by downloading from the web or saving an email attachment. Thus sandboxing protections can depend on *how* the file is accessed rather than *whether* the file is accessed. The problem is a lack of complete mediation.

For an application to be suitably sandboxed, the sandbox should depend on both the executable and the sources of its data. We have designed and implemented *dynamic sandboxes* which quarantine untrusted sources and enable complete mediation. Dynamic sandboxes

- enable each file to be labeled with its trust (or origin) and
- ensure that this labeling is maintained through information flow rules, and either
  - prevent untrusted data from getting near sensitive executables (those with important privileges) or
  - reduce the privilege of an executable when it reads untrusted data.

Because of the need for complete mediation, we have constructed dynamic sandboxing on top of a general purpose mandatory access control model called *LEAP (Language for Expressing Authorization Properties)*. (Dynamic sandboxing is often built on special purpose mechanisms, but these do not ensure the mediation required.) LEAP allows the high level specification of access controls, which the operating system enforces at runtime. In addition to complete mediation, this approach enables the LEAP primitives to be used to construct other non-sandbox protections and also results in highly customizable sandboxes.

LEAP was developed as a high level specification of operating system level authorizations. It has the following advantages:

1. The operating systems enforcement called KernelSec domains [34] is automatically generated [23] from the LEAP specification.
2. It can be automatically analyzed [42, 41].
3. It is succinct and for the most part stateless and hence is (relatively) easy to read.
4. It supports administrative controls [43].

Properties (1) and (2) distinguish it from RBAC [37], while Properties (3) and (4) distinguish it from Type Enforcement.

Another issue with traditional sandbox mechanisms, is that they are not widely used. We therefore have paid particular attention to the complexity of use, by having the mechanism adapt to the user (rather than requiring the user to adapt to the mechanism). Dynamic sandboxes are invoked automatically and are transparent to the user. Dynamic sandboxes can be configured by the system administrator (or distribution packager) who can make appropriate tradeoffs of usability vs. protections.

The contributions of this paper are two-fold: the introduction of a new sandbox mechanism with more complete mediation and the demonstration of the flexibility of LEAP specifications.

The remainder of the paper is organized as follows. Section 2 describes how to define sandboxes in LEAP and provides an example of such a sandbox; Section 3 describes the implementation of dynamic sandboxes at the operating systems level, including performance. Section 4 describes related work. Finally, in Section 5 we conclude.

2. Dynamic Sandboxes

In this section, we show how dynamic sandboxes can be used to quarantine objects of dubious origin. To illustrate this, we shall describe a mail user agent, like Thunderbird, in which mail attachments can be viewed using helper applications. In a full system, our example would be expanded to include many different helper applications (e.g., image viewers, postscript interpreters, video players, etc.) as well as different client applications (e.g., browsers, RSS readers, etc.).

We describe the quarantine mechanism by providing a high level specification for a dynamic sandbox. In LEAP, files are labeled. Downloaded content is therefore differently labeled, and treated, than other files. The quarantine mechanism ensures that any process that reads files containing downloaded content is suitably sandboxed. This content may then be integrated with the other files under controlled circumstances, for example by: *certifiers* (which relabel innocuous content) and *scrubbers* (which remove dangerous content). An example of a scrubber is antivirus software.

Unlike traditional sandboxes, dynamic sandboxes ensure the isolation of tainted information even if the information

[1] It is possible to build bidirectional protections of such a mechanism by essentially sandboxing everything, but this results in a static partitioning.

has been copied between files or the information is integrated from different sources. This requires that dynamic sandboxes use information flow to track file sources.

Information flow implies that objects are labeled with their taintedness, so that the propagation of information can be tracked. Unfortunately, pure *Discretionary Access Control (DAC)* authorization models cannot track such information as they allow a user to arbitrarily change a label. And if a program could arbitrarily change labels, this would allow—either accidentally or on purpose—the evasion of sandbox protections. Hence, pure DAC models are insufficient for dynamic sandboxes.

While tracking information flows is essential, it is important that sufficient flexibility be maintained to balance off protections vs. usability.

1. Dynamic sandboxes need to be able to remove the "taint" of the input before allowing unrestrictive use of a file's contents. For example, a file from a trusted source—even if sent by email—should be able to escape its tainted label and be freely integrated with other files.

2. Different programs have different susceptibility to attack, and the mechanisms will need to account for that.

Unlike traditional sandboxes, dynamic sandboxes need to be tightly integrated with the operating system's authorization model. First, so that the labeling is consistently maintained inside and outside the sandbox. And second, to cleanly integrate privilege attenuation when accessing tainted files.

