Authentication in Ethos

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

Jon A. Solworth
University of Illinois at Chicago
solworth@ethos-os.org

ABSTRACT

Authentication primitives should be simple, general, and robust against attack. We describe the authentication mechanisms of Ethos, an experimental, clean-slate operating system that has been designed for security. We reexamine and redesign software layering for authentication, and evaluate the resulting security properties. In Ethos, integrated network security and simplified local authentication shrink application code size and prevent application-based authentication failures, making systems more robust against attack.

1 Introduction

The abstractions used to provide Operating System (OS) authentication are fundamental to a system's security. Despite this, integrating authentication in a way which is simple, general, and robust against attack has proven elusive.

Authentication identifies the **principals**, representing users, hosts, and other entities, with which a system interacts. Users are the most important principals; they explicitly or implicitly direct a system to perform actions on their behalf. In a distributed system, a user might be represented by many principals, some **identified**, others **strangers** (long-lived but not identified), and yet others **anonymous** (used for a single connection). Systems must authenticate both local and remote users.

Subtle vulnerabilities can arise as a consequence of authentication system design. In **authentication privilege escalation** [52], a process gains unintended privileges using authentication primitives[1]. Here there are two possibilities:

(1) Processes p_0 and p_1 are owned by different users; p_0 crafts p_1's state to trick p_1 into doing something.

(2) A process requires elevated privileges for authentication and is attacked before it drops them.

For an example of (1), p_0 might set an environment variable to specify the file system location from which dynamic libraries are loaded and then exec a setuid-program p_1 [59]. In fact, p_0 can craft p_1's state in many ways, including environment variables, file descriptors, and command-line inputs. A naïve programmer may be unaware of such attacks, and even sophisticated programmers make mistakes. Libraries pose particular problems because of unavailable source code and revisions that affect previously installed programs. In (2), elevated privileges increase the impact of vulnerabilities.

This paper introduces new, safer-to-use authentication mechanisms and describes their implementation in Ethos, an experimental OS designed to make applications safer from attack. Ethos' authentication contributions are as follows:

- All network interaction is authenticated before any application code runs.
- Ethos is free from authentication privilege escalation.
- Ethos has new, simple-to-use authentication system calls.
- System administrators rely on system-level authorization and network authentication, and thus do not need to audit application code for these properties.

In addition, Ethos guarantees high confidentiality of authentication secrets, scales to the Internet using Public-Key Cryptography (PKC), supports multiple identities per user, and supports anonymity.

The deep integration of authentication mechanisms into an OS poses several challenges beyond those for user-space implementations: First, it requires mechanisms which are sufficiently general to broadly satisfy authentication needs. Second, it requires a careful integration with other mechanisms. For example, Ethos tightly integrates system authentication with network security. Third, it places a premium on simplicity of mechanism. Fourth, it requires compatibility. Ethos places security above flexibility, and thus has a unique take on compatibility; we will describe the reasons, trade offs, and especially security implications for our approach in §6.

The deep integration of authentication mechanisms also has several benefits: First, the mechanism is inescapable, and thus protects the entire system. Second, it can provide richer information, enabling more accurate decisions to be made. Third, it enables simpler mechanism since it can rely on co-designed mechanisms in the OS. At first, it may seem that Ethos's design is too simple, but its power derives from its composition with other mechanisms. We will allude to some of these in the paper.

In the remainder of this paper, we discuss threats (§2), an overview of Ethos (§3), the design of Ethos authentication (§4), an evaluation of Ethos' design and implementation (§5), compatibility with the existing software base (§6), and related work (§7). Our evaluation focuses on Ethos' security properties and discusses performance.

2 Threats

We describe both the threat environment (general attacker capability and success) and the threat model (the threats

[1]Privilege escalation from outside of the authentication system is beyond the scope of this paper.

that Ethos authentication addresses).

Threat environment Attackers routinely compromise existing systems [35, 8, 28, 53]. The US National Security Agency (NSA) assumes its networks compromised [74], and it is estimated that 25%–35% of computers are bots, under the control of an attacker [5, 3].

Security has become a major focus for OS suppliers, and the major OSs have all been assured to the Common Criteria (CC) assurance level of EAL3 or EAL4. Since these systems have all been compromised, higher levels of assurance are needed. But above EAL4 requires both design-for-security and low complexity—neither of which describe today's popular OSs [33]. Because these attributes cannot be retrofitted, we believe existing OSs will need to be replaced.

Threat model The attacker considered by this paper has very broad access—he can run applications on the Ethos host; control remote hosts and other unprivileged virtual machines; and control network mediums. He can attempt to violate an Ethos host's security policy by observing network packets, defeating protocols [44, 27], deploying counterfeit services, or making odd requests to Ethos-hosted services. We are especially concerned with his ability to exploit inadvertent application errors with respect to authentication and even to develop applications with faulty authentication which he can then exploit[2] [64]. The attacker might have a local user account. However, we explicitly trust system administrators; thus they are outside the attacker model. We also assume the integrity of the layers below the OS.

3 Ethos overview

Ethos is an experimental, clean-slate OS prototype. Its goal is to make it easier to write, administer, and protect applications from attack. Ethos provides safer semantics; its system calls imply more security properties, thus amortizing security-sensitive code over an entire system.

System semantics affect the security of the entire system. For example, application security is affected by Programming Language (PL) semantics: The problem of fixing application buffer overflows is fixed for all applications by using a strongly typed PL. Similarly, OS semantics can have a profound impact on application robustness. Yet while PL semantics continue to evolve (e.g., Haskell, Erlang, Go, Scala), OS semantics have been relatively unchanged for decades [49].

Just as new PLs are inherently incompatible with existing PLs, Ethos is incompatible with existing OSs. Although OS provider security practices have improved markedly, they have been overmatched by an increasing and worldwide attacker capability. Thus the attackers' success in finding and exploiting bugs in every software stack demands a new generation of designed-for-security OS semantics. Defenders have been repeatedly humbled by the attacker, so we take the conservative approach of reducing complexity and including in Ethos only strong security mechanisms. Compatibility is provided outside of Ethos (see §6).

Ethos' semantics are designed for security, and thus we focus on the effect of its design. NSA calls this the **strength** of the design—the ability of a perfect implementation to withstand attack [7]. Strength is difficult to obtain; NSA spent ten years designing and analyzing the strength of battlefield radios [25]. A new generation of high-strength OSs can shift much of the burden of securing systems from application developers to the OS.

[2]`http://www.rikfarrow.com/Network/net0702.html`

The overall ability to withstand attack is **robustness**—the combination of strength and assurance. Thus our focus complements the substantial and far better understood work on *implementing* reliable OSs, for example by microkernels [4, 24, 67].

Ethos currently provides memory paging, processes, encrypted networking, and a filesystem. We have completed 39 system calls and related Go packages. On this, we have built a shell, basic tools, a remote shell utility, and a networked messaging system. We have ported Go and Python to Ethos. We use C, but only to implement the OS and support other PLs.

3.1 Ethos system calls

The most interesting part of Ethos is its system calls, which target a higher level of abstraction than UNIX. The most relevant to authentication are the nine system calls in three categories shown below.

