Mandatory Security and Performance of Services in Asbestos

by

David Patrick Ziegler

Bachelor of Science in Computer Science and Engineering,
Massachusetts Institute of Technology (2004)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2005

Author .....
Department of Electrical Engineering and Computer Science
May 23, 2005

Certified by .....
M. Frans Kaashoek
Professor
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Mandatory Security and Performance of Services in Asbestos

by

David Patrick Ziegler

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 2005, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis presents the design and implementation for several system services, including network access and database storage, on a new operating system design, Asbestos. Using the security mechanism provided by Asbestos, Asbestos labels, these services are used to support the construction of secure Web applications. The network and database services serve as the foundation for a Web server that supports mandatory security policies, such that even a compromised Web application cannot improperly disclose private data. The methods used in this thesis allow Web application developers to be freed from worries about flawed applications, if developers are willing to place trust in the underlying services used.

Thesis Supervisor: M. Frans Kaashoek
Title: Professor

Acknowledgments

I would like to thank my thesis advisor, Frans Kaashoek, for his guidance, advice, and support on this thesis. He accepted me as a member of the Asbestos project, helped me choose the focus of the thesis, and was a cheerful supporter the whole way.

Many people in the Asbestos project and the PDOS group at MIT have supported me during my work. In particular, Max Krohn has been a backbone of the Asbestos project, providing a strong motivating application. He has consistently helped me solve problems without complaints. Cliff Frey has discussed ideas with me without end. Having him next to me has been a strong motivator.

I also want to thank my parents. They couldn't understand what I was talking about half the time, but have supported me without fail for my entire life. I wouldn't be where I am without them.

Finally, thanks to my wonderful wife, Sarah. She has been my best friend and companion through MIT. She takes care of me and supports me in everything I do. Thank you.

This research was supported by DARPA grants MDA972-03-P-0015 and FA8750-04-1-0090, and by joint NSF Cybertrust/DARPA grant CNS-0430425.

Portions of this thesis are adapted from and/or contain text originally published in

> Max Krohn, Petros Efstathopoulos, Cliff Frey, Frans Kaashoek, Eddie Kohler, David Mazières, Robert Morris, Steve VanDeBogart, and David Ziegler. Make least privilege a right (not a privilege). In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems*, June 2005.

Portions of this thesis also are adapted from and/or contain text written in collaboration with Petros Efstathopoulos, Cliff Frey, Frans Kaashoek, Eddie Kohler, Max Krohn, David Mazières, Robert Morris, and Steve VanDeBogart.

Contents

Chapter 1

Introduction

Computer systems today have a poor record in security. We routinely learn of exploits of buffer overflows [43, 42, 41, 44, 46], vulnerabilities in the Java virtual machine [45], and improper allocation and deallocation of memory [47], to name a few. Typically, complete compromise of the system is the result.

In all these cases, complete failure occurs because applications have rights they do not require. In normal execution, these rights are never expected to be used, yet, whether from ease of programming, lack of system support, or simple mistakes, they remain. However, once a program is compromised or malfunctions, it may accidentally or maliciously abuse the rights it has. The *principle of least privilege* has been known for decades — that a process should be given all of the rights necessary for it to perform its job and no more [36].

We believe that the only viable means of improving security is to design systems that effectively support the principle of least privilege. For example, an email reader should be able to confine an executable attachment by only giving it access to a display window and perhaps a temporary file system. A Web application should be able to ensure that one user's private data cannot be sent to another user's browser by a faulty Web server.

This thesis considers the goal of creating a more secure Web server and the components needed for such an application. Using a novel operating system architecture, Asbestos, and a secure Web server design, the design and implementation for several

supporting services are considered, including a persistent security service, a network daemon, and a database. Using these components and Asbestos, we can ensure that one user's data cannot be exposed to other users.

1.1 Asbestos: A New Operating System

Asbestos combines the advantages of capability-based and mandatory-access systems. Access in Asbestos is based on a simple primitive, *Asbestos labels*, that may be used to implement a variety of discretionary and mandatory access policies in a completely decentralized fashion. Any process may create an access control space, represented by a handle, and control the policies applied relative to that space.

Developers can secure their applications by labeling processes with respect to handles. As processes communicate, they may *contaminate* each other with respect to some handles, allowing the kernel or other processes to track and control data flow. This property allows developers to enforce mandatory access control.

Using a handle to denote a security arena, developers can raise or lower its security level with respect to the handle to denote decreased or increased rights, respectively. Alternately, developers can change the security level with respect to the handle to mark a process's contamination level (i.e., whether it has been exposed to sensitive data).

To extend the security model to operate within a single process, Asbestos introduces a method for labeling individual regions of memory, allowing a process to operate on sensitive data without risking exposure of that data to unprivileged entities. By using a set of system calls, a process can save its current state, operate on sensitive data in a protected region of memory, and return to the saved state in a secure manner.

1.2 Motivation: A Better Web Server

Web servers are increasingly becoming the preferred mechanism to provide access to databases of private information. Banks, universities, and corporations alike allow users to view and modify personal data through Web sites. In this role, a Web server and application should provide users with their data without exposure to other users. Unfortunately, as Web sites grow in size and complexity, Web developers are more likely to forget or misapply access controls.

Even if developers correctly implement access controls, they must rely on the large set of software that a Web application uses to perform properly. In a typical system, the trusted computing base (TCB) includes the operating system (e.g., Linux [48]), the system library (e.g., `libc` [11]), the Web server (e.g., Apache [12]), Web server modules (e.g., PHP [14] or SSL [33]), and the Web application itself (e.g., `account-balance.php`). A vulnerability in any of these modules may allow an attacker access to private data.

Software such as the OK Web server (OKWS) [16] demonstrates that it is possible to contort the existing UNIX interface [49] to achieve some security goals, such as isolation. In OKWS, each logical Web service (e.g., `account-balance` or `transfer-funds`) runs as a separate process in its own address space. If an attacker compromises and controls `account-balance`, he cannot arbitrarily use other services as well (e.g., `transfer-funds`).

Yet OKWS does not provide all the isolation customers may desire. If an attacker controls `account-balance`, the structure of OKWS allows the attacker to have access to *all* account balances, rather than just his own. If a single service is compromised, the attacker gains all privileges of that service.

With Asbestos labeling, developers can guarantee that even if a service becomes compromised, the kernel ensures that an attacker can have access only to his own information. Using labeling, a Web server becomes contaminated after accessing a client's data; that contamination lets developers prevent a client's information from being transmitted over a network connection for a different client.

1.3 Challenges

Several services are required to support a version of OKWS for Asbestos providing these stronger guarantees. Some form of network access is a prerequisite for any Web server. In order to store data in a structured format, some form of persistent database access is appropriate. The focus of this thesis is on the design and implementation for services supporting an OKWS-like Web server.

Several challenges arise in using Asbestos as a foundation for these services. One of the more difficult ones is matching up the data structures used by services with the security primitives Asbestos provides. A logical mapping from service data structures to handles is necessary, but as handles are ephemeral, a persistent "handle" becomes necessary.

By creating services that can be used by multiple processes, we open covert communication channels. Although two processes A and B might be unable to communicate directly, a service S could act as an unwitting intermediary through some unintended communication loophole. Care is required to avoid the creation of such channels.

Another difficult lies in the use of **vm_save** and **vm_restore**, the two system calls Asbestos provides for labeling memory regions. In order to close certain covert channels, it is necessary that an application can begin processing only once it has received a message. For many services, the model of waiting for a request, operating on that request, and returning back to a starting point to wait again is sufficient; however, that model may be insufficient for some processes (e.g., processes that handle events other than client requests).

Finally, over-contamination becomes a problem for services that handle data from multiple sources. After a service has operated on sensitive data from process A, it must be able to process requests from another process B without marking B as having seen A's data. It becomes difficult to provide the mandatory security guarantees we would like while supporting this model of operation.

1.4 Approach

Several different techniques are used in addressing the challenges in the construction of these services. For persistent handles, we introduce *authenticated identifiers* that processes may use across system reboots to uniquely identify protected data. Along with the introduction of authenticated IDs, we discuss a service to manage their creation and distribution.

Several steps are taken to avoid the creation of covert channels. As any process can request a handle, processes might use the rate of handle creation to communicate data; to counter this, we encrypt handle values making them effectively random. Authenticated IDs are similarly accessible to any application; we encrypt them as well, to mask any information about the creation rate. Database query results may leak information to clients; we allow creators to choose the contamination that a process receives through such actions.

Experience with Asbestos has shown that care is required in the security level at which a handle is distributed; small changes can have profound impacts upon the security policy that results. For example, increasing the level at which a process is contaminated at by one unit can effectively render it unable to communicate. When the network or database distributes sensitive data, it does so at the proper level to ensure that the desired security is achieved.