Although we shall describe one particular, but very useful, sandbox scenario we believe that it is necessary to allow the system administrator to tradeoff the protections vs. the usability of the system. This means that the resulting system needs to be very flexible. (Indeed flexibility appears to be a major reason why pure DAC systems continue to be used long after the attack threat has overtaken them.) We provide the needed flexibility here in a high level authorization specification language, which we describe next.

2.1. LEAP

LEAP is a language for describing a broad range of authorizations (it was originally called SPBAC) [42, 43]. LEAP has evolved to now include aspects (§ 2.1.1) and transitions on using privileges (§ 2.1.3). These LEAP mechanisms are general, in that none of them are used *solely* to implement dynamic sandboxes—they all have other uses. We include a short description of LEAP here for completeness, and then use LEAP to describe how to implement dynamic sandboxes.

A LEAP specification consists of (1) users and their subdivisions called aspects, (2) groups, (3) labels, and (4) permissions. Each object—such as a file—has a single label by which permissions to that object are determined.

2.1.1 Groups, users, and aspects

Each individual is represented as a user, denoted u. Each user is further divided into *aspects*, denoted $u_0, u_1, \ldots, u_n$; the aspects of a user are partially ordered. If an aspect $u_i \geq u_j$ then aspect u_i has all the privileges that u_j has (and will generally have additional privileges).

Aspects enable

- location to be a factor in determining permissions. Hence, a person logging in from work may have more permissions than when logging in from home.

- a user to isolate some of their processes from others.

For each group g, there is a set s_g which contains at most one aspect u_i for each user u. We say that $u_j \in g$ iff $\exists_{u_i \in s_g} u_j \geq u_i$; that is, u_j is a member of g iff some aspect less than or equal to u_j is in s_g. Hence, each group can explicitly contain one u_i per u and implicitly contains all aspects greater than u_i. Aspects build upon ideas in RBAC'96, but allow both location and host to be a factor in determining the permissions that a user has.

Because these aspects form a partial order, and because, as we shall see, permissions are based in part on groups and therefore aspects, this mechanism is ideal for implementing a sandbox.

2.1.2 Permissions

LEAP permissions are defined on labels. We next describe the names of unary and binary permissions:

unary permissions Given a label l, the names of the unary permissions on files are $c(l)$, $r(l)$, $w(l)$, and $x(l)$. These permissions are needed to perform the operations create, read, write, and execute respectively, on objects with label l.

There is also permissions to $connect(l)$, $bind(l)$, and $accept(l)$ where l is the label of a network connection.

binary permissions LEAP's binary permissions are defined over a pair of labels:

$relabel(l, l')$ The permission to change an object's label from l to l'.

$mayFlow(l, l')$ The permission to write (resp. create) l' after having read l. For each label l_i that a process has read before trying to write (resp. create) l', it must have the permission $mayFlow(l_i, l')$. The mayFlow's are necessary, but not sufficient; the process also needs permission $w(l')$ (resp. $c(l')$).

As we shall see, the relabel permission can be used to change an object's label from a "tainted" one to an "untainted" one. Because relabel is a permission, it can be used

to control both whether a given label can be changed and if so under what circumstances. We note that there is no label creep, because relabeling is an explicit operation.

The mayFlow permissions allow precise specification of allowed information flows, they differ from lattices in that they are general enough to specify (1) assured pipelines (2) downgrades and most essentially (3) the programs which have the permission (see below).

permission definition A permission is defined by specifying the *holder* of that permission. The holder specifies which processes can use the permission, by specifying a list of group, executable label pairs. Hence, the user in conjunction with the program executing determine the permissions accorded to the process.

Unary permissions are defined at the time the label is created while binary permissions are defined after both specified labels are created. In either case, once defined, the permission definition cannot be changed[2]. Because of this, we define permissions as follows for a read (other permissions are similar):

$$r(l) \quad = \quad (e_0, g_0), (e_1, g_1), \ldots, (e_n, g_n) \qquad (1)$$

where e_i is a label and g_i a group of aspects. The left hand side is called the permission name while the right hand side is called the *holder* of the permission. Consider a process p which executes on behalf of aspect u_a and whose executable file has label e. Then p is a holder of the above read permission if for some $0 \leq i \leq n$, $e = e_i$ and $u_a \in g_i$.

For example, in the below permission definition, an aspect in group *anyUser* can write object with the label l using executables labeled *lform*; but only aspects in group *admin* can write it with executables labeled *xmlEditor*.

$$w(l) \quad = \quad (lform, anyUser), (xmlEditor, admin)$$

Information flow requires multiple privileges. For example, the permissions

$$\begin{aligned} r(l) &= (e, anyUser) \\ w(l') &= (e, anyUser) \\ mayFlow(l, l') &= (e, anyUser) \end{aligned}$$

enable aspects in group *anyUser* to read l and then write l'. (If any of the above permission holders were empty, then information flow form l to l' could not occur.)