Proc.	fork	create a child process
	exec	change process executable
	exit	terminate a process
Auth.	authenticate	authenticate a local user
	fdSend	send a file descriptor
	fdReceive	receive a file descriptor
Net.	advertise	offer a service
	import	accept a connection to a service
	ipc	connect to a service

As in POSIX, the child of an Ethos fork inherits the authentication credentials of its parent. However, Ethos' authentication mechanisms are quite different from those of POSIX: there is no setuid system call and the user associated with a process never changes (§4.1). In addition, exec cannot change the authentication credentials of the process—Ethos does not have a setuid bit. Instead, Ethos provides fdSend/fdReceive (§4.1).

Ethos networking and local Inter-Process Communication (IPC) both use the same system calls: advertise, ipc, and import (§4.3). Network communication is encrypted and protected against tampering by cryptographic checksums. IPC communication remains as cleartext, but Ethos protects it against observation and tampering by memory management. In either case, all communication is subject to authorization on each host participating in the communication.

For local authentication, where the user is physically present at the computer, Ethos provides an authenticate system call to establish a user's identity by password (§4.2). Passwords are a relatively weak mechanism [14]. However, local authentication requires physical proximity to the computer, eliminating access by the vast number of (remote) attackers. If stronger local authentication is required, multifactor authentication using a physical token or biometric can be used [26, 60].

Remote authentication is inherently different than local authentication. Here, direct use of smart cards or biometrics is inappropriate, and passwords are unreliable, due to spoofing, key logging, and network timing attacks [31, 62]. Furthermore, large botnets—containing upwards of a million nodes—allow for a staggering number of password attempts, and have been very effective [41]. Ethos' ipc/import guarantee that all network connections are authenticated using PKC (§4.3).

3.2 Implementation

Ethos is built on top of Xen, a Virtual Machine Monitor (VMM) [9]. A VM provides a small and fixed set of devices regardless of underlying hardware. Ethos is thus compatible with all the x86 devices that Xen supports. This dramatically reduces Ethos' code base and decreases the vulnerabilities associated with device drivers [16]. By encrypting both networking and disk storage, neither integrity nor confidentiality depend on VMM device drivers (although availability does).

Ethos depends on its VMM's security. Orthogonal work to reduce vulnerabilities in Xen [56], disaggregate Dom0 [46, 17], produce alternative hypervisor technologies (e.g., Nova [63] and OKL4 [30]), and verify hypervisors (e.g., MinVisor [21]) will improve the robustness of the layer below Ethos.

When Ethos' design proves to have sufficient strength and usability, we will re-implement Ethos to achieve high assurance. This includes a microkernel implementation [4, 24, 67], a minimalist VMM, and proof of correctness [37]. Given that we are still working on user-space evaluation, such a reimplementation would be premature.

4 Ethos Authentication Design

Ethos authentication is deeply integrated into the system's networking and process mechanisms: PKC is integrated with networking, password-based local login is differentiated from cryptographic-based network authentication, cryptographic keys are managed by Ethos, and both local and network authentication use a common mechanism for creating credentialed processes. Ethos authenticates all local and remote users.

An Ethos host uses two different authentication services. A Local Authentication Service (LAS) contains the name and User ID (UID) of each user who can physically log onto the system, and each host maintains its own LAS. A Remote Authentication Service (RAS) contains user names and their public keys, server names and their public keys, and groups. Multiple RASs form a distributed Public Key Infrastructure (PKI) [61].

An Ethos **host keystore** holds the host's own public and private keys as well as the public key (root of trust) for the RAS. For each local user, there is also a **user keystore** that holds his public and private keys [20]. To better isolate private keys, Ethos never shares them with applications. Instead, Ethos implements system calls for console login and signing operations and encrypts/authenticates all network traffic. All keystores are further protected by Trusted Platform Module (TPM) [6, 10], and user keystores may only be unlocked as we will describe in §4.2. Ethos weakens cold boot attacks [29] by zeroing residual keys. Ensuring that cryptographic implementations are resistant to side-channel attack [11] is simplified because all these operations happen in one place—the OS.

All users are visible to Ethos. Ethos services do not maintain application-level user databases, as does MySQL or Apache with mod_authn_file. A user's application interaction in Ethos can be customized solely through authorization (§4.5).

4.1 Virtual processes

Ethos' authentication mechanisms are centered around its **virtual processes**, per-user processes created on demand. To invoke a virtual process, an application sends a tuple of file descriptors (typically, for a network connection) to it and specifies its user. The Application Programming Interface (API) to send descriptors is:

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

where fd[] is a tuple of file descriptors, u is the user that will own the virtual process, and program is the file containing the virtual process's executable code. fdSend's tuple of file descriptors simplifies its failure semantics—sending a sequence of file descriptors will either completely succeed or completely fail. We call a process which calls fdSend a **distributor**, as it distributes file descriptors to the proper user's virtual process.

fdSend invokes a virtual process (§4.6), which can receive the sent file descriptor tuple using fdReceive:

```
fds ←fdReceive()
```

Naturally, virtual processes require careful authorization (§4.5). Ethos virtual processes are used in one of two ways: (1) Typically, when sending a network file descriptor, a distributor sets u to the remote user associated with the connection. We call this a **remote-user virtual process**. (2) Alternatively, the distributor can specify any user using fdSend's explicit u parameter, called **arbitrary-user virtual process**. The latter allows for local authentication and mail-like messaging programs (where the owner of the virtual process is the message recipient, rather than the message sender/remote user). These two modes are summarized in Table 1 and discussed below.

Ethos is free from authentication privilege escalation. Because Ethos processes never change their owner, their privileges do not change as a result of authentication. A virtual process is not created using fork, so it has no predecessor process. In that sense, it is like init on Unix systems. Thus the environment of a virtual process is defined by Ethos and is unaffected by any process; moreover only the appropriate distributor may invoke it with fdSend. Finally, the permission for authentication is narrowly scoped: it does not allow extraneous actions.

4.2 Local authentication

Ethos allows authentication by password (or physical token/biometric) only when the user can physically interact with the computer—never over the network. For obvious reasons, we call this **local authentication**.

The API to authenticate a user such as Alice locally is:

```
authenticate()
```

The authenticate system call establishes a trusted path [22, 73] and prompts Alice for her password. Alice must input a password locally (not over the network), and the password is never seen by the application. Ethos notes Alice as the owner of the calling process and then verifies her password; based on this check, authenticate returns true (authenticated) or false (rejected). Ethos rate limits authentication and logs its use, providing resistance to excessive local password attempts at the system level.

```
1 do forever
2   user ← read (stdinFd);
3   fdSend ([stdinFd, stdoutFd, stderrFd],
        user, "loginVP")
```

(a) Distributor distributes terminal fds to virtual processes

```
4 stdinFd  ← fdReceive ()
5 stdoutFd ← fdReceive ()
6 stderrFd ← fdReceive ()
7 if authenticate()
8   // User authenticated; exec shell.
```

(b) loginVP checks password and then does user startup

Figure 1: Local authentication

On a client host—one where the user would locally authenticate—Ethos unlocks the user's keystore when authentication succeeds. On a server machine, user keystores are always unlocked. While the protections on server keystores are weaker, keys are never shared across hosts. (Thus all keys are compound keys in the sense of [40].) Server keystores are useful only for server farms when keys are used to authenticate across backend services.