The problem of long-term over-contamination is intermingled with the difficulties of **vm_save** and **vm_restore**. Memory region labeling was developed to help address this problem, allowing a single service to handle data from different clients. However, we find that memory region labeling has difficulties for some service types, including the ones considered in this thesis. Therefore, we consider solutions to decontaminate processes securely, including handle ownership and third-party decontamination, and the costs of such solutions.

1.5 Outline

The remainder of this thesis is organized as follows. First, we summarize the design of Asbestos and its security model. Next, we consider the design and implementation for several components:

- the authenticated identifier service, which implements the authenticated ID design;
- the network service, which provides TCP/IP communication [27, 29] to other Asbestos processes;
- the database service, which supports a large subset of the SQL standard [1] in a secure manner; and
- the complete Web application system, using these services and a modified version of OKWS, including the security properties developers and system administrators obtain.

Then, we evaluate the performance of the system, including individual components as appropriate, and consider future improvements that may be made. Finally, we describe related work and conclude with a review of the contributions of this thesis.

Chapter 2

Background: The Asbestos Security Model

In this chapter we describe the properties of Asbestos, including the primitives it supports for building secure programs. Asbestos is, at its heart, an operating system based on message passing. All communication between user-level processes or a process and the kernel is through *messages*, sent to communication *handles*. First, we discuss the basic primitives for communication between processes, handles and messages. We describe the details of *labeling*, the method for controlling the flow of protected information. Finally, we discuss methods for building secure applications using these primitives.

2.1 Handles

An Asbestos handle is an endpoint for communication between processes or a process and the kernel. However, a handle may also be used as a capability or a kernel-protected secret, as we describe below. A handle is simply a large and unique random number. The kernel maintains information describing a process's privileges with respect to the handle (e.g., ownership).

To create a handle, an application invokes the system call **new_handle**, which generates a new handle, grants the caller ownership of the handle, as described be-

low, and optionally sets up a message queue to receive messages sent to the handle. The handle returned is unique (i.e., the handle has not been given out previously), ephemeral (i.e., the handle is invalid after a reboot), and unpredictable (i.e., the number returned is apparently unrelated to the previous handle value).

In the Asbestos kernel, a routing table stores the device responsible for a handle; messages sent to a handle are delivered to that device, usually a process's message queue. The kernel also maintains per-process labels used to store security information, as discussed below in Section 2.3.

2.2 Messages

As mentioned previously, all communication in Asbestos is through messages. Asbestos defines a standard message type to which all messages must conform, containing six components:

- a *destination handle*, where the message is delivered,
- a *message type*, which defines the operation class requested,
- a *message code*, used as an argument for request messages and as an error code for reply messages,
- an *ID*, used to match requests with replies,
- an optional *reply handle*, which specifies the recipient of any reply message, and
- an optional *payload*, which contains any additional data for the message.

The message types in Asbestos include:

LOOKUP Find an entry in a directory or directory-like object. The payload typically contains the name of the entry to look up. Replies have type LOOKUP_R (the convention for all message types); the payload typically contains the handle values for the objects.

P, Q	Processes	$\star, \mathbf{0}, \mathbf{1}, \mathbf{2}, \mathbf{3}$	Label levels, in increasing order
h, *dest*	Handles	L, C, D, V, E	Labels (functions from handles to levels)

$L_1 \leq L_2$	Label comparison: true if $\forall h, L_1(h) \leq L_2(h)$	P_{S}	Process P's send label
$\max(L_1, L_2)$	Maximum label: $\{h\, k \mid k = \max(L_1(h), L_2(h))\}$	P_{R}	Process P's receive label
$\min(L_1, L_2)$	Minimum label: $\{h\, k \mid k = \min(L_1(h), L_2(h))\}$	h_{R}	Handle h's receive label
$\mathrm{owned}(L)$	Owned-handles label: $\{h \star \mid L(h) = \star\} \cup \{h\, \mathbf{3} \mid L(h) \neq \star\}$		

Figure 2-1: Asbestos notation.

READ, WRITE Read data from an object, or write data to an object; the payload contains the data.

CONTROL A catch-all message for other types of access. The message code specifies any further information about the operation requested.

In order to send a message, a process calls **send**, described in further detail below.

2.3 Labels

Applications specify access and information flow control in Asbestos through a single primitive, labels. Labels are flexible enough to implement a wide range of discretionary and mandatory access control policies. Asbestos labels support several novel features, including temporary restrictions that help implement discretionary policies, decentralized declassification [22], and labeling of regions of memory within a single address space.

In Asbestos, a label is a function from handles to *levels*, members of the ordered set $\{\star, \mathbf{0}, \mathbf{1}, \mathbf{2}, \mathbf{3}\}$ (where $\star < \mathbf{0} < \cdots < \mathbf{3}$). We write labels using set notation, as $\{h_1\, \mathbf{0}, h_2\, \mathbf{1}, \mathbf{2}\}$. The default level, $\mathbf{2}$ in this case, appears at the end of the list and applies to all handles not explicitly listed.

Labels may be compared; we say for two labels L_1 and L_2 that

$$L_1 \leq L_2 \;\text{ iff }\; L_1(h) \leq L_2(h) \;\text{ for all } h.$$

We define similar comparisons $\max$ and $\min$ (see Figure 2-1).

send$(dest, C_S, D_S, V, D_R, \text{data})$
Let Q be *dest*'s controlling process
Let $E_S = \max(P_S, C_S)$
Let $Q_{newR} = \max(Q_R, D_R)$
Let $E_R = \min(Q_{newR}, dest_R, V)$
Let $Q_{own} = \text{owned}(Q_S)$
Requirements:
(1) $E_S \leq E_R$
(2) $D_R \leq dest_R$
(3) If $D_S(h) < \mathbf{3}$, then $P_S(h) = \star$
(4) If $D_R(h) > \star$, then $P_S(h) = \star$
Effects:
Grant D_S, contaminate with E_S, then restore owned handles
$Q_S \leftarrow \max(\min(Q_S, D_S), E_S)$
$Q_S \leftarrow \min(Q_S, Q_{own})$
$Q_R \leftarrow Q_{newR}$

new_handle(L)
Let h be an unused handle
Effects:
$h_R \leftarrow L$
$h_R(h) \leftarrow \mathbf{0}$
$P_S(h) \leftarrow \star$
Return h

set_handle_label$(dest, L)$
Requirement:
dest was created by P
Effect:
$dest_R \leftarrow L$

Figure 2-2: Some Asbestos label operations. P is the calling process.

2.3.1 Process Labels

The most basic use of labels is in processes, where they identify a process's current access restrictions in its *send label* P_S and the maximum restrictions a process is willing and able to accept in its *receive label* P_R. Asbestos performs an access check on every message send:

$$P_S \leq Q_R.$$

That is, a process P can only send a message to Q if P's send label is less than or equal to Q's receive label. If this check succeeds, the message is delivered. This flow of information *contaminates* Q with the same restrictions P has:

$$Q_S \leftarrow \max(Q_S, P_S)$$

The access check and contamination operation form the basis of the Asbestos security policy. See Figure 2-2 for more details.

An important component of Asbestos labeling is that the default send and receive

labels have different default values. The default send label is $\{\mathbf{1}\}$, and the default receive label is $\{\mathbf{2}\}$. Importantly, the default label values are in the middle of the label ordering; $\mathbf{0}$ and $\star$ are less than both defaults, and $\mathbf{3}$ is greater.

Because of this asymmetry, flexible isolation schemes are possible. An application can use an arbitrary handle h to prevent a process P from sending messages to a process Q in three distinct ways:

	A	B	C
P_S	$\{h\,\mathbf{3}, \mathbf{1}\}$	$\{\mathbf{1}\}$	$\{h\,\mathbf{2}, \mathbf{1}\}$
Q_R	$\{\mathbf{2}\}$	$\{h\,\mathbf{0}, \mathbf{2}\}$	$\{h\,\mathbf{1}, \mathbf{2}\}$

In scenario A, the application sets $P_S(h)$ to $\mathbf{3}$. This prevents the communication with Q as intended and additionally restricts P to be unable to send messages to any other process (except one with receive label $\{h\,\mathbf{3}\}$). Because increasing a receive label makes the system more permissive, such an action requires special privilege, as discussed in Section 2.3.3.

In scenario B, the application modifies Q instead, by setting $Q_R(h)$ to $\mathbf{0}$. This instead restricts the processes that may send to Q, as any process wanting to send to Q would need $\{h\,\mathbf{0}\}$ in its send label.

These two isolation mechanisms, similar to those available in most mandatory access control systems, limit P's ability to communicate with Q by limiting either P or Q's ability to communicate with anyone. However, the label asymmetry in Asbestos allows a more flexible policy involving both P and Q, as in scenario C. In this case, the application sets $P_S(h)$ to $\mathbf{2}$ and $Q_R(h)$ to $\mathbf{1}$. Thus, P is unable to send messages to Q. P can communicate with other processes, but as it does so, it will contaminate those processes with the send label $\{h\,\mathbf{2}\}$. Thus, any process X cannot send a message to Q if it has received a message from P, directly or indirectly. Any process that does not wish such contamination can set its receive label default to $\{\mathbf{1}\}$. Since lowering a receive label makes the system more restrictive, this requires no special privilege.