2.1.3 Relinquishing privileges

Finally, we need a mechanism to reduce privileges of a process. When entering a sandbox, privileges are reduced so that the sandboxed processes do not interfere with other processes. This relinquishing of privileges occurs on the exercise of a specified privilege $priv$ by a process whose executable is labeled e. The following reduces the aspect to the minimum for a user; there are many such minimum aspects allowing multiple sandboxes to exist concurrently per user.

$$minAspect(priv, e)$$

The minAspect ensures that multiple sandboxes of the same user will be isolated from each other.

2.2. Dynamic Sandbox Specification

In this section we describe dynamic sandboxing for email attachments. The description uses:

Shell which is the starting point.

Mail User Agent (MUA) such as Outlook or Thunderbird, which reads mail from a network server and labels as MAIL (when storing on the disk).

Viewer A PDF viewer. The sandbox allows it to either read and write ordinary user files, or to read and write MAIL files only. That is, it prohibits the copying of MAIL files to ordinary user files.

Copy this is the label for the `cp` program which creates a copy of a file.

Scubber such as a virus scanner which renders dangerous file contents harmless, thus producing ordinary files from mail files.

Certifier enables files which are created by trusted remote sources to be treated as if they were locally created, by changing their label.

The permissions to implement a dynamic sandbox are given in Figure 1 for the six types of executables above. Since any aspect can execute these programs, for simplicity we leave off the groups in holders and list just the executable labels in the figure.

Figure 2 shows the combinations of allowed read and writes operations that various executable labels could have given the LEAP specification. Note that some executable labels give rise to different combinations of allowed operations.

2.2.1 The mail user agent

The Mail User Agent (MUA) needs to fetch and send mail over a network, store the mail locally and run helper applications on mail attachments.

To ensure that the mail and its attachments are quarantined, only the MUA can access the networked mail service (Imap for fetching mail and Smtp for sending mail). The MUA can write only files with a MAIL label to denote that

[2]While the permission definition cannot be changed, the permission *can* be changed by either changing which aspects are members of a group or by relabeling an object.

connect(IMAP), r(IMAP), w(IMAP)	=	MUA
connect(SMTP), r(SMTP), w(SMTP)	=	MUA
mayFlow (IMAP, MAIL), mayFlow (SMTP, MAIL)	=	MUA
mayFlow (MAIL, IMAP), mayFlow (MAIL, SMTP)	=	MUA
c(MAIL), w(MAIL)	=	MUA,COPY
r(MAIL)	=	VIEWER, SCRUBBER, CERTIFIER, MUA, COPY
x(VIEWER), x(MUA), x(SCRUBBER), x(CERTIFIER)	=	*
r(USERFILES)	=	VIEWER, SHELL, COPY
c(USERFILES), w(USERFILES)	=	VIEWER, SCRUBBER, SHELL, COPY
mayFlow (MAIL, USERFILES)	=	SCRUBBER
relabel(MAIL, USERFILES)	=	CERTIFIER
minAspect(r(MAIL), VIEWER)		

Figure 1. Sample LEAP specification for dynamic sandbox

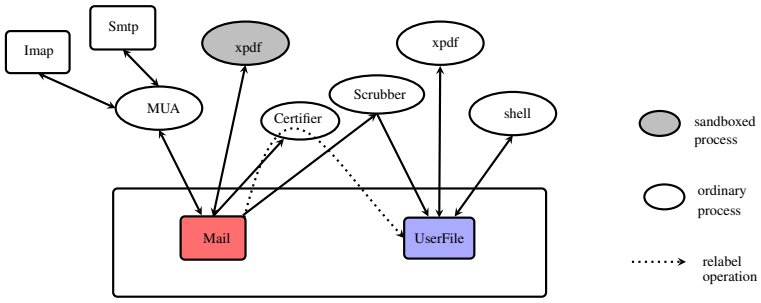

Figure 2. Interaction between filesystem objects and domains. Here ovals represent processes and rectangles represent files or network sockets. The direction of the arrow represents information flow from or to the process.

their (original) source is unknown. This prevents the reading of mail without the resulting files being marked with the MAIL label.

To allow the MUA to send and receive mail, it needs to connect to the IP addresses and port numbers of the SMTP and IMAP services. Hence, `connect` privileges are needed as well as read and write privileges to send and receive network data. In the network configuration for this example (not shown), the correspondence of the labels (SMTP and IMAP) to the IP address and port are given. (LEAP's networking protection associates labels for connect, bind, and accept system calls. The full mechanism for networking authorization and authentication is described in a paper under preparation.)

In addition, the MUA may `exec` helper applications; in this example there is an unrestricted PDF viewer which becomes restricted if it opens the mail.

To provide least privilege, it is desirable to isolate different components of the MUA into different processes so that each component's privileges are minimized. For example, address book management can be put in a separate process which will need to write the address book; the mailer will then only need to read the address book.