Authenticate does not require any privilege. It is safe for any application to invoke, as it does not disclose passwords to the application or change the process's OS state (as does, e.g., POSIX's setuid).

Figure 1 shows the key code in Ethos's login, made up of two programs totaling 68 lines of Go. Alice first provides her name to the login distributor (Line 2). The distributor passes this name to fdSend, invoking the virtual process loginVP and providing the terminal's file descriptors (Line 3). (stdinFd, stdoutFd, and stderrFd correspond to POSIX's 0/stdin, 1/stdout, and 2/stderr.) After receiving the file descriptors with fdReceive, loginVP calls the authenticate system call (Line 7). Of course, loginVP is already running as Alice when it makes this call; the call to authenticate simply means that login will not proceed without a valid password.

4.3 Network authentication

Ethos cryptographically authenticates all network connections, so applications cannot fail to authenticate, authenticate a network connection using a password, or incorrectly authenticate. Ethos' use of PKC means that authentication does not require the sharing of secrets, users can create their own key pairs, and each public key is guaranteed unique. Because of the last property, public keys are Universally Unique IDs (UUIDs) that can serve as UIDs [68, 55, 36], even if the user's real-world identity is not known. This enables uniform support of stranger, anonymous, and identified principals.

An obvious problem is how to associate a name—and therefore a real-world identity—with a UID/public key [54]; we solve this problem using a PKI. Current PKIs have the reputation, deservedly, of being complex and having poor performance, thus we are designing an Internet-scale PKI [61].

Rather than attempt to describe our PKI here, we will describe a simplified version with a single authentication server (denoted S). We note that S is fully capable of satisfying all the authentication needs of an organization; it is structurally identical to an organization which maintains its own password authentication server. We'll assume that Alice (denoted A) is to be authenticated by host Bob (denoted B). Let k_X be X's key, $X \rightarrow Y$ means X sends to Y. The sequence is as follows:

Party	**Message/Action**
$A \rightarrow B$	k_A and an authenticator created with k_A
B	validates the authenticator using k_A
$B \rightarrow S$	"whose key is k_A?"
$S \rightarrow B$	a certificate signed by k_S: "k_A is A's key"
B	verifies the certificate using k_S.

Here B only needs public key k_S. If S does not know key k_A, then A is either a stranger or anonymous. In either case, host B's access controls determine whether to accept A's request.

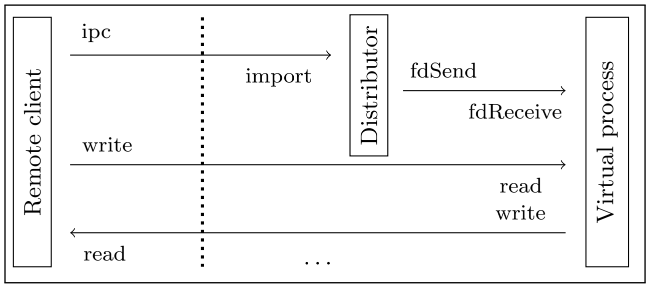

Figure 2: Interaction of client and server components

Figure 2 depicts an Ethos client making a connection to a distributor. The distributor accepts connections and distributes them, via fdSend, to the appropriately user-credentialed virtual process. Ethos cryptographically authenticates the remote user associated with an incoming network request, and subjects connection requests to its authorization policy (§4.5). Thus import only returns for network requests from authorized (and authenticated) users. We present pseudo code for two Ethos services and discuss them in detail in §5.

4.4 Principals, strangers, and anonymity

An Ethos user can assume different principals when making network connections on a per-host-and-service basis. This supports different identities for work, entertainment, and socialization. A principal can be a stranger (i.e., not present in a RAS), and anonymous principals result from creating a new identity on each connection to a service. Whether a service allows anonymous or stranger users is a matter of authorization (§4.5), independent of any application code. Assuming strangers protect their private key, Ethos can isolate their persistent resources as with identified users, and no one else can assume their identity during authentication.

Strong anonymity is difficult to achieve, and depends on its use. Anonymous services need, in addition to anonymous users, further protections against de-anonymizing, such as with Dissent [19] or Tor [23]. This level of anonymity is not included in Ethos at the OS level due to its latency, but Ethos provides a strong foundation on which to build such services.

Table 1: Summary of authentication components: the authorization controls show how Ethos restricts each component; and system administrators must audit each component as indicated (see §5.3–5.5)

Type	Component	Relevant authorization controls	Authentication-related auditing requirements
remote-user based	distributor	Accept network request for service	
		Send fd to virtual process	
	virtual process	Process owner = remote user	
		Commensurate access for user/program	
	client	Make outgoing connection to service	
arbitrary-user based	distributor	Accept network request for service	
		Send fd to virtual process	Confirm fdSend specifies correct recipient
	virtual process	Commensurate access for user/program	
	client	Make outgoing connection to service	

4.5 Authorization

Ethos tightly links authentication with authorization, as authentication influences authorization decisions. Authorization enables organization- or application-specific decisions, whereas authentication is one-size-fits-all.

Like Security-Enhanced Linux (SELinux) [43, 34], Ethos provides mandatory authorization controls. Unlike SELinux, Ethos' authorization system was designed along with its system calls. The users of Ethos services are specified at the OS—not the application—level, and Ethos associates a remote principal with each incoming network connection. Because of this, Ethos can directly make authorization decisions based on a remote user. A remote principal must be authorized before Ethos will provide the connection to a distributor, so unauthorized remote users never interact with Ethos application code.

Each Ethos process bears an immutable user and a label. The user is inherited from the process's parent or set by some call to fdSend, and Ethos reads the label from the process's executable in the filesystem. Objects—including files, directories, and IPC services—bear a group, owner, and a label. Ethos stores these attributes along with the object in its filesystem. In general, Ethos permissions are specified in two ways: (1) an authorization specification describes the processes which can exercise a given permission on an object and (2) certain directory prefixes confer certain permissions on the objects they contain.

Directory permissions permit only administrators to create virtual process/distributor executables, and Ethos' authorization specification restricts virtual processes to accessing connections bearing particular labels (i.e., certain services) and remote users:

- Remote-user virtual processes owned by u may only read connections owned by u.
- Arbitrary-user virtual processes owned by u may read connections of any u'.

In the first case it is unnecessary to audit distributor code, and the virtual process is restricted at the system level based on the remote user. In the second case it is necessary to audit the distributor. This work is quite modest: almost all virtual processes are expected to be remote-user, and distributors are quite small (on the order of 100 lines).

Thus Ethos governs which processes may:

(1) make an outgoing connection to a service,
(2) accept a network request from a given remote user (whether known or not) for a given service,
(3) send a file descriptor to a virtual processes running a given program,
(4) read/write file descriptors.

4.6 Implementation

Ethos maintains a hash table of user records. It initially populates this table with the users in its LAS. Ethos adds on demand other users to the table when it imports their network connections. In this case, Ethos extracts the user's UID (public key) from the authenticator present in its network protocol [48] and extracts his name from the RAS, if it is known. If the user is not known to the RAS, then Ethos sets his name to the string form of his public key. User records also contain the user's running virtual process list.