2.3.2 Effective Labels

Asbestos allows processes to further restrict access relative to their send and receive labels by using *effective labels*. When a message is sent, the sender may use a *contamination label* C_S and a *verification label* V. These labels are used to construct effective send and receive labels E_S and E_R, as follows:

$$E_S = \max(P_S, C_S), \quad E_R = \min(Q_R, V).$$

The message is delivered only if $E_S \leq E_R$. When using a contamination label, E_S is used to contaminate the receiver, rather than P_S. When using a verification label, V is reported to the receiver when the message is delivered.

To see how these features can be used, consider a multi-user file server. For each user, an "identity handle" u_I and a "contamination handle" u_C are allocated. Any process that speaks for u has send label $\{u_I\,\mathbf{0}\}$ and receive label $\{u_C\,\mathbf{3}\}$.

When the file server sends data exclusively for u, it sets C_S to $\{u_C\,\mathbf{3}\}$. Only processes with receive label $\{u_C\,\mathbf{3}\}$ can receive such messages, and once received, their send labels rise to $\{u_C\,\mathbf{3}\}$. This restricts such processes from communication with non-u processes, which have receive label $\{u_C\,\mathbf{2}\}$.

When a process sends data to the file server, it sets V to $\{u_I\,\mathbf{0}\}$ (or $\{u_I\,\star\}$). The file server checks V, accepting the message only if $V(u_I) \leq \mathbf{0}$. In this way, a process can prove that it speaks for u.

2.3.3 Ownership

The special level $\star$ allows processes to distribute handle access and declassify information in a decentralized way. Any process with $P_S(h) = \star$ is said to *own* handle h, and is given two additional privileges:

- P cannot be contaminated by other processes with respect to h. Even if it receives a message with $E_S = \{h\,\mathbf{3}\}$, P's send label remains $\{h\,\star\}$.
- P can *decontaminate* other processes by lowering their send labels or raising

their receive labels with respect to h. This allows P to make the system more permissive with respect to h.

To support decontamination, Asbestos provides two *decontamination labels* with every message, D_S and D_R. The D_S label is used to make the receiver's send label more permissive by lowering some of its handle levels. The D_R label makes the receiver's receive label more permissive by raising some of its handle levels.

This privileged relaxing of permissions is modeled by $P_S(h) = \star$; to loosen restrictions on another process with respect to some handle, a process must own that handle. When the kernel delivers a message, it checks that whenever $D_S(h) < \mathbf{3}$ or $D_R(h) > \star$, the process has $P_S(h) = \star$. Assuming the message is delivered, the kernel sets

$$Q_S \leftarrow \max(\min(Q_S, D_S), E_S) \text{ and}$$
$$Q_R \leftarrow \max(Q_R, D_R).$$

Decontamination of receive labels is a particularly sensitive operation; a decontaminated process may receive a message that later increases its send label, limiting its ability to communicate. To prevent processes from becoming decontaminated unwillingly, Asbestos can control the messages processes are permitted to receive through their *handle labels*. Each handle has its own label, which may be arbitrarily set by the handle's controlling process. The handle label is used to additionally restrict the effective receive label for messages sent to that handle. If a process P sends a message to Q at handle *dest*, the effective receive label is

$$E_R = \min(Q_R, \mathit{dest}_R, V).$$

With handle labels, the kernel additionally checks that $D_R \leq \mathit{dest}_R$, allowing a process full control over the contamination or decontamination it is willing to accept.

2.3.4 Memory Region Labeling

As described so far, labels apply at the granularity of an entire process. From the operating system's perspective, this is the only point at which a boundary can be drawn, as a process's internals are a black box. Unfortunately, this limitation makes it impossible to run untrusted processes that speak for different users at different times, a class of processes that includes most services. One option is for the operating system to trust the application to keep user data internally isolated. Alternately, programming language-level mechanisms can enforce policies at a much finer granularity, but these mechanisms limit language choices available to programmers and break down when communicating with processes that are implemented in different languages.

Asbestos uses a different approach that allows a single untrusted process to contain any number of independent *memory regions*. Asbestos enforces isolation between these regions at the granularity of memory pages. To do so, Asbestos provides three system calls: **vm_save**, **vm_restore**, and **page_taint**. A process begins by allocating a memory region using **page_taint**, which marks existing pages with a given send label.

When a process calls **vm_save**, the current page table and register state are saved and the process waits until a message arrives. When a message arrives, the kernel determines what label changes are induced by message delivery and updates the process's page tables to give appropriate access to memory (read/write, copy-on-write, or none). When the process calls **vm_restore**, the updated page table is discarded and the process jumps back to the **vm_save** system call.

Chapter 3

Authenticated Identifier Service

As discussed in Section 2.1, handles are ephemeral; when the system reboots, all handles disappear. Yet at the same time, applications need to store some security information persistently; when the Web server stores a client's data, it should label that data as belonging to the client. The relationship between these handles and other processes must be maintained across reboots.

In this chapter, we discuss the design and implementation of `authd`, a process that maintains persistent handle mappings. `authd` provides large numbers — *authenticated IDs* — that persist across reboots, unlike handles. Any application may contact `authd` to determine the handle value associated with a authenticated identifiers; processes may provide extra information in order to gain certain privileges with respect to the authenticated ID. In order to ensure agreement among all processes, `authd` guarantees that during one boot cycle of the system, the handle associated with an authenticated ID is unique and constant.

3.1 Design

3.1.1 Maintaining Authenticated Identifiers

The simplest solution for `authd` to maintain authenticated identifiers is to distribute increasing integers to processes that request an authenticated ID. Whenever a new

authenticated ID is requested, the new value is written to disk. When the system reboots, `authd` can start allocating new authenticated IDs starting after the value on disk.

Such a scheme allows applications a covert channel through which they can communicate surreptitiously (as all applications can request authenticated IDs). By observing the rate at which authenticated IDs are allocated, two applications can communicate that otherwise would not be permitted to do so.

To solve this problem, `authd` encrypts authenticated IDs as they are given to applications. Internally, `authd` uses increasing integers as authenticated IDs. When an authenticated ID is transmitted to or from `authd`, `authd` automatically decrypts or encrypts the client's value, respectively. This eliminates the covert channel described above.

Next, `authd` must maintain a mapping between authenticated IDs and handles. To do so, `authd` maintains a simple database filled lazily. During a single boot of the system, we expect that only a fraction of all authenticated IDs ever allocated will be used. The memory cost of storing handles for all authenticated IDs may be unacceptable, so `authd` stores only those are in use since boot.

3.1.2 Mandatory Security Through Contamination

Given authenticated identifiers, applications may be written to enforce mandatory security policies. Consider a Web server providing access to private data to several different users. We would like to prevent a compromised Web server from allowing attackers to view others' data. To do so, the Web server can associate an authenticated ID with the user's data.

For every user, a persistent record is maintained in a database. Included in that record is an authenticated ID specific for that user. When the Web server accesses authenticated data for a client, the database arranges for the Web server to become contaminated with the handle associated with the authenticated ID for that user. At this point, the Web server will further contaminate any process it speaks with, directly or indirectly, with the handle associated with the user. Using this contami-

nation, the database can enforce a variety of mandatory access control schemes on that user's data.

3.1.3 Contamination and Services

Unfortunately, the use of authenticated IDs as described thus far presents a large problem. Contamination of services can never decrease, even for trusted services; there is no mechanism for a service to decide that data may be declassified. Consider the database maintaining authenticated IDs for records. As soon as the database asks `authd` about an authenticated ID for a single record, it becomes permanently contaminated with that record's authenticated ID A_1. When a request arrives for a different record and the database looks up the new authenticated ID, it becomes additionally contaminated with a second record's authenticated ID A_2. Any message from the database to a client will contaminate the recipient with respect to *two* records, A_1 and A_2, rather than only the one requested. To solve this problem, we need a mechanism similar to level $\star$ for handles; we need a way to demonstrate "ownership" of an authenticated ID.

To provide ownership, we allow a process to optionally associate an *access* and an *owner* password with an authenticated ID. This complicates `authd`, as now passwords must be stored with authenticated IDs on disk. Additionally, we associate two handles with each authenticated ID. The first handle is the *contamination* handle, used for contamination of processes as described previously in Section 3.1.2. The second handle is the *identity* handle. The identity handle is used to demonstrate that a process "speaks for" an authenticated ID. As described in Section 2.3.2, the identity handle can be presented in a verification label to prove that a process speaks for an authenticated ID.

When a process requests information about an authenticated ID, it may ask for one of three things:

1. The contamination and identity handles as numbers, without any granting or contamination.

2. Contamination with the contamination handle at **3** and granting of the identity handle at $\star$.

3. Granting of both the contamination and identity handles.

No special privilege is necessary to request the first; any process is permitted to determine the handle values associated with a authenticated ID. This allows a process to verify a handle presented in a verification label without requiring special privilege or accepting contamination.

