Simple-to-use, Secure-by-design Networking in Ethos

W. Michael Petullo
University of Illinois at Chicago
mike@flyn.org

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

ABSTRACT

We describe networking in Ethos, a clean-slate operating system we designed to meet the security requirements which arise on the Internet. Through careful layering, Ethos makes network encryption, authentication, and authorization protections compulsory. This means that application developers can neither avoid nor incorrectly use them, and system administrators need not audit their use. We show through a case study how Ethos reduces application complexity and makes it easier to write and deploy robust applications.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection

Keywords

Authorization, software layering

1 Introduction

It is surprisingly difficult to write and configure software to provide isolated, authenticated, and authorized networking which is sufficiently robust. Robustness measures the ability to withstand attack; to be effective, it must be commensurate with the threat level. The threat level found on the Internet is very high—as evidenced by the large number and broad range of successful attacks. Attacks routinely target weaknesses in isolation, authentication, and authorization.

It is not merely that the current state of affairs takes too much work. System complexity fundamentally limits the level of assurance possible. Ultimately, high assurance levels are possible only if a system has low complexity and security is part of its initial design. Unfortunately, existing systems are very complex and were designed in a time of low security requirements. They have accreted functionality over time, resulting in extraordinarily difficult-to-understand systems and, therefore, security holes.

This paper describes the design and implementation of networking in Ethos, an experimental Operating System (OS) with security as its primary goal. Ethos is a clean-slate design; its software layering is carefully engineered to meet security requirements. A clean-slate design enables Ethos to be far simpler, provide new semantics, and avoid existing error-prone interfaces. Existing mechanisms are routinely misused, even by competent, well-intentioned programmers and administrators [10, 9, 6].

We focus on the interfaces available to application programmers and system administrators. We will show that Ethos provides the following properties:

P1 A system model which guarantees confidentiality, integrity, and authentication of networking, independent of applications (§4.3).

P2 Complete mediation (authorization) of all network connections (§4.4).

Ethos protections are **compulsory**, meaning that application developers cannot avoid or incorrectly use them, and system administrators need not audit their use.

P1 is a guarantee to application programmers. As we will discuss in §2, use of Transport Layer Security (TLS) is often discretionary. Alternatively, Internet Protocol Security (IPsec) can be made compulsory, but in general an application developer cannot know if his application will be used with IPsec. Consequently, application code must be provided for network authentication and encryption. In contrast, Ethos's system calls guarantee these protections for *all* applications, with zero lines of application code.

P2 is a guarantee to administrators that builds on P1. Ethos' authorization policy is mandatory, simple, and powerful. Because the Ethos kernel performs network authentication, it can fully mediate network connections. Whereas the system-wide authorization policy of existing systems has difficulty governing remote users (because they are authenticated by applications), Ethos equally regulates both local and remote users.

Application programmers need better abstractions to help them avoid the security pitfalls which result in security holes. System administrators need to understand and control their systems; it must take less work to secure systems against attack. We evaluate the effect of both P1 and P2 against these goals.

In the remainder of this paper, we discuss related work (§2), our threat model and the resulting security requirements (§3), the design of networking in Ethos (§4), and an evaluation of Ethos' design and implementation (§5). Our evaluation contains a case study that compares an Ethos messaging program to Postfix.

EuroSec'13 April 14 2013, Prague, Czech Republic

2 Related work

TLS (previously Secure Socket Layer) provides cryptographic network protections above the transport layer and is normally implemented as a user-space library. In general, each application is individually configured to use TLS. Even well-meaning developers routinely misuse TLS Application Programming Interfaces (APIs) to the detriment of security [10, 9]. TLS can optionally provide user-level authentication when using client-side certificates but leaves authorization to application logic. Ethos forgoes backwards compatibility to provide a simpler, less mistake-prone platform.

Like Ethos, tcpcrypt [2] investigated ubiquitous encryption, but it maintains backwards compatibility with TCP/IP. Tcpcrypt provides hooks that applications may use to provide authentication services and determine whether a channel is encrypted. This approach differs from that of Ethos, which is clean-slate and subsumes authentication and encryption services in its system calls to ease assurance.

IPsec provides very broad confidentiality and integrity protections because it generally is implemented in the OS kernel. IPsec's major shortfall is that its protections stop at the host; it focuses on network encryption and host authentication/authorization. For example, IPsec does not authenticate or restrict users across the network.