When a process calls fdSend, Ethos first checks whether running process is authorized to run the particular virtual process specified by the system call (§4.5). Ethos then checks the user record to determine if the target virtual process is already running. If not, Ethos creates it by allocating the process's page tables, switching to the process's address space, setting the process's UID to the appropriate public key, loading the program image into memory, nullifying all file descriptor table entries, and allocating a heap and stack.

Finally, Ethos adds the fdSend file descriptor tuple argument to the virtual process's incoming fdReceive queue. Ethos will then schedule the virtual process which can obtain the file descriptors by calling fdReceive.

5 Evaluation

Our evaluation focuses on the security of Ethos' authentication mechanisms. Ethos enhances security through (1) careful design of the OS interface to reduce applications' responsibility with respect to security and (2) unified security-related administrative controls at the system level. The first strategy reduces the Trusted Computing Base (TCB) by moving critical components into the OS, minimizing replication; this is in contrast to minimizing the OS but then requiring parts of the TCB to be implemented in applications. The second strategy aids administrators who must tailor Ethos' protections for their particular organization. We apply these strategies throughout Ethos; here we evaluate their effect on authentication.

We first analyze the code required for authentication in existing systems (§5.1). Next, we analyze Ethos' individual authentication-related mechanisms (§5.2) and their effect on common security holes. We present three code-based case studies: remote-user virtual process (§5.3), arbitrary-user virtual process (§5.4), and local authentication (§5.5). We also analyze system administration, comparing traditional application-by-application security properties to Ethos' security settings, which apply to *all* applications (§5.6). Finally, we consider the Ethos code base (§5.7) and evaluate performance (§5.8).

Ethos user space is coded in Go, but we compare mostly to POSIX C code. Although this is an apples-to-oranges comparison, two things should be kept in mind: First, in all networking code, Ethos applications require zero lines of code for authentication vs. thousands in POSIX. Second, we normalize this overhead by presenting the ratio of C authentication code to C application size.

5.1 POSIX attack surface

Application development on POSIX requires significant code and care to ensure the application properly authenticates users. For example, version 2.0 of the Dovecot mail server contains approximately 15,000 authentication Lines of Code (LoC), 8% of its total C codebase. Apache 2.2 provides several authentication modules, totaling over 1,800 LoC. The Kerberos authentication provider, mod_auth_kerb, contains another 1,500 LoC. In these cases, bugs in application code might cause an authentication failure. The US National Vulnerability Database [2] identified 699 authentication-related flaws in the last three years, 5% of the total reports.

The above code counts do not include external libraries. In principle, libraries should consolidate code and therefore ease system assurance. In practice, the layering of POSIX systems results in something quite different. Consider Transport Layer Security (TLS), which provides a fundamental component of network authentication on the Internet: encryption and (in sensitive environments) public-key-based client authentication. The Fedora Project's Linux distribution contains three major TLS C libraries: OpenSSL, GnuTLS, and Network Security Services (NSS). Fedora's effort to consolidate on NSS [1] and thus aid in assurance efforts began in 2007 and is still ongoing, having faced many obstacles. Furthermore, libraries must often be rewritten for different programming languages, and they often provide both weak and strong mechanisms.

These factors increase the amount of security-sensitive code, increase the application attack surface, and make it more difficult for administrators to ensure that the applications they install have the requisite protections.

5.2 Ethos Protections and the CWE/SANS Top 25

Ethos is differently layered: protections are transparently provided by the OS, ensuring that application programmers cannot incorrectly invoke or bypass them. Ethos' inescapable authentication-related protections include:

- **P1** Processes cannot change owners (§4.1).
- **P2** Applications do not have access to secrets (§4).
- **P3** All network connections are authenticated (§4.3).
- **P4** Authentication uses strong techniques (§4.3).
- **P5** Confidentiality of the authentication server database is not essential to security (§4.3).
- **P6** All communication made (client-side) or received (server-side) are authorized by user (§4.5).

Protections P1–P6 address several bug classes which result in security holes. This represents seven error classes, or 28% of the CWE/SANS' *Top 25 Most Dangerous Software Errors* [45]. Table 2 shows how Ethos' protections avoid the Top 25 vulnerabilities.

5.3 Remote-user virtual process case study

Ethos is a clean-slate design, so we wrote a new remote shell utility, resh, to evaluate Ethos' authentication mechanisms

Table 2: Authentication vulnerabilities and protections

Vulnerability	Protection
Missing authentication for critical function	P3, P6
Use of hard-coded credentials	P4
Unnecessary privileges	P1*
Broken/risky cryptographic algorithm	P2, P3, P4
Allow excessive authentication attempts	P4
Use of a one-way hash without a salt	P4
Incorrect permission assignment for critical resource†	P2, P5

*in conjunction with Ethos' authorization system
†with respect to authentication secrets

```
netFd ← ipc ("resh", "example.com")
do forever
  command ← readCommand ()
  write (netFd, command)
  response ← read (netFd)
  print (response)
```

(a) Client resh connects, issues requests, and receives responses

```
listenFd ← advertise ("resh")
do forever
  netFd, user ← import (listenFd)
  fdSend ([netFd], user, "reshVP")
```

(b) Server reshDistributor sends connections to reshVP

```
func processCommand (command)
  // Not shown: Decode and
       pipe/fork/exec.

fd ← fdReceive ()
do forever
  command ← read (fd)
  if command = "exit"
    exit ()
  response ← processCommand (command)
  write (fd, response)
```

(c) Server reshVP services shell requests

Figure 3: Remote shell application

with respect to remote-user virtual processes. We compare this Ethos application to OpenSSH.

Like most Ethos services, resh is made up of a client, distributor, and virtual process. A user can use the client to log in to a remote server and issue commands, similar to OpenSSH. Though simple, this application illustrates several of Ethos' advantages.

We wrote resh in Go; it contains 547 LoC, with 383 of that in Go YACC. Resh and its corresponding server, reshDistributor/reshVP, are shown as pseudo code in Figure 3. This design pattern can also be used for webserver-style services.

Discussion If Alice is the remote user, then both resh and reshVP run as Alice. Ethos accomplishes this without any application authentication code (Figure 3 has none), because network authentication and encryption are provided at the system level (§5.2). Furthermore, no application code ever has access to authentication secrets.

By default, Ethos' authorization policy disallows networkfacing distributors from reading and writing a network file descriptor. Thus this distributor has two network privileges: it may advertise services and import connections of the appropriate type. Once it imports a file descriptor, it may only fdSend it to a virtual process. Only a process matching the remote user can use this descriptor. If the descriptor was sent to a virtual process running on behalf of the wrong user, Ethos authorization would prevent the process from reading or writing it (§4.5). Thus this distributor can only affect availability, no matter how it is written. The resulting virtual process runs as the remote user who is restricted through Ethos' authorization policy.

OpenSSH implements its protections very carefully, requiring substantial effort [52]. OpenSSH also has significant responsibilities not required of an Ethos application: e.g., determining the remote user through some application-layer authentication protocol. The former requires a substantial amount of application-specific code; OpenSSH contains nearly 12,000 lines of cryptographic- and authentication-related code, 12% of its total.