A process may request the second and become willingly contaminated with the authenticated ID in exchange for the right to speak for that authenticated ID, but only if it presents the proper "access" password in the request. This allows authenticated ID creators to choose the distribution policy for access to the authenticated ID.

Finally, a process may request the third, but will receive it only if it presents the proper "owner" password for the authenticated ID. This solves the ownership problem above. Now, a process that creates an authenticated ID may gain the right to decontaminate other processes with respect to the contamination handle, so that the flow of contamination may be stopped. In the database example above, now the database (which we assume for the moment creates the authenticated IDs) can possess the contamination handle at $\star$, allowing it to remain uncontaminated.

3.2 Implementation

To perform its service, `authd` must maintain two types of information:

- The persistent record of all authenticated IDs that have been created, including the authenticated ID and passwords. This must persist across reboots.

- The in-memory record of current authenticated IDs, containing the authenticated ID, the contamination handle, and the identity handle. This should be destroyed after each reboot.

For the persistent record, `authd` maintains a simple SQLite database [24] on disk containing several items; for each authenticated ID that has been created, `authd` stores the authenticated ID and the access and owner passwords. `authd` can determine the maximum authenticated ID (and thus, the next ID value) by performing a simple database query.

For the in-memory record of currently used authenticated IDs, `authd` uses another SQLite database stored only in memory. When an authenticated ID is first requested, `authd` checks if the authenticated ID is in the table. If not, `authd` creates the contamination and identity handles, storing the authenticated ID and handle values in the in-memory database. Once it has located the entry for the authenticated ID, `authd` checks for a password in the request.

In an application's request, it specifies which of the three actions of Section 3.1.3 it prefers:

- Choosing action 1, an application receives the contamination and identity handles as numbers, without any granting or contamination, and with no privilege required.

- Choosing action 2, an application provides the access password. If correct, the application receives contamination with the contamination handle at $\mathbf{3}$ and is granted the identity handle at $\star$. Otherwise, `authd` responds as in action 1.

- Choosing action 3, an application provides the owner password. If correct, `authd` grants the application both the contamination and identity handles at $\star$. Otherwise, authd responds as in action 1.

3.3 Discussion

The most needed feature in `authd` is different methods of demonstrating ownership. While the access/owner password is sufficient, more exotic or secure schemes might be desirable. A feature that many might miss in the current design is a method for

changing the access or owner password. Additionally, some other mechanism of persistent handle ownership could be developed and used as an alternative.

Chapter 4

Network Service

Many of the services we might want to provide require network access; a motivating example for much of our design is Web servers. In supporting network access, we must also consider the security implications associated with it. To the network, the data being transmitted has no meaning; the network service cannot understand every protocol, and even if it could, data may be encrypted. Only the process using the connection understands the significance of the data and the labeling that should be applied.

In this chapter, we consider the design and implementation for `netd`, a process that supports network access. `netd` provides TCP/IP connections [27, 29] to user applications and handles buffering, network card interfacing, and other networking issues internally. To apply Asbestos labels to network connections, `netd` allows a process to attach a handle to a connection, with which all processes using that connection will be contaminated.

4.1 Design

4.1.1 Application Programming Interface

The lifetime of a connection begins when an application requests a socket. Unlike in UNIX [49], `netd` wraps the entire process of creating a socket and either connecting

to a remote host or listening for connections in one Asbestos message. When an application requests a new socket, `netd` responds by granting a handle for the new socket at $\star$.

On a connected socket, an application may perform `READ` and `WRITE` operations to transfer data, `CONTROL` operations to close the connection or change connection options (such as the low-water mark), and `SELECT` operations to determine the available buffer space.

When a process has a handle for a listening socket, it may accept new connections by sending a `READ` message to `netd`; `netd` replies with a handle for the newly accepted connection once one arrives. The resulting connection may be used as above.

4.1.2 Securing Network Connections

Network connections are more complicated to label than processes. We have no information about the data arriving over the network that might make it easy to apply labels to the connection. Our solution to this problem is to maintain an optional contamination handle per connection. When a socket is created, the new owner of that connection can tell `netd` to add a contamination handle to the connection. From that point forward, whenever `netd` sends a message in response to an operation on a connection, it contaminates the recipient with that contamination handle at **3**. In this way, the creator of a socket can enforce mandatory security policies on a connection.

4.1.3 Avoiding Contamination

This design for securing network connections contains a problem. Consider a process P using a network connection with a contamination handle h. When P first accesses the connection, it becomes contaminated with $\{h\,\mathbf{3}\}$ in its send label. This is as expected; `netd` uses the contamination label C_S to perform the contamination, and remains uncontaminated itself.

However, after becoming contaminated, P will no longer be able to communicate with `netd` (or any other similar service), as `netd` is unwilling to become contaminated. In fact, `netd` must refuse to accept any contamination, on the assumption that other applications wish service as well. If `netd` accepted contamination for one connection, any application with an open connection would eventually become similarly contaminated as soon as it communicated with `netd`. No application would be willing to accept this.

One option to avoid contamination is to use memory region labeling, as described in Section 2.3.4, with **vm_save** and **vm_restore**. This allows a service to handle contaminated data without steadily increasing its contamination level. Unfortunately, the **vm_save/vm_restore** design is poorly suited to `netd`. When using memory region labeling, a process runs only when it receives a message; for processes that handle events besides messages (e.g., TCP timers and packet arrival), this style of service is not acceptable. While Asbestos could produce messages in response to such events, it does not currently do so.

Another method of avoiding accumulating contamination is to use decontamination; any process which owns a handle may decontaminate a third party with respect to that handle. However, this requires a third party to interact with `netd` every time `netd` is contaminated (i.e., after every message). Ignoring the overhead of doubling the message traffic, it is unreasonable to require this behavior from all applications.

The only remaining option is for `netd` to own the handles it uses to contaminate (i.e., $P_S(h) = \star$). This requires that the service either creates the handle or is granted the handle by the handle's creator (directly or indirectly). Although this approach solves the problem, it requires that the process contaminating the connection owns the handle used to contaminate. To evaluate whether this is a reasonable requirement, we consider typical usage patterns.

We expect that an application would want to label a connection in order to enforce contamination on another process that uses the connection. For example, a front-end process F might establish the connection, determine what restrictions to apply (in the form of contamination), and hand the connection to a worker process

W. In this model, F must create a handle to use in contaminating the connection.

Thus, it is expected that no process would label a connection that it does not intend to distribute; to do so, it would have to create a handle to use in contaminating the connection, and therefore own the handle. Since the process owns the handle, it cannot become contaminated, even if it receives a message with $E_S = \{h\,\mathbf{3}\}$. We see that any process attaching a contamination handle to a connection must own the handle, and would contaminate the connection only in order to contaminate other processes. Thus, in practice, the approach of `netd` owning handles is acceptable and sufficient for the expected usage model.

4.2 Implementation

`netd` uses the lwIP TCP/IP stack [9] as the support for handling the TCP and IP protocols and internally handles connection maintenance issues.

4.2.1 Buffering

For each connection, `netd` maintains buffers separate from those provided by lwIP. One buffer is used for received data; one portion is allocated for the TCP window [4] and the rest for "preemptive acknowledgments," where `netd` acknowledges the data before any application using the connection has accepted it. We observed that the time required for the application to accept the data led to acknowledgements being returned with high latency; to counter this difficulty, we allowed `netd` to send acknowledgments before an application accepts the data. Since the amount of data a remote host is permitted to send without acknowledgment is the same as the TCP window size, no data will ever be transferred without available buffer space.

`netd` additionally maintains a send buffer for data that cannot yet be transmitted (e.g., because the interface is full, because the remote host's window is full). When an application attempts to send data when `netd` has insufficient buffer space, `netd` enqueues only the data that fits in the buffer; the application is told how much data has been enqueued and must retransmit the remainder.

4.2.2 Callback-Based Communications

`netd` uses callbacks to process packets; there are no blocking operations in `netd`. When an application sends data to a socket, only the data that can be immediately buffered is queued. The remaining data is not transmitted, and the application is informed of the situation. When an application reads data from a socket, if the read may be immediately satisfied, `netd` returns the data immediately. If not, the application's requests is put on a queue, which is examined whenever new data arrives from the remote host.

This style lends itself to applications that send a message to `netd` and continue other processing until the command finishes. When reading data, the application receives a reply as soon as the minimum data length (i.e., the low-water mark) is available. When an application has a minimum amount of data it wishes to send, it performs a `SELECT` on a handle, and `netd` replies as soon as there is sufficient buffer space.

This callback-based style of processing extends to `netd`'s interaction with lwIP as well. When data arrives from a socket, lwIP calls into `netd` to inform `netd` of the data; `netd` informs lwIP once it has completed processing. Similarly, lwIP calls up to `netd` whenever data has completed transmission, so that `netd` may enqueue more data on the lwIP transmission buffer.