Quarantining foreign objects On execution of a PDF viewer, the process is not sandboxed. This non-sandboxed process can read and write USERFILES; if it instead reads a MAIL file then two things happen:

1. The mayFlow's prevent MAIL files from being copied to USERFILES,
2. The minAspects reduces the privilege to interact with non-sandboxed processes (see Section 2.1.3).

Note that the PDF viewer or the MUA takes no action to create a sandbox, rather it is the consequence of the above permissions.

Copy The copy is almost identical to the viewer, the only difference is that the copy can, in addition, create and write mail files. However, the copy is not allowed to perform information flow *between* different labels (note that it is not specified for any mayFlow). Hence, having read MAIL the copy can at most create new MAIL files *or* having read USERFILES can at most create new USERFILES.

Scrubbers Not all email attachments need to be permanently untrusted. For example, a *scrubber*, that creates a

safe version of any MAIL file, may be run to remove dangerous content from a file. Of course, it is not possible to remove all dangerous content from all interpreters and hence an effective scrubber would be with respect to a single (or perhaps related set) of interpreters. The scrubber domain can write files with the USERFILES label.

Certifiers Unlike the scrubber, the certifier can be used when a MAIL file is safe for all uses. It is safe if is created by a trusted system which certifies that it is safe (for example, by providing a digitally signed certificate). The certifier is only able to change the label and read the file, it does not have (and does not need) privileges to modify the file being certified.

2.2.2 Discussion

Two important properties of this mechanism are (1) sandboxes are automatically entered without explicit user action and (2) the taintedness of objects is automatically tracked using LEAP information flow rules. Thus the burden on users is reduced while the system is made more secure.

We have shown just one short example of a dynamic sandbox implemented in LEAP. The LEAP mechanisms are fairly simple and yet are extremely flexible. It would be easy to extend this example to many more helper applications. It is also easy to extend it in other ways such as:

- a sophisticated user might be trusted to determine when it was safe to relabel—but the authorization system could still track which files contained mail, thus relieving the user of that burden.
- a system might have 3 levels, say COREORGANIZATIONALFILES, USERFILES and MAIL, with the requirement that MAIL could never be integrated into the COREORGANIZATIONALFILES.

As always, authorization is a balancing act between flexibility and security, and LEAP provides sufficient flexibility to allow the organization to determine this balance.

3. Implementation

LEAP, as described in the previous section, is used to configure the authorization system. It has three fundamental properties which make it attractive for specification: it is succinct, it is composable, and it is (mostly) stateless[3].

In contrast, access matrix level representations do not have any of these three properties and hence their analysis requires a simulation of state transitions. Moreover, manual changes to the access matrix are tedious. But access matrix implementations have proven very efficient. We have implemented an access matrix based enforcement engine called KernelSec Domains in the Linux Operating System using Linux Security Modules [48].

[3]The permissions themselves are stateless, the only state in the high-level specification are the group memberships.

KernelSec domains are produced automatically from a LEAP specification via a algorithm we call *factoring* [23]. KernelSec Domains are designed to work tightly with LEAP. In particular, group definition, labels, users, and aspects are essentially identical in KernelSec domains as in LEAP. An overview of our system is shown in Figure 3.

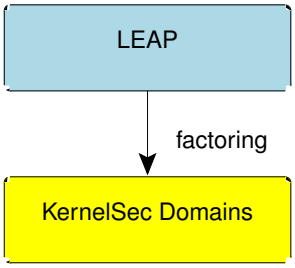

Figure 3. Factoring LEAP specifications into KernelSec domain

3.1. KernelSec issues

Space restrictions preclude a full discussion of the KernelSec mechanisms. We note here only a few relevant issues.

3.1.1 Aspects

KernelSec has implicit permissions based on aspects which are extensions of POSIX implicit permissions. For example, in POSIX, sending a signal from process p_1 to p_2 requires that both processes have the same UID (or that the sender is root). In KernelSec, it requires that p_1's aspect is greater than or equal to p_2's which refines the POSIX rules. (This rule applies not only to signals, but to all communication between processes which POSIX requires to be on behalf of the same user.) Hence a sandboxed process operating at a minimum aspect can only send signals to other processes that share its sandbox (and hence its aspect).

3.1.2 System calls

Only one KernelSec specific system calls (calls from the process to the operating system) is need to support the sandbox semantics. (The remaining semantics are part of the KernelSec domains.)

A process performs a relabel on an object, by invoking the syscall:

```
relabel(objectName obj, label l)
```

which relabels an object `obj` to `l`. For this call to be allowed, assuming that before the call `obj`'s label is l', the domain must have the permission $relabel(l', l)$.