Application-based techniques must be repeated for each application and not all of these are as carefully implemented as OpenSSH. Often such POSIX applications do not execute with remote user privileges. Instead, such applications (e.g., Apache or MySQL) themselves implement authorization controls, further adding to the application programmer's load.

5.4 Arbitrary-user virtual process case study

We implemented on Ethos a networked messaging application named eMsg to demonstrate an arbitrary-user virtual process. Figure 4 contains a pseudo-code listing of eMsg. We wrote eMsg in Go; it contains 698 LoC, and is patterned after such Mail Delivery Agents (MDAs) as Postfix and qmail. eMsg consists of five programs, two of which (msgDistributor and msgReceive) pertain especially to authentication:

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 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 4.

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. msgSend reads the user's outgoing spool and attempts to send each message over the network by calling ipc and write.

To receive a message, msgDistributor listens for connections. After import returns, msgDistributor calls peek to obtain the recipient of the message without disturbing the stream. msgDistributor then uses fdSend 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 incoming spool.

```
do forever
  waitOnNewMessageOrTimeout ()
  for filename in "~/out"
    msg ← readVar(filename)
    netFd ← ipc("msg", msg.To.Host)
    write(netFd, msg)
    removeFile(filename)
```

(a) Client msgSend sends spooled mail after notification

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

(b) Server msgDistributor sends file descriptors to msgReceive

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

(c) Server msgReceive reads messages/writes incoming spool
Figure 4: eMsg network components

Discussion In contrast to remote-user authentication, arbitrary-user authentication allows the client to be owned by Alice and the virtual process msgReceive to be owned by Bob. This allows msgReceive to write to Bob's incoming spool (and run Bob's mail filters) even though the sender (i.e., remote user) is Alice.

Like resh, eMsg dedicates zero LoC to network authentication. It differs from resh in that authorization must be more permissive: the distributor can read messages (necessary to determine the recipient) or route a file descriptor to the wrong user. Furthermore, the msgReceive virtual process may read from any network file descriptor from the distributor, not just those that are owned by the virtual process's user; so here misrouted messages can be read. Thus msgDistributor's code must be audited to ensure it will not deliver a message to the wrong recipient, but msgDistributor is very simple—35 LoC. The audit assures that it always calls fdSend with the proper recipient as the user argument. This is much less work than that required of a POSIX messaging system which implements its own authentication; for example, an audit of Postfix requires checking its use of OpenSSL and Simple Authentication and Security Layer (SASL).

5.5 Local authentication case study

As mentioned in §4.2, Ethos prevents password authentication over the network. The authenticate system call cannot be used for networked communication since it requires that the password be typed on a physical, local keyboard. The local login distributor (Figure 1) is particularly restricted; it is limited to reading from stdinFd and calling fdSend. Thus the only file descriptors the login distributor may pass to fdSend are the terminal file descriptors: stdinFd, stdoutFd, and stderrFd. The distributor may prompt for a username and pass the terminal file descriptors to the corresponding loginVP virtual process using fdSend. A compromised distributor might fdSend to the wrong user's loginVP, but that would cause the authentication to fail. Furthermore, since authenticate is a system call that does not release user secrets to user space, this breach could not be used by one user to collect the password of another. The distributor can only affect availability. This leaves loginVP which runs as the target user—it has no special privileges. The administrator needs to check that loginVP calls authenticate; however, loginVP is unique—there is only one local login program and it is distributed with Ethos.

5.6 System administration

Configuring POSIX The presence of authentication code in POSIX applications also has consequences for system administrators. Even small but nonetheless disparate configuration requirements—such as disabling the use of passwords by Secure Shell (SSH)—combine to create overall complexity. Performing these tasks takes inordinate time and skill [47], and is different for each application. As a result, configuration errors remain common [70]. Even the most security-sensitive organizations are unable to adequately assure their systems [74], and administrators routinely lose track of the means with which users may connect to their systems [65].

Consider configuring an application that runs on top of a web server such as Apache. An administrator must first choose an Apache authentication provider; we chose a simple one, mod_authn_file, as a lower bound on the required work. Apache also requires explicit configuration of TLS. The necessary authentication-related modules—mod_ssl, mod_auth_digest, mod_authz_host, and mod_authn_file—have 54 configuration points; one mistake can result in missing or weak encryption or authentication. As this work is performed at the application layer, the system administrator must independently configure each application (e.g., Postfix, MySQL, etc.).

Configuring Ethos For the most part, Ethos administrators can safely treat applications as black boxes—application-specific security settings are unnecessary. Furthermore, Ethos partitions general security requirements from business logic. The former is the responsibility of administrators; only the latter must be considered by application programmers. Of course, good code partitioning by application programmers will enable the system administrator to be more effective. That is, programmers should split their applications into separate processes so that they can be subjected to the Multics principle of least privilege [18].

No configuration is needed for protections P1–P6 (§5.2); they are transparently provided by Ethos. The security settings Ethos does have are centralized in the LAS/RAS and system-level authorization configuration (both are policy specifications, and so must be configurable). Authorization is external to application code, and covers network authorization by user, host, and executable. Anonymity in Ethos allows universally accessible, but authenticated, service without requiring application-level user accounts.

Ethos reduces code audit requirements. Network authentication is expected to be the common case, and this does not require any application source code audits for integrity and confidentiality. Few services require local or arbitrary-user virtual process authentication; these applications are easy to audit, especially as distributors are quite small. Distributors might also be audited for Denial of Service (DoS) protections, but this is only necessary for stranger- or anonymous-authorized services. (More complex DoS are associated with Ethos' cryptography, but that is not performed by the application.)

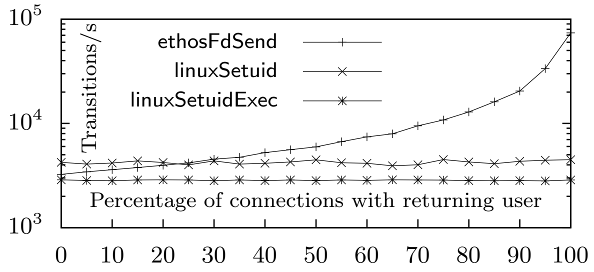

Figure 5: User transitions: fdSend vs. setuid

5.7 Ethos code base

To ensure authentication, Ethos must be properly implemented. Towards this goal, the Ethos environment is engineered to remain small, even while consolidating security services. As mentioned previously, targeting VMMs is a major advantage in this effort. We have also carefully chosen a small set of interfaces to export. As a result, Ethos is presently made up of 35,378 LoC, excluding its cryptographic library. Of this, 427 lines implement fdSend, fdReceive, and authenticate. Another 51,733 LoC come from NaCl, the cryptographic library used in Ethos, which has been carefully verified [13].

5.8 Performance

We evaluated Ethos on a computer with a 4.3 GHz AMD FX-4170 quad-core processor and 16GB of memory, which we used to compare the performance of Ethos to Linux. Ethos itself uses only a single core in these experiments. We ran all of our Linux benchmarks with nscd, to minimize the latency of authentication database lookups.