Security-Enhanced Linux (SELinux) is a mandatory access control enforcement system for Linux [14] that has traditionally focused on enforcing an authorization policy on a single machine. IPsec and SELinux have been combined to produce labeled IPsec, which provides more comprehensive network protections [12]. Ethos' co-designed system call interface and authorization system results in significantly simpler policy specification than with SELinux.

Many application architectures have adopted Multics' principle of least privilege [4]. Such techniques isolate sensitive code and unify authorization policy; they are useful on Ethos too. However, Ethos goes further because it identifies security properties that should be shared by all applications—regardless of design—and turns them into system-wide guarantees.

Distributed firewalls overloads POSIX networking APIs (connect and accept), providing kernel-based, user-aware enforcement of a distributed authorization policy [11]. Ethos was inspired by this approach and adds transparent encryption and authentication to its networking system calls (§4.3). The STRONGMAN architecture [13] addresses multiple, overlapping security mechanisms which complicate system administration. Like STRONGMAN, Ethos identifies users by their public key.

Plan 9 implements its security in two layers: the kernel and factotum [5]. Authentication and key management are handled by factotum, a per-user agent that supports a wide range of authentication protocols. Once factotum has authenticated a user and established a shared key, the key may be used to encrypt communication using OS-based TLS; Plan 9 provides user space access to its encryption routines through a pseudo device. However, the use of both factotum and network encryption remain discretionary—an application need not use them. In contrast to Plan 9, Ethos always encrypts and authenticates network connections regardless of application code.

HiStar provides a simplified, low-level interface and implements mandatory information-flow constraints, providing a UNIX layer for compatibility [19]; DStar builds on HiStar to provide information flow across hosts on a network [20]. Ethos also provides a simplified interface, but it focuses on aiding application developers by providing a high level of abstraction. Thus developments in HiStar and Ethos are complimentary: HiStar's UNIX layer could be replaced with Ethos' higher abstractions, and Ethos could adopt many of HiStar's contributions to flow control.

3 Threat model

The attacker considered here has very broad access: he can run applications on the Ethos host, including malicious software; control remote hosts and other unprivileged virtual machines; and control network media. Using these capabilities, he can attempt to violate an Ethos host's security policy by observing network packets, deploying counterfeit services, or making odd requests to Ethos-hosted services. We assume the attacker is highly skilled; we are especially concerned with his ability to exploit inadvertent application or configuration errors with respect to network protections. An attacker might have a local user account.

Some threats are beyond the scope of this paper. A trusted application programmer with malicious intent could corrupt a security-sensitive application while it is being written. We will describe how Ethos' authorization restricts such a program, but the program could still corrupt data that it must be authorized to manipulate. System administrators likewise present a particular concern. Thus administrative and security-sensitive software development processes must be protected using external means. Covert channels and crooked hardware are also beyond this paper's scope.

4 Design

We discussed the complexity of programming and configuring current systems in §2, and we will provide a concrete, in-depth example in §5. Here we introduce Ethos and discuss the three strategies that influenced Ethos' design:

S1 Developers use a simple API.

S2 Application errors cannot cause a failure in (1) network encryption, (2) the authentication of remote principals, or (3) authorization controls.

S3 Security configuration requires only configuring and populating (1) a user database and (2) an authorization policy. The database and policy are *universal*, i.e., sufficient for each application on a Ethos system.

4.1 Encryption

Ethos encrypts *all* network communication to provide confidentiality and integrity. All communication between a pair of hosts flow within a single cryptographic tunnel, and Ethos multiplexes application connections within these tunnels. Ethos establishes a tunnel when an application requests the first connection to a new host. To establish a tunnel, an Ethos client includes a public key and nonce in the first packet sent to a server (the client knows the server's public key by the authentication database described below), and both parties establish a symmetric key using Diffie-Hellman (DH) key exchange.

Because Ethos' network encryption is implemented in the kernel, an Ethos application cannot prevent networking from being encrypted. In contrast, traditional techniques implement functionality by libraries. Libraries share address space with their applications, so application failures can cause protections to be bypassed.

4.2 Authentication

Within host-to-host encrypted network tunnels, Ethos identifies the user associated with an individual connection by using a public-key-based authenticator. If the receiving host is able to verify the authenticator, then this host associates the user's public key with the network connection.