3.1.3 Domains

In KernelSec, domains specify the privileges currently associated with a process. In addition, KernelSec domains can specify actions to occur on the exercise of a permission, which is used to change privileges when reading a MAIL file (thus implementing dynamic information flow restrictions) and to use a minimal aspect when sandboxing a process.

Description	(in clock ticks) Unix	KernelSec	Overhead
minimum viewer read	7,017	8,040	14.58%
minimum viewer invocation	1,115,830	1,154,400	3.46%
client and minimum viewer invocations	2,272,270	2,345,240	3.21%

Table 1. Elapse times (in clock ticks).

3.2. Performance

In this section we present the performance results of executing some micro-benchmarks using the domains described in the previous section. We have begun porting X11 applications to KernelSec. So far these include `xpdf`, `bash` (the bourne again shell), and `thunderbird`. These constitute substantially all of the functionality for executables described here. The performance overhead of KernelSec is negligible for these applications, so we report only micro-benchmarks (for which performance can be seen).

We measure the performance of very small executables, a minimum viewer (corresponding to a PDF viewer) and a client application (corresponding to a MUA). Because the executables are small, the overheads are noticeable vs. the insignificant one on the PDF viewer and MUA.

We measured the elapse times for the following operations:

Jailing minimum viewer on read: This transition happens on the read operation and involves an aspect reduction followed by a domain transition. Since we only measure the elapse time, and the switch happens in the kernel as part of the read operation, the time measured here also includes the time taken to perform the actual read (of 1K bytes).

minimum viewer invocation: Performs a fork-exec of the minimum viewer. This requires a domain transition and various inode permission checks for exec and reading files as well as transiting directories. All the KernelSec security checks happen on the exec operation, the fork does not require any permissions.

client application and minimum viewer invocations: This starts from the shell and does fork-exec of the client application followed by a fork-exec of the minimum viewer. This is essentialy twice the work of the minimum viewer invocation and also, not surprisingly takes twice the time.

We did not measure the scrubber or certifier overhead, as these will be essentially the same as the minim viewer invocation. The measurements taken therefore reflect the entire sandboxing mechanism described in the previous section.

The overhead is fairly modest even in the microbenchmark, a few percent for the larger operations; even the minimum viewer read, which reads 1000 bytes and, in the kernelSec case switches domains is modest at 14.58%. When compared to executing xpdf on this paper, about .2 seconds or 400,000,000 ticks, the increased overhead of 40,000 ticks is insignificant.

4. Related Work

Sandboxing is a form of isolation. The work on isolation can be viewed as taking place in two parts: the isolation of execution environments and the isolation of data (also called information flow). We also address the use of dynamic mechanisms in authorization models.

Isolation The isolation of execution environments occurs through address space separation and restricted interfaces for interacting with the external world. Virtual Machines (VMs), such as Xen [14], VMware [46] and UML [13], provide highly isolated environments in which applications running on different VMs are (ideally) as well isolated as if the applications were running on different hardware. For example, such techniques have been used to create a WebOS [12]. Thus attacks on VMed applications are limited to attacks through the network. But VMs are coarse grained, and the controlled sharing of resources static [36].

Finer grain techniques can either be implemented entirely inside the kernel or via system call interposition. System call interposition techniques [20, 22, 29, 32, 18] are orthogonal to the access control model and have been used to create sandboxes, perform intrusion detection, prevent harmful side-effects of untrusted code, or selectively elevate privileges. Such techniques have also been used to prevent process-subversion attacks which exploit system vulnerabilities [25, 28]. In general, kernel-based mechanism are more efficient while system call interposition mechanisms are easier to implement and are more extensible.

Sandboxing provides protection at the process (process group) granularity. In contrast to VMs, these sandboxes exist within an operating system, and are designed to restrict the address space and interaction of sandboxed applications (consisting of one or more processes) with the rest of the system. Sandboxing can (typically) communicate via networking and, unlike VMs, are asymmetric in that a sandboxed process has very limited visibility or effect outside the sandbox while non-sandboxed processes can access sandboxed processes (e.g., via signals) and files.

The earliest OS-based sandboxing technique appears to be TRON [6], which supports both traditional Unix access controls and TRON capabilities; an operation is allowed only if the process has both of the corresponding TRON and Unix permissions. Tron operates by system call wrappers, which is entered by discretion, and controls file access.

Another early sandboxing technique is Janus [20]. Janus seeks to contain helper applications to browser and mail user agents using a mailcap file to initiate these helper applications. Janus uses system call interposition, is explicitly invoked from user space, and enables modules to define file, network, and interprocess communication. MAPbox [1], built on top of Janus, adds more classes of confinement mechanisms, essentially replacing Janus's programmable module framework with a fixed set of modules. Peterson et al. [31] has a hybrid sandboxing mechanism using kernel-level enforcement while relying on the parent process to confine the child.