4.2.3 Connection Hand-Off

As described in Section 4.1.3, we expect that a common usage pattern of `netd` is to create a connection, either by contacting a remote host or accepting an incoming connection, and hand off its handle to a process to serve that connection. This style complicates connection tracking, as we cannot count on the death of a process as a sign that a connection should be terminated.

To solve this problem, we added a feature to the Asbestos kernel to generate an optional `DEAD` message addressed to the creator of a handle as soon as no other process has access to that handle. This extension allows `netd` to garbage collect

connections that may no longer be accessed. If a front-end process establishes a connection as described above and hands it to a worker to service it, as soon as the worker finishes service, perhaps by calling **vm_restore**, the kernel informs `netd` that no other process may access it, and `netd` closes the connection.

Chapter 5

Database Service

In this chapter, we describe the design and implementation of the OK database service (`okdb`), a service supporting structured storage, which Web server applications use to store data. `okdb` provides a nearly-complete implementation of the SQL standard [1], allowing applications to create tables to store data, insert new data in the form of table rows, and query existing data. Additionally, `okdb` applies labeling to database content in the form of authenticated IDs, which are used to allow controlled-flow data access and ensure that only authorized clients can modify data.

5.1 Design

5.1.1 Application Programming Interface

All interactions with `okdb` take the familiar form of SQL queries. When performing a query (i.e., requesting data from the database), `okdb` responds with a "result handle" for the query results. The application may examine each row in the results in turn by sending further messages to the result handle. When performing a command (e.g., inserting, modifying, or removing data), `okdb` simply replies with a result code.

5.1.2 Labeling Database Content

To support mandatory access control on database content, we use a scheme similar to that described for a multi-user file server in Section 2.3.2. For each row in a table, `okdb` optionally maintains an authenticated ID. When an application performs a query, receiving data from `okdb`, `okdb` contacts `authd` to obtain the contamination handles for all authenticated IDs in the rows returned. In response, `okdb` automatically contaminates the recipient with the contamination handle from `authd` at 3. Similarly, when an application performs a command, modifying database content, `okdb` checks the verification label presented by the application, to ensure that the application possesses the identity handle associated with the authenticated ID. In this manner, we obtain similar security properties as for the multi-user file server example but with the flexibility to perform arbitrary labeling on database content, rather than labeling based solely on specific users.

Row-level labeling, however, is insufficient. For example, we may wish to prevent arbitrary insertions into a table; with row-only labeling, this is not possible. To solve this problem, we additionally associate authenticated IDs with tables. Using similar checks as before, when an application performs a query, we additionally contaminate the recipient with the table's contamination handle. When an application performs a command, we additionally check the verification label for the table's identity handle.

5.1.3 Avoiding Contamination

The design described so far has the same problem for `okdb` as we found for `netd` in Section 4.1.3; `okdb` will steadily increase contamination as it interacts with other processes.

As before, requiring a third party to decontaminate `okdb` after every message is an unacceptable demand. Memory region labeling is now a possibility, but there are difficulties. For `okdb`, memory region labeling would require separating differently labeled data into separate pages. For a database, where each row in a table may correspond to a different client (and therefore, differently labeled data), having a

small amount of data in a page of memory could be wasteful.

Again, it appears that using handle ownership is necessary. For `okdb`, we expect that applications would want to label records to enforce access control on themselves. For example, a Web server might rely on labeling to ensure that it will not distribute one client's data to a different client's network connection. In this example, two types of interactions between the Web server and `okdb` might occur:

- A new client performs a request, adding new records to the database with a new authenticated ID.
- A client performs requests that require fetching information from the database.

In the first interaction, the Web server creates a new authenticated ID using an owner password. The use of an owner password ensures that a process presenting the proper password may be granted the contamination and identity handles. To ensure that `okdb` does not become contaminated, we require that any request to contaminate database records with an authenticated ID includes the owner password for the password ID. This requirement is akin to requiring that `okdb` is granted both the contamination and identity handles, but the inclusion of the password ensures that the "grant" persists across reboots.

When a client performs requests requiring existing database content, we presume that it begins with no knowledge of authenticated ID passwords and no contamination (e.g., by using **vm_save** and **vm_restore**). When `okdb` returns records, it can contaminate the Web server without becoming contaminated itself, as it can obtain the contamination and identity handles in its send label at $\star$. If the Web server sends further messages, `okdb` continues to remain uncontaminated.

5.2 Implementation

5.2.1 SQL Processing

`okdb` relies heavily on the SQLite database library [24] to perform SQL processing. When queries and commands arrive, `okdb` simply hands the query to SQLite to

perform the work. Once SQLite returns a result, `okdb` determines whether the query returned data. If it did, `okdb` creates a new results handle and returns it to the requester. Otherwise, it simply returns the result code from SQLite.

An application may examine the results by iterating through the rows of the results handle. A client can ask the database to step to the next row of the results; given a row, the client can ask for the values of any field in the row in any of a number of formats (e.g., double, integer, string). An application may also ask the native type of a column of the results; when a request is made for data not in the native format, it is automatically converted.

5.2.2 Storing Authenticated IDs

We considered several methods for storing authenticated IDs for rows and tables. The solution we arrived at uses two different methods to store authenticated IDs. For rows, `okdb` requires every table contains a column called `authid`. When a query arrives requesting data, `okdb` modifies the query to return the authenticated ID as well. Without exposing the authenticated ID in the results, `okdb` looks up the authenticated ID with `authd` as each row is examined and contaminates the recipient. When data is inserted, `okdb` modifies the query to additionally insert the authenticated ID into the new record.

For tables, `okdb` maintains a separate SQL table, `table_authid`, of table names and authenticated IDs. When queries arrive, `okdb` examines `table_authid` to determine what access checks (using identity handles and verification labels) must be performed and what contamination must be applied.

5.2.3 Storing Authenticated ID Passwords

To store owner passwords for authenticated IDs, we take advantage of the database abstraction already available from SQLite, and create a separate database (inaccessible to clients) in which we store a SQL table, `authid_info`, of authenticated IDs and passwords. When an authenticated ID is presented by an application to be asso-

ciated with a database row, `okdb` first ensures that a password was presented, and looks up the authenticated ID and password with `authd` to verify that it is correct. Once `okdb` has done so, `okdb` inserts a new row into `authid_info` containing the authenticated ID and password and performs the query as described above.

5.2.4 Caching Contamination and Identity Handles

A naïve implementation of `okdb` would look up authenticated ID values on-demand; when a row is accessed, determine the contamination and identity handles before allowing the access. Our implementation of `okdb` does exactly this, but caches found values in memory, using a temporary SQLite database. `okdb` uses a table, `authid_values`, to cache the contamination and identity handles for all authenticated IDs that have been retrieved. When `okdb` needs a contamination or identity handle, it first checks the database to determine whether the values have already been determined. The database is not saved on disk, so the table is recreated when `okdb` is restarted or the system reboots.

5.3 Discussion

The area most needing further research in `okdb` is the requirement that `okdb` owns all authenticated IDs that it uses. As described in Section 5.1.3, other mechanisms available in Asbestos, such as memory region labeling, fail for `okdb`. We need to examine whether the features provided can be augmented to support different service models.

One possible alternative is the use of a custom database persistence layer. By writing a database from the ground up, we can structure it to support memory region labeling, although still with the overhead of wasted space. It is unclear whether such a design would be comparably simple to the SQLite-based system in place now, although it would come with a stronger security guarantee.

Another feature to investigate is remote database access over a network. Most databases used are available through a network interface, allowing the database to

reside on a separate, high-performance machine. In Asbestos, this is not possible, as the labeling mechanism does not automatically transfer across network connections. Developing an extension to Asbestos that allows Asbestos messages to be transmitted across a network would allow remote database access, as well as other interesting possibilities such as network file access.

Chapter 6

Putting it All Together

In this chapter, we consider the structure of a complete Web server system using `authd`, `netd`, and `okdb`, as described in Chapters 3 , 4, and 5. Using a modified version of OKWS, we present a complete system that provides stronger security guarantees than the original OKWS design.

The Asbestos implementation of OKWS isolates logically distinct services in different worker processes and also enforces user isolation to prevent one compromised service from leaking information about other users. Rather than using a separate process per user, OKWS uses memory region labeling with **vm_save** and **vm_restore** to provide full isolation of one user's data from others.

6.1 The `launcher`

OKWS first starts `launcher`, which starts the separate OKWS components and ensures proper communication privileges for each. The launcher creates N *worker verify handles* (where N is the number of services), one for each `worker`. These are used to verify that a worker process is valid.

The `launcher` starts `demux` first, which grants its own handle (the *demux handle*) to the launcher. After starting `demux`, `launcher` grants `demux` each of the worker verify handles.

Next, `launcher` starts N `worker`s, granting each `worker` the demux handle.