Configuring Ethos to know every principal is clearly impractical. Given Ethos's universal authentication and the scale of the Internet, a mechanism is needed to authorize communication with unknown parties. Consider a remote **stranger**, an individual who is not known a priori to an Ethos system (i.e., he does not exist in Ethos' local database or distributed PKI, so Ethos does not know his real-world identity). Like a traditional user, a stranger can generate a key pair and attempt to communicate with an Ethos system. In some cases, a client user may even choose not to provide any authenticator at all; the server treats such a connection as anonymous and assigns the user a random identifier. (Unlike with strangers, this identifier is not persistent across connections.) Whether Ethos accepts a given stranger or anonymous connection request is a matter of authorization.

4.3 The Ethos network system call interface

For all Inter-Process Communication (IPC)—including networking—Ethos uses a single mechanism, whereas POSIX provides sockets, pipes, message queues, shared memory, and signals. Ethos' mechanism is made up of the advertise, import, and ipc system calls.

```
serviceFd ←advertise(serviceName)
```

To register the intent to provide a service, an Ethos application calls advertise. The argument to advertise is a service name, which we describe in §4.4.

```
netFd, user ←import(serviceFd)
```

The import system call takes as an argument a service file descriptor that was obtained from a previous call to advertise. Import returns a network file descriptor and remote user.

```
netFd ←ipc(serviceName, host)
```

An ipc call attempts to make an encrypted connection to a service. This system call takes two parameters: serviceName—the service name to connect to—and host—the remote host name. Setting host to nil implies a local connection; otherwise, Ethos resolves (using either a local database or trusted PKI) host to an Internet Protocol (IP) address and public key as a part of servicing the system call.

Read reads an object from a file descriptor. Peek behaves like read but does not remove the object from the file descriptor's stream. Write adds an object to a stream. Beyond the scope of this paper is Ethos' type system which allows read and write to automatically deal with architecture-specific issues of endianness, word size, and alignment.

For our case study we need two more system calls: fdSend, and fdReceive. These system calls concern **virtual processes**, which are processes fabricated on-demand as the result of an fdSend. There is at most one virtual process executing per virtual executable-user pair. fdSend sends a tuple of file descriptors fd to the executable program (i.e., an executable file in Ethos' filesystem), which runs as a virtual process owned by the specified user.

```
fdSend(fd[], user, program)
```

If program is already running on behalf of user, then Ethos will provide it with the file descriptors. Otherwise, Ethos will execute program, running as user, before doing the same. The resulting virtual process calls the corresponding system call, fdReceive, to receive the file descriptors.

```
fd ←fdReceive()
```

4.4 Authorizing system calls

Ethos integrates Discretionary Authorization Control (DAC) and Mandatory Authorization Control (MAC) in its Language for Expressing Authorization Properties (LEAP) [18]. Ethos has fewer, more abstract system calls than traditional OSs. This simplifies authorization policy specification compared to, for example, SELinux. Additionally, Ethos' authorization system has more information available to it. In particular, Ethos can make authorization decisions based on a remote user because network authentication is performed by the OS.

Ethos associates all network connections with a special filesystem node called a service name. Passing s as the service string argument to advertise and ipc corresponds to the filesystem path /services/s. Unlike TCP/UDP port numbers, which have a very small name space and thus must be reused, Ethos network service names are unbounded. Hence each name is one-to-one with a service, simplifying authorization. Ethos can also differentiate security-levels by service name, so that, for example, Top Secret mail might use a different name than unclassified mail.

Ethos **objects**—including files, directories, program executables, and service names—bear a label. LEAP regulates the operations **subjects**—user/program pairs—can invoke on objects. The LEAP permissions we will discuss here are:

c	create a file
r	read a file descriptor
w	write a file descriptor
x	execute a program
adv	advertise/import a service
ipc	ipc to a service
fdS	send a file descriptor to a process

LEAP grants permissions as follows, where r is a permission (e.g., read here); l is an object label; p is a program label; and G is a group, defined by a set of user public keys:

$$r(l) = [p, G]$$

For user u with key $pk_u \in G$, we simplify the conjunction:

$$r(l) = [p, G] \ \wedge \ pk_u \in G$$

to the more concise:

$$p, u \rightarrow r(l)$$

to mean that p, when running on behalf of u, has $r(l)$. In specifications, group names are written $gs.G$. To support DAC semantics a special group notation, $gs\{\$\}.G$, indicates a group specific to the owner of the object in question.