5.8.1 Microbenchmarks

We measured the performance of setuid on Linux with two programs, linuxSetuid and linuxSetuidExec. To these we compared ethosFdSend, an Ethos program. We padded each program to be approximately 800,000 bytes.

linuxSetuid repeatedly forks and in the child process transitions with setuid to a user known at compile time (but resolved at run time to a UID using getpwnam) before exiting. linuxSetuidExec extends linuxSetuid by calling exec after setuid. This better isolates the post-authentication logic and allows controls—such as SELinux—to restrict the server separately from the distributor. This trades performance for robustness.

ethosFdSend is a program that measures the performance of fdSend, including both sending a file descriptor and creating the virtual process. ethosFdSend loops for a set period of time, calling fdSend with some user on each loop iteration. Each invoked virtual process runs a loop that continuously tries to fdReceive file descriptors.

Figure 5 displays the results of comparing linuxSetuid, linuxSetuidExec, and ethosFdSend. This figure plots the performance in user transitions per second across a range of returning user percentages. From left to right, the figure depicts performance from the case of 2,000 iterations with no returning users (0%) to the case of a single user serviced

Table 3: Local eMsg delivery

	1KB msg/s
eMsg single recipient	1,435
eMsg 50% new recipient	824
eMsg 100% new recipient	454
Postfix	250

2,000 times (100%). When ethosFdSend encounters a new user, Ethos creates a new virtual process; loading this program is the primary source of latency. On the other hand, if a user returns, then the user's virtual process already exists, lowering latency.

In the worst case, where each user encountered is new, ethosFdSend was able to create 3,243 virtual processes per second. When all users are repeats (i.e., high user locality), its throughput was 73,803 user transitions per second. ethosFdSend always outperformed linuxSetuidExec, this performance advantage ranged from 113% to 2,573%. At approximately 25% returning users, ethosFdSend outperformed linuxSetuid. The relative performance of ethosFdSend to linuxSetuid ranged from 77% to 1,645%.

These experiments resulted in the execution of a virtual process per user, and this consumed both memory (due to process duplication) and CPU (due to scheduling overhead) resources proportional to the number of users. In Ethos, it is possible to perform network authentication while running the serving process as a single pseudo-user. For example a ping service might service ping requests with a single set of privileges, but only from authorized users. This will reduce resource use but might also result in lower security.

5.8.2 Application benchmark

To benchmark an entire application, we replaced msgWrite with msgBenchmark, a program that sends 2,500 messages sequentially. We ran msgBenchmark so that it sent mail locally instead of over the network. Thus msgBenchmark tests eMsg's authentication rate, along with the speed at which Ethos can read/write message spools. We compared eMsg to Postfix, listening on a UNIX domain socket, and driven by smtp-source.

Our results appear in Table 3. Like the microbenchmarks, this benchmark is sensitive to the number of running virtual processes. Thus we ran msgBenchmark so that it sent messages under three different scenarios: to a single user, a new user 50% of the time, and a new user 100% of the time. In the first case, the cost of executing msgReceive is amortized across the number of iterations. At the other extreme, Ethos must execute a new virtual process for each iteration. In each case, eMsg outperforms Postfix.

6 Compatibility

Because of the huge computing infrastructure and installed software base, compatibility is essential. Compatibility impacts several levels in a system, including: applications, libraries, system calls, networking, and protocols. Ethos' clean-slate design is intended to avoid semantics that have lead to errors in the past, and thus Ethos is not source code compatible. Another means is therefore needed for real-world compatibility.

Ethos' VMM-based implementation bridges the issues of needed applications, libraries, and system calls. A computer can run several OSs, which means that the other OSs' applications are instantly available, as are its system calls and libraries (albeit with different security properties). Furthermore, Ethos is directly compatible with IP and traverses Network Address Translation (NAT) networks.

By design, Ethos speaks only a single family of RPC-based application protocols. Other protocols can be bridged through the use of proxies running on other OSs. Thus we have ported Ethos's networking stack to Linux. This allows Ethos to service requests originating as any number of existing protocols. Of course, this is a compromise; existing OSs will benefit only partially from Ethos' design. But it has the advantage that Ethos' code is small and unified, leading to a more analyzable and secure core.

Collectively, these techniques allow a transition to Ethos (or another designed-for-security OS) over a period of decades, if necessary, since using Ethos does not preclude the use of any other OS. We believe that during that period, many applications will need to be rewritten to be more secure, and we hope that they will target Ethos to do so. (We note that massive rewrites and application changes were made to deal with the Y2K issue; security is a much broader issue and so its solution is likely to be more expensive). This is a large amount of work, but we believe it to be far smaller than trying to secure critical applications on traditional OSs.

7 Related work

POSIX POSIX authentication is predominantly discretionary. The setuid system call family changes a process's user credentials, but its rules are complex, inconsistent, and ill defined [15]. Attacks can inflict substantial damage, as setuid often requires the process to run as root before transitioning to the target user.

Pluggable Authentication Modules (PAM) provides configurable authentication methods (e.g., password, smart card, etc.) using various back-end databases [57]. Because PAM modules typically execute in the application's address space, a compromised application may leak user secrets or bypass authentication altogether.

Networked services often use frameworks such as GSS-API [42] to perform an authentication handshake, but security failures can stem from incorrectly invoking or bypassing library-based authentication.

Many network services run as a pseudo user, requiring the application to implement authorization logic separate from any system-wide policy. Worse, some monolithic network services run with superuser privileges, where security holes can provide an attacker unlimited access to the machine [12]. Countermeasures for attacks on over-privileged processes burden system administrators, requiring application code audits [52, 38].

Operating systems UNIX provides a pure form of privilege escalation through the setuid bit in its filesystem. An attacker can craft an environment and then invoke a highly-privileged setuid-bit process. Ethos avoids authentication privilege escalation because its processes never change owners (see §4.1).

Plan 9 improves on POSIX/UNIX authentication. It provides a system service called factotum which protects users' authentication keys/passwords and implements a suite of authentication protocols [20, 51]. In addition, services may invoke kernel-based network encryption. Thus applications are isolated from authentication secrets and protocols. Instead of a highly privileged setuid system call, Plan 9 provides similar functionality through two device interfaces, /dev/caphash and /dev/capuse, and a protocol by which factotum provides a program the capability to transition to a new user.

We were inspired by Plan 9's careful isolation, but we aimed to make Ethos' protections inescapable. In doing so, we subsume the work of configuring/auditing applications to ensure they use factotum, strong authentication protocols, and encryption. This strategy resembles distributed firewalls [32], which also make certain protections inescapable by adding to the semantics of their network system calls. Plan 9's secstore inspired Ethos' keystore (§4). Ethos' LAS/RAS also resembles Active Directory, which permits local password authentication, but can associate certificates with users for network authentication [50].

Many OSs are capability based [58]. For example, Capsicum adds a capability system to FreeBSD [66]. However, applications continue to rely on traditional means for authentication. Thus Ethos' authentication mechanism and Capsicum (separate from FreeBSD) are complementary; Ethos provides system-level authentication guarantees and more abstract system calls; Capsicum promotes least privilege by easing application decomposition and sandboxing.