WindowBox [3] creates permanent sandboxes which were associated with workspaces. Each workspace could have a different set of privileges, and one workspace could operate on all the others (the one-way property). WindowBox attenuates privileges by access tokens (containing credentials) and access control lists; it has been implemented in the Windows OS kernel.

AppArmor [11] is a sandboxing technique designed to better protect servers. It implements a notion of sub-process protections, based on reducing the privileges during execution of certain code of the process. EVM/SLIM provide file integrity mechanisms and are integrated on top of a TPM mechanism [35]. AppArmor, EVM/SLIM, and KernelSec domains are all implemented on top of LSM.

Ostia is a system call interposition delegation-based sandbox in which the most sensitive system calls, rather than being performed by the application, are delegated to a user space process [19].

Recursive sandboxes provides privilege attenuation in the child processes based on calls that the parent makes [27], thus enabling a process to voluntarily give up privileges.

BSD jails [24] are a sandboxing technique which is closest to VMs. The BSD jails are each allocated their own IP number, and the visibility from the jail to the outside world is severely restricted. BSD jails are explicitly invoked from user space, and provide very little visibility outside the jail.

Many of the sandboxing techniques include resource limits; these are used to prevent denial of service attacks. While these are not currently included in LEAP, it would be trivial to add them.

Another OS-based isolation mechanism is Type Enforcement (TE) [8, 39, 47, 2], based on the access matrix, in which the privileges are based on the domain of the process.

To support least privilege, it is necessary that monolithic applications either be broken up into separate processes for privilege separation to enable the OS to control accesses [33] (e.g., qmail[4], PrivTrans [9], Data Sandboxing [26] or OKWS [15]) or retrofitted with checks into the application as performed by Ganapathy, Jaeger, and Jha for the X-server [17].

Information flow The isolation of data implemented by our dynamic sandboxing is based on information flow techniques. Information flow foundations were established by Bell-LaPadula [4] and later by Biba [7]. Asbestos [15] provides a DAC-based mechanism for information flow based on the decentralized label model [30]. Other models which can support information flow include RBAC [37] and EROS's extended capability model [38]. The above techniques do not change process level labeling, which seems necessary to implement dynamic sandboxing.

In subOS [21] objects whose content is from remote sources are tagged with an immutable sub-user ID denoting the source of the object. When an object with remote content is read, permission is reduced by any permissions the sub-user doesn't have.

Dynamic mechanisms The high level specification of dynamic sandboxes have explicit dynamic mechanisms to reduce privileges on use of permissions and implicit mechanisms based on past reads. We note that these mechanisms take place in the authorization system, not in the user program, and hence cannot be bypassed. TE domain transitions, in contrast, occur only on `exec` and hence are not dynamic. We have described elsewhere how KernelSec can be used to implement groups, information flow, Chinese Wall [34] and dynamic separation of duty [40].

Many special purpose mechanisms have been used for dynamic authorization, but they are each (and collectively) far less general than KernelSec's dynamic mechanisms. In POSIX systems, the setuid-bit on files changes the process' privileges on an `exec` [10]. In TE, domains change on `exec`, although SELinux also allows domain changes to be explicitly requested[5]. Compartmented Mode Workstations (CMW) allows labels to "float up" to higher levels rather than deny access [5] and limits the amount they can float up by specifying the maximum level. LOMAC prevents core filesystem components from being "infected" by untrusted sources [16] by reducing a process' permissions after reading an untrusted source.

A project with similar goals to KernelSec was DTAC which added constraints to the system to provide a more dynamic TE [45, 44]. DTAC was the first OS-based authorization model which could represent dynamic separation of duty.

[4] http://cr.yp.to/qmail

[5] http://www.nsa.gov/selinux/list-archive/0411/9712.cfm

5. Conclusion

Sandboxes have traditionally been used to attenuate the privileges of executables and thereby implement least privileges, thus reducing the dangers posed by applications. Sandboxes provide a one-way protection mechanism. Processes inside a sandbox have very limited visibility outside of the sandbox but the sandbox and its contents are visible to processes external to it. Hence, a process outside a traditional sandbox could copy a file which was downloaded from an untrusted source, after which the sandbox protections would be lost. This is desirable to allow a user to incorporate these outside entities into her activities.

These files pose a danger to applications which later read them. On the other hand, data produced internally by trusted mechanisms should be given greater permissions than those which originate from untrusted sources. Yet traditional sandboxes are oblivious to this distinction.

We introduce *dynamic sandboxes*, give a sample specification in LEAP, and describe their implementation in KernelSec, a kernel-level authorization model implemented in the Linux Kernel. Using KernelSec we track (via labels) those files which come from untrusted sources and using the dynamic domain transitions of KernelSec, automatically enter a dynamic sandbox when an interpreter reads an untrusted entity.