LEAP also specifies two restrictions, which resemble permissions. These are in, a restriction on the users who may connect to a local service, and out, which restricts the remote hosts that may provide a service to local programs.

Thus *in* governs advertise/import and *out* governs ipc. Both of these restrictions apply to labels, but describe host groups (*out*) or user groups (*in*) instead of subjects. For example,

$$in(l) = U$$

restricts services labeled l so that they may be accessed only by users with public keys in U. We will now describe the permissions needed by various operations.

ipc On the client side, an administrator may restrict to which services and hosts a subject may connect. As we have discussed, networking-related permissions are derived from filesystem nodes, contributing to consistent authorization mechanisms. An outgoing connection to the host h for the service with label l requires:

$$p, u \rightarrow ipc(l) \ \wedge \ h \in out(l).$$

Ethos' host protection is stronger than IP address-based restrictions because it cryptographically authenticates servers.

advertise and import On the server side, Ethos restricts advertise and import. Ethos performs user authentication at the system layer; if import returns, Ethos has authenticated the remote user and he is authorized to connect to the given service. As mentioned before, authorizing strangers allows even previously unknown users to maintain a unique user ID. To advertise the service labeled l and import a connection from the remote client user u_c requires:

$$\text{advertise: } p, u \rightarrow adv(l)$$

$$\text{import: } p, u \rightarrow adv(l) \ \wedge \ u_c \in in(l).$$

fdSend/fdReceive To fdSend a file descriptor to a program labeled l requires $p, u \rightarrow fdS(l)$. The fdReceive call requires no special permission. Any virtual process may receive a file descriptor using fdReceive, but successive reads and writes are subject to access controls.

read/peek/write The read and peek calls require $p, u \rightarrow r(l)$, and write requires $p, u \rightarrow w(l)$.

5 Evaluation

Our evaluation focuses on the security of Ethos. We wrote a very simple but robust messaging system, eMsg, to study the security properties that result from Ethos' networking API. Throughout this case study, we compare eMsg to Postfix 2.8.7, a popular email server. Postfix's primary author is a security expert; thus we consider Postfix as a rough upper bound on large POSIX network application quality.

5.1 Programming

Postfix has many more features than eMsg, so we focus on comparing how each attempts to make use of the network in a secure way. POSIX provides much weaker security guarantees than Ethos and its API is more complex. Both of these factors complicate developing and configuring services.

Postfix (POSIX) Postfix is made up of several programs so that each may be granted minimal privileges. We focus on how Postfix receives a message from an authenticated sender and delivers it to a local user's incoming spool. Here the key components are: **master** which runs as root and executes other programs on demand; **smtpd** which runs as the pseudo user *postfix*, accepts SMTP connections, and enqueues messages received; **qmgr** which runs as the pseudo user *postfix* and manages the message queue; and **local** which transitions its effective UID between *postfix* and the message recipient in order to deliver mail to the recipient's incoming mail spool. To support this functionality, local maintains a real UID of root.

For security, application developers need to get network authentication and encryption right. Programmers must properly invoke security libraries and must choose appropriate cryptographic parameters, including algorithms and key size. Mechanisms such as seteuid are also troublesome—seteuid semantics are complicated and its use is susceptible to attack [3].

For encryption, Postfix relies on the OpenSSL library. This library is invoked in several modules. We count only OpenSSL function calls rather than all encryption-related Lines of Code (LoC), because it is difficult to isolate the latter from total LoC. Postfix makes 98 calls of OpenSSL functions (those that start with BIO_, ERR_, SSL_, or X509_). We estimate that at least six LoC are needed for each of these function calls to set up parameters and deal with error conditions, resulting in almost 600 lines of application code to invoke OpenSSL.

We used a hybrid approach to measure the complexity of the use of Cyrus Simple Authentication and Security Layer (SASL) network authentication in Postfix. First, there are 987 LoC in Postfix's xsasl directory; this code's single purpose is authentication (we excluded Dovecot SASL LoC). Then, we found 12 calls of functions beginning with xsasl_ in the rest of the code. Including OpenSSL calls, almost 2,000 lines of application code are needed to protect networking.

Significant application code to support robust networking is not unique to Postfix. Another study found that 9,000 LoC or 37% of the Internet Message Access Protocol (IMAP) service code in Dovecot was dedicated to network confidentiality, integrity, and authentication [16]. Similar patterns emerge when examining most servers that are designed to run on POSIX.