Ethos' distributors drew from many existing systems including inetd, OKWS' okd, and qmail [39, 12]. OKWS on Unix provides only per-service isolation due to performance concerns. Efstathopoulos et al. extended OKWS to run on the Asbestos OS and provide per-user isolation, and qmail demonstrates that per-user isolation can perform well on Unix. Ethos uses virtual processes to perform per-user isolation; we showed in §5 that fdSend performance benefits from locality of reference.

HiStar to provides strict information flow control [71], and DStar extends information flow controls across the network [72]. HiStar and Ethos are complementary: HiStar's Unix layer could be replaced with Ethos' higher abstractions, and Ethos could adopt many of HiStar's contributions to flow control.

Taos implements distributed security as part of the OS [68], and makes no distinction between local and remote principals. A Taos process may make a request on behalf of a set of allowed principals; the recipient can identify the requester using GetPrin and confirm permission using Check. Like Taos, Ethos uses global identifiers—public keys—to identify users. Moreover, Ethos shares system-level authentication with Taos. Ethos' ipc bears some resemblance to GetPrin, but Ethos performs mandatory authorization, so Check is not necessary.

Singularity simplifies Taos' authentication model somewhat [69]; its processes have a single, immutable principal associated with them. Thus it is likely that applications will need to spawn processes for different purposes, but Singularity's Software Isolated Processs (SIPs) lower this cost. Ethos' virtual processes provide a similar benefit yet do not require an application virtual machine (i.e., Common Language Runtime (CLR)).

Like Singularity, an Ethos process speaks for only one principal, but for simplicity we do not explicitly support compound principals. Rather, we rely on public key uniqueness and virtual processes. Thus an Ethos public key always corresponds to user u at host h. This requires Ethos to support equivalence classes of users.

Ethos authorizes a chain of requests involving several processes along the way, whereas Taos and Singularity consider a compound principal at the endpoint. We believe that this is only useful in (tightly coupled) server-to-server configurations. In contrast, Ethos relies on authorization and client-to-end-server certificates for such protections. The sophistication of Taos and Singularity is more expressive, while Ethos pursues simplicity.

8 Conclusion and further work

Typically, it takes less than a handful of system calls to implement authentication. At first appearance, it may seem that authentication is an isolated mechanism and that the highly privileged processes required remain simple enough to rarely contain vulnerabilities. But such an approach results in mechanisms which are subject to many abuses. As authentication is a powerful operation—often the sole basis for determining permissions—these abuses poses great dangers to the system's security. For this reason, authentication has been subject to many attacks.

In reality, authentication is quite subtle, and analyzing an application for flaws has proven extraordinarily difficult. Thus Ethos authentication takes place at the system level. For high integrity, it integrates network authentication and local authentication mechanisms—ensuring strong authentication, even across networks. PKC enables remote principals to be identified by their public key, thus ensuring a globally unique identifier for every user—even strangers or anonymous users. Every principal is identified to Ethos, enabling Ethos to (differentially) authorize each user. Additionally, Ethos authorization prevents misuse of the most common authentication case purely at the system level. This subsumes the need to audit any application authentication code.

Ethos eliminates thousands of lines of authentication code in each networked application. The benefits include: (1) security critical code is moved to the system so it is unaffected by application code; (2) the system guarantees strong techniques rather than relying on application programmers; (3) all remote users—even anonymous ones—are authenticated and authorized; and (4) application-specific security configuration is reduced. The result is fewer opportunities to make security errors.

In addition, Ethos' authentication system is carefully designed to require user secrets to be unlocked by local authentication, isolate authentication secrets within the OS, eliminate authentication privilege escalation, and provide support for anonymous users.

Future work on Ethos includes expanding its user-space libraries, developing user interfaces, and developing applications. These are substantial tasks due to Ethos' clean-slate design. We will focus on the most security-sensitive applications, where the advantages of Ethos are most compelling. In the long run we also plan to produce a high-assurance implementation of Ethos.

9 References

[1] Fedora crypto consolidation. http://fedoraproject.org/wiki/FedoraCryptoConsolidation.

[2] National vulnerability database. http://nvd.nist.gov/.

[3] Phishing activities trends report: 1st quarter 2012. anti-phishing.org/reports/apwg_trends_report_q1_2012.pdf,

July 2012.

[4] Accetta, M. et al. Mach: A new kernel foundation for UNIX development. *Proceedings Summer USENIX* (1986).

[5] Anderson, N. Vint Cerf: One quarter of all computers part of a botnet. `http://arstechnica.com/old/content/2007/01/8707.ars`, 2007.

[6] Arbaugh, W. A. et al. Reliable bootstrap architecture. In *Proc. IEEE Symp. Security and Privacy* (1997), pp. 65–71.

[7] ASD(C3I). DoD directive 8500.1 information assurance (IA). Tech. rep., ASD, Oct. 2002.

[8] Bacher, P. et al. Know your enemy: Tracking botnets, 2005.

[9] Barham, P. et al. Xen and the art of virtualization. In *SOSP* (Bolton Landing, NY, USA, Oct. 2003), ACM, pp. 164–177.

[10] Berger, S. et al. vTPM: Virtualizing the trusted platform module. In *USENIX Security* (2006), pp. 305–320.

[11] Bernstein, D. J. Cache-timing attacks on AES, 2004. `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`.

[12] Bernstein, D. J. Some thoughts on security after ten years of qmail 1.0. In *Proceedings of the 2007 ACM workshop on Computer security architecture* (New York, NY, USA, 2007), CSAW '07, ACM, pp. 1–10.

[13] Bernstein, D. J. Cryptography in NaCl. `http://cr.yp.to/highspeed/naclcrypto-20090310.pdf`, 2009.

[14] Bonneau, J. et al. The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *Proc. IEEE Symp. Security and Privacy* (2012), pp. 553–567.

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

[16] Chou, A. et al. An empirical study of operating system errors. In *SOSP* (2001), pp. 73–88.

[17] Colp, P. et al. Breaking up is hard to do: Security and functionality in a commodity hypervisor. In *SOSP* (Cascais, Portugal, Oct. 2011), ACM.

[18] Corbato, F. J. et al. Multics—the first seven years. In *Spring Joint Computer Conference* (1972), AFIPS Press, pp. 571–583.

[19] Corrigan-Gibbs, H. and Ford, B. Dissent: accountable anonymous group messaging. In *CCS* (2010), pp. 340–350.

[20] Cox, R. et al. Security in Plan 9. In *USENIX Security* (2002), pp. 3–16.

[21] Dahlin, M. et al. Toward the verification of a simple hypervisor. In *10th International Workshop on the ACL2 Theorem Prover and its Applications* (Nov. 2011).

[22] Department of Defense. Trusted computer system evaluation criteria. Tech. Rep. DOD 5200.28–STD, U. S. Department of Defense, 1985.

[23] Dingledine, R. et al. Tor: The second-generation onion router. In *USENIX Security* (2004), pp. 303–320.

[24] Engler, D. R. et al. Exokernel: An operating system architecture for application-level resource management. In *SOSP* (Dec. 1995), ACM SIGOPS, pp. 251–266.

[25] George, D. Evolution of information assurnace. In *USENIX Security* (2012). Keynote address.

[26] Gobioff, H. et al. Smart cards in hostile environments. In *Proceedings of the 2nd USENIX Workshop on Electronic Commerce (EC-96)* (Berkeley, Nov. 18–21 1996), USENIX Association, pp. 23–28.