The mechanisms used in KernelSec to implement dynamic sandboxes are general purpose and policy neutral; these mechanisms have uses other than for constructing sandboxes. This generality is important in finding a compact set of building blocks with which to protect systems.

The construction of effective mechanisms to provide the authorization needed so that programs execute with least permissions is not difficult. But providing such mechanisms with sufficiently low complexity that they are used, and used correctly, is indeed challenging. We believe that dynamic protections which adapt to the actions of the users and suitable high-level specifications are keys for dramatically reducing the complexity of using these protections, and thus can usher in a new generation of authorization models which strongly protects the system and its users.

Acknowledgements

Xpdf, thunderbird, and bash were ported by Saurabh Abichandani and Shuxia Feng. We would like to thank Saurabh Abichandani, Shuxia Feng, Jorge Hernandez-Herrero for their helpful comments. We would also like to thank the anonymous reviewers for their feedback.

References

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

[2] L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker, and S. A. Haghihat. Practical domain and type enforcement for UNIX. In *Proc. IEEE Symp. Security and Privacy*, pages 66–77, Oakland, CA, 1995.

[3] D. Balfanz and D. R. Simon. WindowBox: A simple security model for the connected desktop. In *Proceedings of the 4th USENIX Windows Systems Symposium (WSS-00)*, pages 37–48, Berkeley, CA, Aug. 3–4 2000. The USENIX Association.

[4] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations and model. Technical Report M74-244, Mitre Corporation, Bedford MA, 1973.

[5] J. L. Berger, J. Picciotto, J. P. L. Woodward, and P. T. Cummings. Compartmented mode workstation: Prototype highlights. *IEEE Transactions on Software Engineering*, 16(6):608–618, 1990. Special Section on Security and Privacy.

[6] A. Berman, V. Bourassa, and E. Selberg. TRON: Process-specific file protection for the UNIX operating system. In *Proceedings of the USENIX 1995 Technical Conference*, pages 165–175, New Orleans, LA, USA, Jan. 16–20 1995.

[7] K. Biba. Integrity considerations for secure computer systems. Technical Report TR-3153, MITRE Corp, Bedford, MA, 1977.

[8] W. E. Boebert and R. Kain. A practical alternative to hierarchical integrity policies. In *8th National Computer Security Conference*, pages 18–27, 1985.

[9] D. Brumley and D. X. Song. Privtrans: Automatically partitioning programs for privilege separation. In *USENIX Security Symposium*, pages 57–72, 2004.

[10] H. Chen, D. Wagner, and D. Dean. Setuid demystified. In *Proc. of the USENIX Security Symposium*. USENIX, 2002.

[11] C. Cowan, S. Beattie, G. Kroah-Hartman, C. Pu, P. Wagle, and V. Gligor. Subdomain: Parsimonious security server. In *14th Systems Administration Conference (LISA 2000)*, pages 355–367, New Orleans, LA, 2000.

[12] R. S. Cox, S. D. Gribble, H. M. Levy, and J. G. Hansen. A safety-oriented platform for web applications. In *IEEE Symposium on Security and Privacy*, pages 350–364. IEEE Computer Society, 2006.

[13] J. Dike. User-mode Linux. In USENIX, editor, *Proceedings of the 5th Annual Linux Showcase and Conference*. USENIX, Nov.5–10 2001.

[14] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the art of virtualization. In *In Proceedings of the ACM Symposium on Operating Systems Principles, October 2003.*, 2003.

[15] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and event processes in the asbestos operating system. *SIGOPS Oper. Syst. Rev.*, 39(5):17–30, 2005.

[16] T. Fraser. LOMAC–low water-mark mandatory access control for Linux. In *Proc. of the USENIX Security Symposium*, Washington D.C., 1999.

[17] V. Ganapathy, T. Jaeger, and S. Jha. Retrofitting legacy code for authorization policy enforcement. Technical Report 1544, University of Wisconsin–Madison, Computer Science Department, nov 2005. Decscribes semi-automatic techniques for retrofitting an X-server with a mechanism which can.

[18] T. Garfinkel. Traps and pitfalls: Practical problems in in system call interposition based security tools. In *Proc. Network and Distributed Systems Security Symposium*, February 2003.

[19] T. Garfinkel, B. Pfaff, and M. Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *Proc. Network and Distributed Systems Security Symposium*, February 2004.

[20] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications (confining the wily hacker). In *Proc. of the USENIX Security Symposium*, San Jose, Ca., 1996.

[21] S. Ioannidis, S. M. Bellovin, and J. Smith. Sub-operating systems: A new approach to application security. http://www.research.att.com/ smb/papers/subos.ps, Nov. 2001. draft, sandbox.

[22] K. Jain and R. Sekar. User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. In *NDSS*, 2000.