A failure in any of the Postfix code described above could result in a lack of network encryption, weak network encryption, or incorrect authentication. Particularly dangerous is smtpd (because it directly interacts with the network) and local (because it can activate root privileges via seteuid). These programs are 9,100 and 2,500 LoC, respectively; mitigating the risk they pose requires extensive code auditing.

Failures in such code are common occurrences. Encryption, authentication, and authorization-related failures accounted for 9 of CWE/SANS' *Top 25 Most Dangerous Software Errors* in 2011 [15]. The US National Vulnerability Database lists over 50 flaws found in or in the use of OpenSSL in the last three years. Postfix itself was recently affected by a security flaw due to its misuse of TLS [6]. Fixing this flaw in Postfix does not guarantee that it does not exist in another application—Pure-FTPd and the Cyrus IMAP server [7, 8] were later found vulnerable.

eMsg (Ethos) Figure 1 contains a pseudo-code listing of eMsg, followed by a diagram depicting the system's process interaction. We wrote eMsg in Go and it contains 698 LoC. The actual code contains error handling, but for brevity the pseudo code does not.

Ethos provides network protections at the system layer; thus eMsg itself dedicates zero LoC to network confidentiality, integrity, authentication, and authorization. eMsg is made up of five programs, four of which demonstrate Ethos' network protections: **MsgWrite** is used to compose a message and store it in the sender's outgoing spool. **MsgSend** is a virtual process that is either invoked by msgWrite or every ten minutes. MsgSend delivers outgoing messages to a remote msgDistributor. **MsgDistributor** accepts messages

```
1 msg ← readMsgEnteredByUser()
2 writeVar("~/out/" + gettime (), msg)
3 fdSend([FdNull], getuser (), "msgSend")
```

(a) Client, msgWrite

```
4 do forever
5   wait on fdReceive() or beep(600)
6   for filename in "~/out"
7     msg ← readVar(filename)
8     netFd ← ipc("msg", msg.To.Host)
9     write(netFd, msg)
10    removeFile(filename)
```

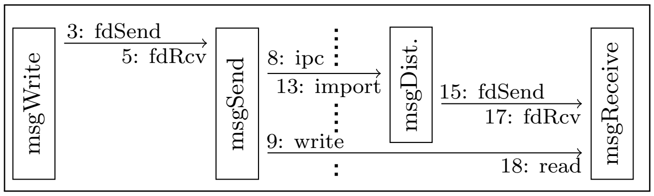
(b) Client, msgSend

```
11 listenFd ← advertise("msg")
12 do forever
13   netFd, user ← import(listenFd)
14   msg ← peek(netFd)
15   fdSend([netFd], msg.To, "msgReceive")
```

(c) Server, msgDistributor

```
16 do forever
17   fd ← FdReceive()
18   msg ← read(fd)
19   writeVar("~/in/" + gettime (), msg)
```

(d) Server, msgReceive



(e) Process interaction, with corresponding line numbers

Figure 1: eMsg application in pseudo code

received over the network, peeks at the recipient, and executes msgReceive with the recipient's credentials. **MsgReceive** runs as a virtual process with the recipient's credentials and writes incoming messages to the recipient's spool. **MsgView** displays the messages in a user's local incoming spool. It does not directly interact with the network, hence its listing is not included in Figure 1.

A user runs msgWrite to create a message. msgWrite writes the message to the user's outgoing spool and notifies msgSend of the outgoing message via fdSend (Line 3). The purpose of this fdSend is to wake up the receiving process.

MsgSend calls fdReceive and beep (a timer) and then blocks until one of the system calls completes (Line 5). In either case, msgSend reads the user's outgoing spool and attempts to send each message over the network by calling ipc and write (Line 9); this communication is protected by Ethos' implicit encryption.

To receive a message, msgDistributor listens for connections to the "msg" service. As described before, import (Line 13) returns only for authorized remote users. When import does return, msgDistributor calls peek to obtain the recipient of the message without disturbing the stream. MsgDistributor then uses fdSend (Line 15) to pass the network file descriptor to msgReceive, a virtual process running on behalf of the recipient which writes the message to the recipient's spool.

Of the components in eMsg, only msgDistributor needs to have its code audited for the security considerations discussed here, but msgDistributor is very simple—35 LoC. The audit must show that the program always calls fdSend (Line 15) with the recipient as the user argument. A violation would cause a message to be delivered to the wrong recipient. As we discussed, the corresponding Postfix audit requires examining over 11,600 LoC.