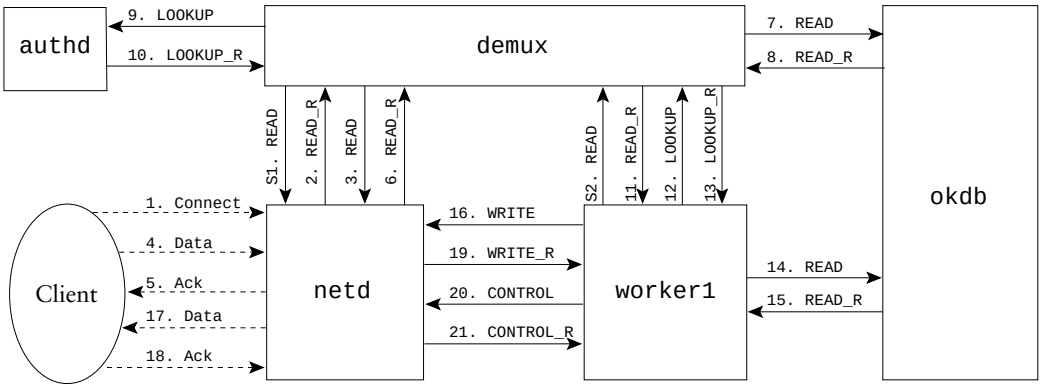

Figure 6-1: The sequence of messages when processing a Web request. Note some messages are omitted for clarity.

This allows each `worker` to contact `demux` to announce that it is ready to service a request. The `launcher` also grants each `worker` its worker verify handle.

6.2 The `demux`

After sending the demux handle to `launcher`, `demux` waits for each `worker` to contact it. When a `worker` contacts `demux`, it passes its worker verify handle, to prove to `demux` that it is the correct `worker` for its service. After verifying `worker` W, `demux` creates a new handle and sends it to W. The `worker` later uses this handle to communicate with `demux`.

This sequence of events for handling connections is shown in Figure 6-1. The `demux` contacts `netd` and opens a listen socket for incoming connections (message S1). When a connection arrives (messages 1–2), `demux` reads enough of the HTTP headers to determine what user U is making the request, and what worker W that user is requesting (messages 3–6). Once it has done so, it contacts `okdb` (message 7), determining what authenticated ID is applied to U's data (message 8). It next contacts `authd`, presenting the authenticated ID and owner password (message 9), and is granted U's contamination and identity handles (message 10) in response. At this point, the `demux` is ready to hand the connection off to W; however, it must ensure that as soon as W reads from U's socket, it becomes contaminated with U's contamination handle. To do so, `demux` tells `netd` to apply U's contamination

handle to the connection.

6.3 The `workers`

An OKWS system will typically have many workers running, each implementing a logically distinct Web service. Each service sends a `READ` message (message S2) when it first starts up, requesting incoming requests. When the `demux` is ready to hand a particular worker W a connection, it simply replies to this `READ` message (as in message 11). The worker then immediately can reply with another `READ` message (as in S2), since it is capable of serving overlapping clients.

The handoff shown in messages 11 through 13 requires care. Each worker maintains server-side state for each active user with which it is communicating, including send and receive buffers. In our implementation, each worker W sets aside a 64-page region for each user U that becomes active, and it allocates pages there lazily as the user requires them. Moreover, it taints this entire region with U's contamination handle, so that it might later write to it when it has U's contamination handle in its send label.

If the `demux` were to deliver U's connection contaminated with U immediately in message 11, the worker would be at a loss. It would have to set aside a region for U's state, but it could not write to any persistent, non-tainted memory to indicate that it had done so. OKWS on Asbestos instead uses a two-phase handoff protocol. In message 11, the `demux` informs W that it is about to deliver a connection for user U but does not contaminate W as it does so. W then consults a table T (implemented as a quadratic hash table), either finding a previously allocated region for U or allocating a new one. Note that T resides in *uncontaminated* memory. When W writes to T that it has allocated a region for the user U, it can later read this mapping without becoming contaminated.

After noting this new region assignment in T, W is ready to accept the connection and the contamination, and it does so in messages 12 and 13. Once it has received U's connection, it enjoys a reserved, pre-tainted region of pages, which it

can access across connections when contaminated with U's contamination handle. The `worker` then services U's connection. After parsing the request, the `worker` performs any database access necessary (represented by messages 14–15). This may include querying existing data (which may contaminate the `worker` with contamination handles) or modifying data (for which the `worker` must present the appropriate identity handle). Finally, the `worker` sends the reply back to the client over the connection (messages 16–19), closes the connection (messages 20–21), calls **vm_restore**, and waits to service another connection.

One interesting issue remains: freeing memory. When worker W corresponds with U over many HTTP requests, it can grow and shrink U's region while contaminated, always leaving at least one page allocated to store a map of the region. When the user U explicitly logs off, W would like to reclaim the last page and reassign U's region to other users who become active. To achieve this, W sends a message to the `demux`, informing it that U's region should now be available to other users. The `demux` immediately sends back an acknowledgment, telling W to free the last page in the region. The demux then waits a random amount of time, on the order of ten minutes, and sends a second uncontaminated message, telling W to mark U's region unallocated in the table T.[1] The region is now available for reassignment to a different user. The `demux` must be careful to synchronize these last messages with potential requests from U, so as to avoid race conditions on W; if `demux` tells W to free the memory region between the two steps of the two-phase protocol, W will discover the expected memory region does not exist.

6.4 Security Considerations

In this section, we consider the security of the complete system as described previously. For each component, we consider weaknesses and unsolved problems.

[1] `demux`'s response represents a storage channel, which we mitigate by a long delay.

6.4.1 authd

It is clear that authd plays an important role as a member of the trusted computing base (TCB) of our system. Many processes must interact with authd to obtain new authenticated IDs and obtain details of existing authenticated IDs. In our implementation, we allow all processes to contact authd. The correctness of our implementation is therefore critically important; bugs might allow applications to inappropriately communicate, own contamination handles, or modify existing data.

Fortunately, authd is a simple enough application that it is relatively straightforward to audit. All the functions it performs are simple: performing essentially static queries on a database and sending and receiving messages. It is reasonable to verify that authd properly implements the policies we have described.

6.4.2 netd

Analyzing the security of netd requires examination of several components: our network card driver implementation, lwIP, and netd's interactions with lwIP. Because netd plays a privileged role, owning all handles it uses to contaminate, correctness of our implementation is again critical.

Our network card driver is based on the Linux ne2000 driver and lwIP, both of which are in widespread use. However, we have no guarantees of their correctness, and an audit of either component would be difficult. Additionally, the code for netd itself is sufficiently complex, using several callbacks and complex structures, that we believe a thorough audit would be difficult to perform.

6.4.3 okdb

okdb is the most complex service of all that we have considered so far. Although SQLite is a relatively small database, that component alone is approximately 30,000 lines of code. Additionally, our implementation relies on properly analyzing SQL statements to determine what rows and tables are referenced and modified. The entire okdb service represents a complex task to audit. As with netd, okdb "owns"

all authenticated IDs that it uses. A flaw in `okdb`'s logic could lead to improper disclosure of sensitive data.

Another concern with `okdb` is that it relies upon persistent storage, as provided by our filesystem implementation. Flaws in our filesystem service could lead to disclosure of sensitive data. The SQLite database format is public, and it is therefore easy to construct the database given the file's content.

6.4.4 OKWS

The new design of OKWS provides a strong guarantee to system designers and clients: even if an attacker discovers a bug in a `worker` process, the labeling already in place prevents the attacker from viewing other users' data. By explicitly enumerating the labeled traffic that may be transmitted over the network, we ensure that even flawed software does not result in improper disclosure.

However, OKWS relies on other components to perform their roles properly. Specifically, a bug in the `demux` process would destroy all security guarantees. As `demux` owns all handles and authenticated IDs, an attacker controlling `demux` could arbitrarily distribute labeled data, as all data is labeled with an identifier created by `demux`. While the functionality of `demux` is limited for precisely this reason, the trust placed in `demux` remains a concern.

6.4.5 Developing Secure Web Applications

For Web application developers, the system comprised of `authd`, `netd`, `okdb`, and OKWS provides a strong and simple foundation for building secure applications. Developers are freed from the requirement of remembering to apply security policies to data consistently, as those policies are automatically applied. As contamination is automatically applied by `okdb` and `netd`, as selected by the OKWS `demux`, Web applications are automatically marked as contaminated when they examine privileged data.

The great gain of this system is that bugs in code created by real-world developers

are not catastrophic, as long as the OKWS `demux` properly contaminates worker processes. In doing so, we assure that even compromised services cannot improperly release data. While we do rely on the correctness of several supporting services, as described above, the applications developed by end-developers need not be bug-free.

Chapter 7

Evaluation

In this chapter, we consider the effectiveness of the complete Web application system, consisting of OWKS, `okdb`, `netd`, and `authd`, and discuss the performance of the individual services as appropriate. We also discuss areas of improvement.