[23] K. Kahley, M. Radhakrishnan, and J. A. Solworth. Factoring high level information flow specifications into low level access controls. In *IEEE Workshop of Information Assurance*, Apr. 2006.

[24] P.-H. Kamp and R. N. M. Watson. Jails: Confining the omnipotent root. In *SANE 2000*. NLUUG, 2000.

[25] G. S. Kc and A. D. Keromytis. e-nexsh: Achieving an effectively non-executable stack and heap via system-call policing. In *ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference*, pages 286–302, Washington, DC, USA, 2005. IEEE Computer Society.

[26] T. Khatiwala, R. Swaminathan, and V. N. Venkatakrishnan. Data sandboxing: A technique for enforcing confidentiality policies. In *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference*, pages 223–234, Washington, DC, USA, 2006. IEEE Computer Society.

[27] A. Kurchuk and A. D. Keromytis. Recursive sandboxes: Extending systrace to empower applications. In *SEC*, pages 473–488, 2004.

[28] W. Li, L. chung Lam, and T. cker Chiueh. How to automatically and accurately sandbox microsoft iis. In *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference*, pages 213–222, Washington, DC, USA, 2006. IEEE Computer Society.

[29] Z. Liang, V. N. Venkatakrishnan, and R. Sekar. Isolated program execution: An application transparent approach for executing untrusted programs. In *ACSAC*, pages 182–191, 2003.

[30] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *Software Engineering and Methodology*, 9(4):410–442, 2000.

[31] D. S. Peterson, M. Bishop, and R. Pandey. A flexible containment mechanism for executing untrusted code. In USENIX, editor, *Proc. of the USENIX Security Symposium*, pages 207–225, Berkeley, CA, USA, 2002. USENIX.

[32] N. Provos. Improving host security with system call policies. In *Proceedings of the 12th USENIX Security Symposium*, pages 257–272. USENIX, Aug. 2003.

[33] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *Proceedings of the 12th USENIX Security Symposium*, pages 231–242. USENIX, Aug. 2003.

[34] M. Radhakrishnan and J. A. Solworth. Application security support in the operating system kernel. In *ACM Symposium on InformAtion, Computer and Communications Security (AsiaCCS'06)*, pages 201–211, Taipei, Taiwan, Mar. 2006.

[35] D. Safford, M. Zohar, and R. Sailer. EVM, SLIM, IMA. http://lwn.net/Articles/160126/, nov 2005.

[36] R. Sailer, T. Jaeger, E. Valdez, R. Cáceres, R. Perez, S. Berger, J. L. Griffin, and L. van Doorn. Building a MAC-based security architecture for the Xen open-source hypervisor. In *ACSAC*, pages 276–285. IEEE Computer Society, 2005.

[37] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[38] J. S. Shapiro and S. Weber. Verifying the EROS confinement mechanism. In *Proc. IEEE Symp. Security and Privacy*, pages 166–176, 2000.

[39] S. Smalley, C. Vance, and W. Salamon. Implementing SELinux as a Linux security module. Report #01-043, NAI Labs, Dec. 2001. Revised April 2002.

[40] J. A. Solworth. Approvability. In *ACM Symposium on InformAtion, Computer and Communications Security (AsiaCCS'06)*, pages 231–242, Taipei, Taiwan, Mar. 2006.

[41] J. A. Solworth and R. H. Sloan. Decidable administrative controls based on security properties, 2004. Available at http://www.rites.uic.edu/ solworth/kernelSec.html.

[42] J. A. Solworth and R. H. Sloan. A layered design of discretionary access controls with decidable properties. In *Proc. IEEE Symp. Security and Privacy*, pages 56–67, 2004.

[43] J. A. Solworth and R. H. Sloan. Security property-based administrative controls. In *Proc. European Symp. Research in Computer Security (ESORICS)*, volume 3139 of *Lecture Notes in Computer Science*, pages 244–259. Springer, 2004.

[44] J. Tidswell and T. Jaeger. An access control model for simplifying constraint expression. In *Proc. ACM Conference on Computer and Communications Security (CCS)*, pages 154–163, 2000.

[45] J. F. Tidswell and T. Jaeger. Integrated constraints and inheritance in DTAC. In *Proc. of the ACM Workshop on Role-Based Access Controls (RBAC)*, pages 93–102, 2000.

[46] C. Waldspurger. Memory resource management in VMware ESX server. In *Fifth Symposium on Operating Systems Design and Implementation*, Dec. 2002.

[47] R. Watson. TrustedBSD: Adding trusted operating system features to FreeBSD. In *USENIX Technical Conference*, Boston, MA, 2001.

[48] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux Security Modules: General security support for the Linux Kernel. In *Proc. of the USENIX Security Symposium*, San Francisco, Ca., 2002.