5.2 System administration

Even organizations with many skilled administrators fail to protect their systems [17]. Thus simplifying system administration is also a key goal of Ethos.

Postfix We focus on configuring a subset of Postfix's functionality—setting it up to receive mail using SMTP—and pay attention only to security configuration decisions.

Assuming 400 words per page, Postfix's TLS documentation [1] spans some 30 pages. Properly configuring Postfix to require TLS—including client-side certificate verification—and restricting TLS to use only strong cryptographic ciphers and protocol versions requires nine configuration points. Misconfigurations can result in accepting connections not secured by TLS, accepting clients with improper credentials, or employing weak encryption algorithms or protocols.

Postfix uses SASL for authentication, and its SASL documentation is approximately 16 pages long. The configuration of SASL requires five configuration points in Postfix and several in Cyrus SASL. Here misconfigurations can result in passing secret authentication credentials over an unencrypted channel or accepting a connection without authentication.

Next, an administrator must configure the authentication database used by Cyrus SASL. This is generally done using Pluggable Authentication Modules (PAM) which has its own configuration parameters. Misconfiguring PAM can result in authentication failures. In general, variations of all of these steps must be repeated for each server installed.

Finally, an administrator must configure the system's authorization policy to restrict Postfix. On Linux, this includes installing and possibly modifying a distribution's SELinux policy. The reference SELinux policy distributed by Tresys in July of 2011 contained 650 lines in its Postfix module.

Ethos authorization On Ethos, the only security settings are the authorization policy and authentication database. The system administrator reads two pages of eMsg documentation, which describes the file paths used (for both files and networking) and suggests how they should be protected. While the administrator must also learn LEAP, LEAP applies equally to *every* application. Thus we focus on authorization policy specification.

Figure 2 describes an authorization policy for the eMsg components which we described in §5.1. Figure 2a shows the filesystem labels associated with eMsg. The terminal at which a user writes and views messages bears the label terminal. eMsg places outgoing messages in an outgoing spool directory, labeled spoolOut; the incoming spool directory bears the label spoolIn. Each program has a program-specific label (e.g., msgWrite bears the label msgWrite). We described the correspondence between a service name and filesystem node in §4.4; here the node bears the label svcMsg.

Figure 2b lists a simplified version of the corresponding LEAP policy (without label or group definitions). The policy references two group sets, *ug* and *hg*, respectively collections of users and hosts. Each of these group sets contains two groups, *in* and *out*, concerning incoming and outgoing connections. The actual policy is 30 lines, plus lines for defining group membership.

First, the policy regulates msgWrite. When run by a user in group *ug.out*, msgWrite can read a message that a user enters at the terminal (Line 2), write the message to the outgoing spool (Lines 3 and 4), and invoke msgSend as a virtual process (Line 5). Here the notation $ug\{\$\}.out$ further qualifies permissions; msgWrite, running on behalf of some user u, may create or write *only to directories owned by u.* (label spoolOut and $pk_u \in ug.out$ must still be satisfied.)

Path	Label
Terminal	terminal
Each program	*Program name*
/user/*user*/out/*	spoolOut
/user/*user*/in/*	spoolIn
/services/msg	svcMsg

(a) eMsg LEAP filesystem labels; spool queues are per user

```
 1 x(msgWrite)    = [shell, ug.out]
 2 r(terminal)    = [msgWrite, ug.out]
 3 c(spoolOut)    = [msgWrite, ug{$}.out]
 4 w(spoolOut)    = [msgWrite, ug{$}.out]
 5 fdS(msgSend)   = [msgWrite, ug.out]

 6 r(spoolOut)    = [msgSend, ug.out]
 7 ipc(svcMsg)    = [msgSend, ug.out]
 8 out(svcMsg)    =  hg.out
 9 w(svcMsg)      = [msgSend, ug.out]

10 x(msgDist)     = [init, nobody]
11 adv(svcMsg)    = [msgDist, nobody]
12 in(svcMsg)     =  ug.in
13 r(svcMsg)      = [msgDist, nobody]
14 fdS(msgRecv)   = [msgDist, nobody]

15 r(svcMsg)      = [msgRecv, ug.in]
16 c(spoolIn)     = [msgRecv, ug{$}.in]
17 w(spoolIn)     = [msgRecv, ug{$}.in]
```

(b) LEAP specification (msgView not shown)

Figure 2: eMsg authorization policy

Lines 7–9 allow msgSend's use of ipc and write; the program can connect and write to the service labeled svcMsg if running as a user in *ug.out*. Line 8 restricts the eMsg servers msgSend may connect to to those in the group *hg.out*.