[27] Goodin, D. Hackers break SSL encryption used by millions of sites. `http://www.theregister.co.uk/2011/09/19/beast_exploits_paypal_ssl/`, Sept. 2011.

[28] Gu, G. et al. BotSniffer: Detecting botnet command and control channels in network traffic. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)* (February 2008).

[29] Halderman, J. A. et al. Lest we remember: Cold boot attacks on encryption keys. In *Usenix Security* (2008).

[30] Heiser, G. and Leslie, B. The OKL4 microvisor: convergence point of microkernels and hypervisors. In *Proceedings of the first ACM Asia-Pacific workshop on systems* (New York, NY, USA, 2010), APSys '10, ACM, pp. 19–24.

[31] Hiltgen, A. et al. Secure Internet banking authentication. *Security Privacy, IEEE 4*, 2 (March-April 2006), 21 –29.

[32] Ioannidis, S. et al. Implementing a distributed firewall. In *CCS* (2000), ACM Press, pp. 190–199.

[33] ISO/IEC Standard 15408. *Common Criteria for Information Technology Security Evaluation*, version 3.1 ed., July 2009. `http://www.commoncriteriaportal.org/cc/`.

[34] 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* (Aug. 2006).

[35] John, J. P. et al. Studying spamming botnets using Botlab. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation* (Berkeley, CA, USA, 2009), NSDI'09, USENIX Association, pp. 291–306.

[36] Keromytis, A. D. et al. The STRONGMAN architecture. In *DISCEX* (2003), vol. 1, pp. 178–188.

[37] Klein, G. et al. seL4: formal verification of an OS kernel. In *SOSP* (New York, NY, USA, 2009), ACM, pp. 207–220.

[38] Krohn, M. et al. Make least privilege a right (not a privilege). In *HotOS* (Santa Fe, NM, June 2005).

[39] Krohn, M. N. Building secure high-performance web services with OKWS. In *USENIX ATC* (2004), pp. 185–198.

[40] Lampson, B. et al. Authentication in distributed systems: Theory and practice. *TOCS 10*, 4 (1992), 265–310.

[41] Le Malécot, E. et al. Preliminary insight into distributed SSH brute force attacks. *Proceedings of the IEICE General Conference* (2008).

[42] Linn, J. Generic interface to security services. *Computer Communications 17*, 7 (July 1994), 476–482.

[43] Loscocco, P. and Smalley, S. Meeting critical security objectives with security-enhanced Linux. In *Proceedings of the Ottawa Linux Symposium* (Berkeley, CA, 2001), The USENIX Association.

[44] Lowe, G. An attack on the Needham-Schroeder public-key authentication protocol. *Information Processing Letters 56*, 3 (1995), 131–133.

[45] Martin, B. et al. 2010 CWE/SANS top 25 most dangerous software errors. Tech. rep.

[46] Murray, D. G. et al. Improving Xen security through disaggregation. In *VEE* (New York, NY, USA, 2008), ACM, pp. 151–160.

[47] Oliveira, F. et al. Barricade: defending systems against operator mistakes. In *EuroSys* (New York, NY, USA, 2010), ACM, pp. 83–96.

[48] Petullo, W. M. et al. Minimalt: Minimal-latency networking through better security. In submission.

[49] Pike, R. System software research is irrelevant, Aug. 2000.

[50] Pittaway, G. Report highlights: Distributed security services in microsoft windows nt 5.0 - kerberos and the active directory. *Inf. Secur. Tech. Rep. 4* (Jan. 1999), 20–21.

[51] Presotto, D. and Winterbottom, P. The organization of networks in Plan 9. In *USENIX Association. Proceedings of the Winter 1993 USENIX Conference* (1993), USENIX, pp. 271–280.

[52] Provos, N. et al. Preventing privilege escalation. In *USENIX Security* (Aug. 2003), USENIX, pp. 231–242.

[53] Rajab, M. A. et al. A multifaceted approach to understanding the botnet phenomenon. In *IMC '06: Proceedings of the 6th ACM SIGCOMM on Internet measurement* (2006), ACM Press, pp. 41–52.

[54] Reiter, M. K. and Stubblebine, S. G. Authentication metric analysis and design. *TISSEC 2*, 2 (1999), 138–158.

[55] Rivest, R. L. and Lampson, B. SDSI — a simple distributed security infrasture. Tech. rep., MIT, Apr. 1996.

[56] Sailer, R. et al. Building a MAC-based security architecture for the Xen open-source hypervisor. In *ACSAC* (2005), IEEE Computer Society, pp. 276–285.

[57] Samar, V. Unified login with Pluggable Authentication Modules (PAM). In *CCS* (1996), C. Neuman, Ed., ACM Press, pp. 1–10.

[58] Shapiro, J. S. et al. EROS: a fast capability system. In *SOSP* (1999), pp. 170–185.

[59] Shinagawa, T. and Kono, K. Implementing a secure setuid program. In *PDCN* (2004).

[60] Sinclair, S. and Smith, S. W. PorKI: Making user PKI safe on machines of heterogeneous trustworthiness. In *ACSAC* (2005), IEEE Computer Society, pp. 419–430.

[61] Solworth, J. A. and Fei, W. High performance global authentication. In submission.

[62] Song, D. X. et al. Timing analysis of keystrokes and timing attacks on SSH. In *USENIX Security* (Berkeley, CA, USA, 2001), USENIX Association, pp. 25–25.

[63] Steinberg, U. and Kauer, B. Nova: a microhypervisor-based secure virtualization architecture. In *EuroSys* (New York, NY, USA, 2010), ACM, pp. 209–222.

[64] Thompson, K. Reflections on trusting trust. *CACM 27*, 8 (1984), 761–763.

[65] Waterman, S. Glitch imperils swath of encrypted records. *The*

Washington Times (December 2012).

[66] Watson, R. et al. Capsicum: practical capabilities in UNIX. In *USENIX Security* (Aug. 2010), USENIX.

[67] Williams, D. et al. Device driver safety through a reference validation mechanism. In *OSDI* (Berkeley, CA, USA, 2008), OSDI'08, USENIX Association, pp. 241–254.

[68] Wobber, E. et al. Authentication in the Taos operating system. In *SOSP* (1993), pp. 256–269.

[69] Wobber, T. et al. Authorizing applications in Singularity. In *EuroSys* (New York, NY, USA, 2007), ACM, pp. 355–368.

[70] Yin, Z. et al. An empirical study on configuration errors in commercial and open source systems. In *SOSP* (New York, NY, USA, 2011), ACM, pp. 159–172.

[71] Zeldovich, N. et al. Making information flow explicit in HiStar. In *OSDI* (Seattle, Washington, Nov. 2006).

[72] Zeldovich, N. et al. Securing distributed systems with information flow control. In *NSDI* (Berkeley, CA, USA, 2008), NSDI'08, USENIX Association, pp. 293–308.

[73] Zhou, Z. et al. Building verifiable trusted path on commodity x86 computers. In *Proc. IEEE Symp. Security and Privacy* (May 2012).

[74] Zorz, Z. NSA considers its networks compromised. `http://www.net-security.org/secworld.php?id=10333`, Dec. 2010.