We expect that the additional security offered by the complete Web application system comes at a small cost to performance. To verify this, we have implemented and measured a simple version of OKWS on Asbestos, using `okdb`, `netd`, and `authd`. Using this implementation, we measure the time to process client requests and determine bottlenecks.

7.1 End-to-End Measurements

We performed experiments on an AMD Athlon 1500+ (1.3 GHz) with 64 MB of memory, a 10 Mb Ethernet network, and with a local Linux client generating requests.

Figure 7-1 shows the cumulative distribution function of request latency. In this example, all client requests arrive sequentially; due to bugs in our TCP/IP implementation, concurrent connections lead to timeouts and do not offer a performance gain. All clients but one are serviced within 7.2 ms, the last in 24 ms. The complete system can serve nearly 200 connections per second.

As shown in Figure 7-2, performance is not limited by available processor time,

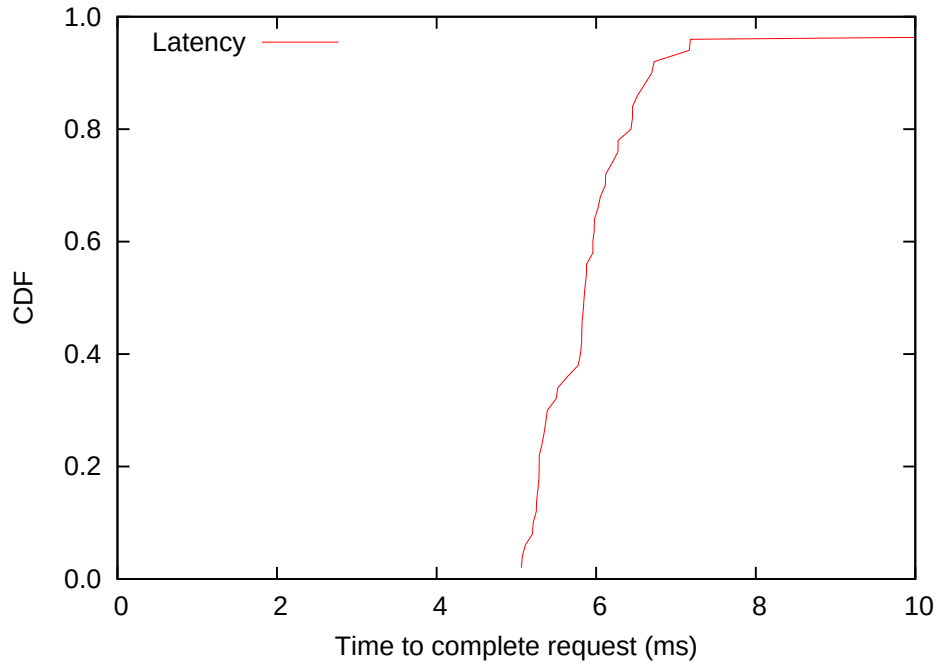

Figure 7-1: CDF of request latency as seen from the client.

suggesting concurrency will offer a performance benefit when the TCP/IP implementation is improved. CPU time is dominated by time spent on IPC. Service for a typical request has approximately 20 messages, as shown in Figure 6-1, although several messages are omitted there for clarity. The actual number of messages varies depending on the service requested, but is often near 50.

The cost of the security in this system is low. The cost of `authd` and kernel security-related tasks sums to only 8% of the CPU time, and we believe that with a tuned implementation this cost can be lowered.

7.2 Discussion

7.2.1 Asbestos Kernel

There are performance areas of the Asbestos kernel that may benefit from further study. In particular, the cost of IPC as shown in Figure 7-2 is quite high. It is clearly significant if this cost can be reduced, as all communication is through IPC. For example, a typical `okdb` query uses approximately dozens of messages: two to

Module	Execution Time
Kernel: miscellaneous	0.30%
Kernel: memory management	0.14%
Kernel: **vm_restore**	3.46%
Kernel: IPC	14.02%
Kernel: network	5.49%
Kernel: console	0.29%
User: `authd`	4.23%
User: `netd`	2.87%
User: `okdb`	4.49%
User: OKWS `demux`	4.04%
User: OKWS `worker`	3.98%
Idle	56.72%

Figure 7-2: Execution time of various modules while OKWS is under heavy load.

create the query (request and response), ten for each row (two to access the row, two more for each field in the row), and two to destroy the results and free memory. Once OKWS improves to the point where it has saturated the CPU, the cost of IPC will become a further bottleneck.

Further improvements to **vm_restore** would likely prove valuable as well. In this example, nearly as much time is spent by the OKWS `worker` performing its service as is spent in **vm_restore**. Under heavy load, this may nearly halve the performance a `worker` can provide.

7.2.2 `authd`

We were surprised that `authd` required more CPU time than each of the OKWS components, as the actions it performs are all relatively simple. As `authd` uses a very straightforward, untuned implementation, we suspect further performance can be easily gained using simple techniques. For example, `authd` uses no caching of results; if a two sequential requests are made for the same contamination handle, two database queries are performed. Similarly, each request performs two separate database queries, one for the contamination handle, and one for the identity handle.

7.2.3 `netd`

One significant change we would like to investigate is the use of a better Ethernet driver and card. Our network card only supports 10 Mb speeds, and we expect that a faster card would expose shortcomings in our implementation that might later slow performance.

In the same vein, our Ethernet driver and card appear to have bugs that lead to high CPU usage. In the example above, the CPU usage is low, but under full network usage, the Ethernet driver takes up to 20% of the CPU. As the driver has been used successfully in many Linux installations, we believe the Ethernet card itself may be flawed; this issue still remains to be investigated.

We remain unconvinced that our use of lwIP is optimal; we have noticed that under high network usage, lwIP uses up to 25% of the CPU. This unexpected result is worrisome; while it has not yet become a performance bottleneck (CPU idle time is available), we are concerned that this is symptomatic of deeper problems.

In `netd` itself, we would like to reduce the number of copies performed. From when data arrives from the network until it is handed to an application, it is copied several times:

- from the network card buffer to an lwIP buffer,
- from an lwIP buffer to a circular buffer in `netd`,
- from `netd`'s circular buffer to a temporary buffer,
- from the temporary buffer to kernel memory, and
- from kernel memory to the application's buffer space.

Of course, there may be further application-level copying once the data has been received. Clearly some of these copies are unnecessary, and may lead to performance bottlenecks in the future. More experimentation is necessary to determine the impact, if any, of this copying.

Finally, we would like to add further protocol support to `netd`. While TCP support is necessary and sufficient for many applications, a notable omission is DNS support. In addition, adding UDP support would make the network implementation more complete.

Chapter 8

Related Work

The security model of Asbestos derives from several ideas in previous systems, including capabilities [3, 17, 37, 23], virtualizable interfaces to both the kernel and other processes (a logical extension of system-call interposition libraries [13, 34]), and decentralized mandatory access control [22].

The Flask System applies MAC to the Fluke Microkernel [38], and many of Flask's core design principles have been implemented in SELinux [18]. SELinux adds mandatory access control to Linux by allowing administrators to create static policy files describing which resources applications can access and how processes may interact with one another. Similar to SELinux is TrustedBSD [50], providing the same functionality for FreeBSD.

Systems such as SELinux and TrustedBSD are attractive to developers and administrators as they preserve the POSIX interface to which users are accustomed. Implementing a secure application using either system is straightforward: implement the application as on a traditional POSIX system and create a policy file defining the rights of the application. However, this model does not support the dynamic creation of security domains; an implementation of the original UNIX OKWS design would be possible, but the extended design separating user data as described in this thesis would not be.

Application developers can take steps to ensure secure software with existing systems; OpenSSH [35] and qmail [2], among others, use mechanisms to ensure that

bugs are confined and do not lead to complete failure, most often by separating logically distinct functions into separate processes as in OKWS. However, intricate steps are required to initialize such an environment and ensure that a compromise in one component does not lead to compromise of the complete system. Asbestos provides a simple mechanism for enforcing such isolation, and the kernel enforces mandatory access control automatically.

Databases such as LDV [39, 40] have supported multilevel access. However, Asbestos labeling provides a more flexible system, and `okdb`'s use of the operating system's security mechanisms allows a consistent method for securing and tracking the flow of data across different processes.

Chapter 9

Conclusion

In this thesis, we have presented the design and implementation of several services for Asbestos, including persistent identification, network access, and database persistence. Using the features provided by Asbestos, we have designed services that limit the flow of data and allow programmers to ensure that even flawed software does not allow information leakage. By analyzing typical usage patterns, including those of our motivating application, we have shown that these services support the styles of security we consider most critical. Although the services themselves are privileged, applications using them have mandatory policies enforced that prevent improper disclosure of data.

Further work is still necessary to reduce the amount of trusted code. More effective ways of preventing contamination, such as a different implementation of memory region labeling, might provide stronger mandatory security guarantees and reduce the size of the trusted computing base.