On Line 12, the policy restricts incoming connections to those originating from users in *ug.in*; such principals are cryptographically authenticated by Ethos. Of note in the remainder of the specification are Lines 16 and 17, which ensure msgReceive writes only to the recipient's own spool.

6 Conclusion and further work

Ethos provides network security with zero lines of application code. In contrast, Postfix requires almost 2,000 lines of security-sensitive code. Ethos' networking system calls *implicitly* invoke encryption, authentication, and authorization, thereby shrinking application code bases and reducing their attack surface. An application programmer cannot fail to invoke or incorrectly invoke these protections.

In Ethos, network security does not require any application-specific configuration. Instead, system administrators configure a system-wide user database and an authorization policy. This saves administrators from reading application security configuration documentation (typically dozens of pages), configuring services (typically dozens of configuration points), and auditing thousands of lines of code. System administrators can thus better secure applications and are less dependent on programmers for application and overall system security.

Ethos' more abstract systems calls give rise to a more abstract authorization policy language. Ethos administrators use LEAP to directly restrict actions performed by remote users. This is in contrast to SELinux which requires cooperation between properly-implemented application-based network authentication and authorization policy.

We are beginning to shift our focus from kernel development to user space. Projects underway include writing Go packages to aid application development, designing a graphics subsystem, and developing substantial applications. We are focusing on particularly security-sensitive applications which will most stress Ethos' other security services, much like we investigated networking here. This will allow us to certify our design and ultimately reimplement and verify our research prototype for high assurance.

7 Acknowledgments

This material is based upon work supported by the US National Science Foundation under grant CNS-0964575. We also thank Ameet Kotian for his early work on the Ethos network stack, Wenyuan Fei for his work on Ethos' authentication infrastructure, and our anonymous referees.

8 References

[1] Postfix documents. http://www.postfix.org/documentation.html.

[2] Bittau, A. et al. The case for ubiquitous transport-level encryption. In *USENIX Security* (2010).

[3] Chen, H. et al. Setuid demystified. In *USENIX Security* (2002).

[4] Corbato, F. J. et al. Multics—the first seven years. In *Spring Joint Computer Conference* (1972).

[5] Cox, R. et al. Security in Plan 9. In *USENIX Security* (2002).

[6] CVE-2011-0411. National Vulnerability Database, March 2011.

[7] CVE-2011-1575. National Vulnerability Database, May 2011.

[8] CVE-2011-1926. National Vulnerability Database, May 2011.

[9] Fahl, S. et al. Why Eve and Mallory love Android: an analysis of Android SSL (in)security. In *CCS* (2012).

[10] Georgiev, M. et al. The most dangerous code in the world: validating SSL certificates in non-browser software. In *CCS* (2012).

[11] Ioannidis, S. et al. Implementing a distributed firewall. In *CCS* (2000).

[12] Jaeger, T. et al. Leveraging IPsec for mandatory access control across systems. In *Proc. of the Second International Conference on Security and Privacy in Communication Networks* (2006).

[13] Keromytis, A. D. et al. The STRONGMAN architecture. In *DISCEX* (2003).

[14] Loscocco, P. and Smalley, S. Integrating flexible support for security policies into the Linux operating system. In *Proc. of the FREENIX Track* (2001).

[15] Martin, B. et al. 2011 CWE/SANS top 25 most dangerous software errors. Tech. rep.

[16] Radhakrishnan, M. and Solworth, J. A. NetAuth: Supporting user-based network services. In *USENIX Security* (2008).

[17] Rahman, H. A. et al. Identification of sources of failures and their propagation in critical infrastructures from 12 years of public failure reports. *IJCIS 5*, 3 (2009), 220–244.

[18] Solworth, J. A. and Sloan, R. H. Security property-based administrative controls. In *ESORICS* (2004).

[19] Zeldovich, N. et al. Making information flow explicit in HiStar. In *SOSP* (2006).

[20] Zeldovich, N. et al. Securing distributed systems with information flow control. In *NSDI* (2008).