Bibliography

[1] American National Standards Institute. *American national standard for information systems: database language — SQL: ANSI X3.135-1992*. American National Standards Institute, 1430 Broadway, New York, NY 10018, USA, 1992. Revision and consolidation of ANSI X3.135-1989 and ANSI X3.168-1989, Approved October 3, 1989.

[2] Daniel J. Bernstein. qmail. `http://cr.yp.to/qmail.html`.

[3] Allen C. Bomberger, William S. Frantz, Ann C. Hardy, Norman Hardy, Charles R. Landau, and Jonathan S. Shapiro. The KeyKOS(R) nanokernel architecture. In USENIX, editor, *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures: 27–28 April, 1992, Seattle, WA, USA*, pages 95–112, Berkeley, CA, USA, April 1992. USENIX.

[4] D. D. Clark. RFC 813: Window and acknowledgement strategy in TCP, July 1982. Status: UNKNOWN.

[5] S. E. Deering. RFC 988: Host extensions for IP multicasting, July 1986. Obsoleted by RFC1054, RFC1112 [6, 7]. Obsoletes RFC0966 [8]. Status: UNKNOWN.

[6] S. E. Deering. RFC 1054: Host extensions for IP multicasting, May 1988. Obsoleted by RFC1112 [7]. Obsoletes RFC0988 [5]. Status: UNKNOWN.

[7] S. E. Deering. RFC 1112: Host extensions for IP multicasting, August 1989. Obsoletes RFC0988, RFC1054 [5, 6]. See also STD0005 [30]. Updated by RFC2236 [10]. Status: STANDARD.

[8] S. E. Deering and D. R. Cheriton. RFC 966: Host groups: A multicast extension to the Internet Protocol, December 1985. Obsoleted by RFC0988 [5]. Status: UNKNOWN.

[9] Adam Dunkels. Minimal TCP/IP implementation with proxy support. Technical Report T2001-20, Swedish Institute of Computer Science, October 26, 2001.

[10] W. Fenner. RFC 2236: Internet Group Management Protocol, version 2, November 1997. Updates RFC1112 [7]. Status: PROPOSED STANDARD.

[11] Free Software Foundation. GNU C library. `http://www.gnu.org/software/libc/libc.html`, 2005.

[12] The Apache Software Foundation. The Apache HTTPD server project. `http://httpd.apache.org/`, 2005.

[13] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications. In *Proceedings of the 6th Usenix Security Symposium*, San Jose, CA, USA, 1996.

[14] The PHP Group. PHP: Hypertext preprocessor. `http://www.php.net/`, 2005.

[15] Max Krohn, Petros Efstathopoulos, Cliff Frey, Frans Kaashoek, Eddie Kohler, David Mazières, Robert Morris, Steve VanDeBogart, and David Ziegler. Make least privilege a right (not a privilege). In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems*, June 2005.

[16] Maxwell Krohn. Building secure high-performance web services with OKWS. In *Proceedings of the 2004 USENIX*, Boston, MA, June 2004.

[17] Henry M. Levy. *Capability-Based Computer Systems*. Digital Press, 1984.

[18] Peter Loscocco and Stephen Smalley. Integrating flexible support for security policies into the Linux operating system. In USENIX, editor, *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference: June 25–30, 2001, Marriott Copley Place Hotel, Boston, Massachusetts, USA*, pages ??–??, Berkeley, CA, USA, 2001. USENIX Association.

[19] J. C. Mogul. RFC 919: Broadcasting Internet datagrams, October 1984. See also STD0005 [30]. Status: STANDARD.

[20] J. C. Mogul. RFC 922: Broadcasting Internet datagrams in the presence of subnets, October 1984. See also STD0005 [30]. Status: STANDARD.

[21] J. C. Mogul and J. Postel. RFC 950: Internet Standard Subnetting Procedure, August 1985. Updates RFC0792 [28]. See also STD0005 [30]. Status: STANDARD.

[22] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proceedings of the sixteenth ACM Symposium on Operating System Principles*, pages 129–142, October 1997. Appeared in ACM Operating Systems Review volume 31, number 5.

[23] R. M. Needham and A. J. Herbert. *The Cambridge CAP Computer and Its Operating System*. Addison-Wesley, Reading, MA, 1982.

[24] Michael Owens. Embedding an SQL database with SQLite. *Linux Journal*, 110:62–64, 66, 68, June 2003.

[25] J. Postel. RFC 760: DoD standard Internet Protocol, January 1980. Obsoleted by RFC0791, RFC0777 [27, 26]. Obsoletes IEN123 [32]. Status: UNKNOWN. Not online.

[26] J. Postel. RFC 777: Internet Control Message Protocol, April 1981. Obsoleted by RFC0792 [28]. Obsoletes RFC0760 [25]. Status: UNKNOWN. Not online.

[27] J. Postel. RFC 791: Internet Protocol, September 1981. Obsoletes RFC0760 [25]. See also STD0005 [30]. Status: STANDARD.

[28] J. Postel. RFC 792: Internet Control Message Protocol, September 1981. Obsoletes RFC0777 [26]. Updated by RFC0950 [21]. See also STD0005 [30]. Status: STANDARD.

[29] J. Postel. RFC 793: Transmission control protocol, September 1981. See also STD0007 [31]. Status: STANDARD.

[30] J. Postel. STD 5: Internet Protocol: DARPA Internet Program Protocol Specification, September 1981. See also RFC0791, RFC0792, RFC0919, RFC0922, RFC0950, RFC1112 [27, 28, 19, 20, 21, 7].

[31] J. Postel. STD 7: Transmission Control Protocol: DARPA Internet Program Protocol Specification, September 1981. See also RFC0793 [29].

[32] Jon Postel. DOD standard Internet protocol, December 1979.

[33] The OpenSSL Project. OpenSSL: The open source toolkit for SSL/TLS. `http://www.openssl.org/`, 2005.

[34] Niels Provos. Improving host security with system call policies. In *Proceedings of the 11th USENIX Security Symposium*, pages 257–272. USENIX, August 2003.

[35] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *Proceedings of the 11th USENIX Security Symposium*, pages 231–242. USENIX, August 2003.

[36] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[37] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: a fast capability system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP'99)*, pages 170–185, Kiawah Island Resort, near Charleston, South Carolina, December 1999. Appeared as ACM Operating Systems Review 33.5.

[38] Ray Spencer, Stephen Smalley, Peter Loscocco, Mike Hibler, David Andersen, and Jay Lepreau. The Flask security architecture: System support for diverse security policies. In *Proceedings of the 8th USENIX Security Symposium*, pages 123–139, Washington, D.C., USA, August 1999. USENIX.

[39] P. Stachour and B. Thuraisingham. Design of LDV: A multilevel secure relational database management system. *IEEE Transactions on Knowledge and Data Engineering*, 2(2), June 1990.

[40] P. Stachour, B. Thuraisingham, and P. Dwyer. Update processing in LDV: A secure database system. In *Proc. 11th NIST-NCSC National Computer Security Conference - Postscript*, pages 96–115, 1988.

[41] Computer Emergency Readiness Team. Microsoft Internet Information Server (IIS) 4.0, 5.0, and 5.1 buffer overflow in chunked encoding transfer mechanism for ASP. `http://www.kb.cert.org/vuls/id/669779`, 2002.

[42] Computer Emergency Readiness Team. Microsoft Internet Information Server (IIS) vulnerable to buffer overflow via inaccurate checking of delimiters in HTTP header fields. `http://www.kb.cert.org/vuls/id/454091`, 2002.

[43] Computer Emergency Readiness Team. Apache HTTP Server contains a buffer overflow in the mod_proxy module. `http://www.kb.cert.org/vuls/id/541310`, 2003.

[44] Computer Emergency Readiness Team. Microsoft SQL Server vulnerable to buffer overflow. `http://www.kb.cert.org/vuls/id/584868`, 2003.

[45] Computer Emergency Readiness Team. Sun Java Runtime Environment allows untrusted applets to access information within trusted applets. http://www.kb.cert.org/vuls/id/393292, 2003.

[46] Computer Emergency Readiness Team. Apache vulnerable to buffer overflow when expanding environment variables. http://www.kb.cert.org/vuls/id/481998, 2004.

[47] Computer Emergency Readiness Team. MIT Kerberos krb524d insecurely deallocates memory (double-free). http://www.kb.cert.org/vuls/id/340792, 2004.

[48] The Kernel.Org Organization, Inc. The Linux kernel archives. http://www.kernel.org/, 2005.

[49] The Open Group. *The Single UNIX Specification: The Authorized Guide to Version 3*. The Open Group, Publications Department, Apex Plaza, Forbury Road, Reading, Berkshire RG1 1AX, UK, 2002. Open Group Document Number G906.

[50] Robert N. M. Watson. TrustedBSD: Adding trusted operating system features to FreeBSD. In USENIX, editor, *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference: June 25–30, 2001, Marriott Copley Place Hotel, Boston, Massachusetts, USA*, pages 15–28, Berkeley, CA, USA, 2001. USENIX Association.