Security Requirements for the Deployment of the Linux Kernel in Enterprise Systems

Trent Jaeger, David Safford, Hubertus Franke

IBM T.J. Watson Research Center, Route 134, Yorktown Heights, NY 10598

Email: {jaegert,safford,frankeh}@watson.ibm.com

Introduction

The goal of this white paper is to establish an architecture for providing security services for Linux-based enterprise-level Internet servers. While Linux provides good security features, equivalent to other comparable operating systems, the emerging security threats require stronger security mechanisms in the Linux kernel. We present a design that will make Linux substantially more secure against modern threats while maintaining a high degree of compatibility for existing enterprise software.

We organize this paper as follows. Initially, we state the goals that are associated with a secure system. We outline the common security threats that enterprise systems face on the Internet and argue that existing security mechanisms are inadequate to meet this challenge. We then derive functional requirements that achieve the desired security goals in the modern threat environment. We then outline a design that meets these requirements. As much as possible, we want to incorporate existing open source projects that already meet one or more of the necessary requirements.

Generic Security Goals

Some of the basic goals associated with security services are:

- **Integrity:** The ability to assert that the system and its data are authentic and have not been tampered with, intentionally (intruder) or unintentionally (software bug).
- **Secrecy:** The ability to restrict access to data and applications based on an identity.
- **Auditability:** Ability to log all or a subset of security related events and make them available for review.
- **Robustness:** The ability of the system to maintain service without interruption in the presence of faults and attacks.
- **Manageability:** Security services should be easy to configure and understand.

The traditional paradigm for meeting these goals is a combination of authentication and authorization. The first step is to identify the party making a request. The second step is to determine whether that party is authorized to make that request. There are many specific

implementations of these mechanisms with varying degrees of flexibility, performance, and strength.

Linux-Specific Goals

Linux implements a traditional reference monitor authorization mechanism that provides controlled access to system resources like files and communication devices based on password authentication of users. A process desiring access to these resources must go through the monitor via a system trap. Linux implements the monitor as a monolithic kernel with hardware trapping of all communication to and from the kernel.

We will show that Linux security is comparable to that of other enterprise operating systems, but such security is still quite vulnerable to current enterprise security threats. Thus, we will develop a target for Linux and show the steps that can be taken to significantly improve its security along all goal dimensions.

In doing so, we want to take a practical approach realizing that Linux is in widespread use and a radical redesign to accommodate best of bread security features is unrealistic and also might be unnecessary. In fact, it is desirable for the resultant system to look and act very much like a traditional Linux system to the general user. In addition, any changes must be reasonable efficient as most users will not accept significant reduction in performance.

The emphasis is on protecting against common and well-known attack methods, such as exploiting software bugs (buffer overflow), exploiting cgi-bin scripts, and exploiting of typical misconfiguration and weak authentication in traditional Linux system as described below.

Common Security Threats

While Linux is comparable to other server operating systems in its security features, it can be made more resistant to current hacking threats. To understand how additional features can help, one needs to understand the nature of the current threats. Server systems normally become vulnerable in one of two ways: through buggy software or through misconfiguration.

By far, the most common attack on server machines is to exploit known bugs or vulnerabilities in the software providing one of the services. Network services inherently accept request data over the network, even from potentially malicious sources. If the software is not careful with this malicious data, the data may be able to exploit the lack of care to “take over” the server process, making it run code of the hacker’s choosing.

The two most common categories of such “data driven” attacks are buffer overflows and parsing errors. In a buffer overflow attack, the hacker sends data longer than the server expects, causing the server software to copy this data over its own code. The long data contains executable code, which when run, gives the hacker access to the server. While it may seem simple to write server code that rejects this malformed input data, buffer overflows have been known for over 28 years

and continue very frequently to this day. The recent "CodeRed" worm, which compromised over five hundred thousand Microsoft web servers, attacked a single buffer overflow in the IIS server.

In parsing errors, the hacker again sends malicious data, but rather than long data, he sends data with embedded control characters that cause the server to do a sensitive operation that it normally would not allow. Here again, while it would seem simple to reject such malicious data, it can be very hard. IIS has had six variations of vulnerabilities related simply to rejecting pathnames with embedded "../" characters.

Vulnerabilities due to such buggy software are being discovered at the rate of roughly 2500 per year, or 9 per day. Studies have estimated that there will exist one such vulnerability per thousand lines of code. For a typical Linux distribution, with 30 million lines of code, this means that there are likely some 30,000 such vulnerabilities. Given the history of buffer overflows and parsing errors, it seems very unlikely that software developers will ever magically start producing perfect code, so operating systems must provide some method of tolerating or containing the results of buggy software.

Misconfiguration is the second major way in which servers can become vulnerable to attack. In misconfiguration, a service is not buggy, but is accidentally configured to give out services that it shouldn't. For example, a web server normally will not allow a client to access sensitive data like the password file, but the server may accidentally be misconfigured to allow this access. While it may seem simple to configure a service correctly, in practice this can be quite difficult.

The first challenge is that there is no standard way to express a desired security policy across all services on a server. Each service tends to have its own arcane configuration methods. On a typical Linux server, there are dozens of security critical configuration files spread throughout the filesystem, which all must be set correctly. If any one of them has an error, the entire system can be vulnerable.

Even if an administrator manages to configure a machine securely, it is very easy to cause subsequent inadvertent misconfiguration. Every installation of even minor new applications or application updates can change or overwrite configuration files. Major applications have been known to install security nightmares, such as setuid root executable, and world writable executables. This configuration problem is made almost impossible by the need to install all too frequent security critical patches.

Traditional Solutions

The NSA Orange Book is the traditional source for the definition of security requirements and their evaluation criteria. Although the Orange Book is mainly used in government systems, it is a good starting point to state and analyze enterprise requirements.

The Orange Book specifies the following increasingly more secure levels and their minimum functional requirements.

- C2: Authentication, Discretionary Access Control (DAC)
- B1: Mandatory Access Control (MAC), Audit
- B2: Structured Security, Elimination of Storage Covert Channels
- B3: Minimized Trusted Computing Base (TCB), Elimination of Timing Covert Channels
- A1: Proven Security (this is not a functional requirement)

The first level C2 identifies the need for password authentication and discretionary access control. Authentication enables identification of the users making system requests. Discretionary access control means that the users define control over their objects that their discretion. Thus, the system can use the authenticated identity to make access control decisions to protect the secrecy (i.e., ability to read) and integrity (i.e., ability to write) of objects as defined by the users.

Most commercial, server-level operating systems, including AIX, Windows NT, and Solaris, have been certified to this C2 level. Linux also contains basic C2 security features, such as password authentication, file system discretionary access control, and security auditing. While Linux has a few minor omissions in its auditing and password management features, it is largely C2 compliant, and could be certified at this level with only minor tweaks.

At the B1 level, mandatory access control and auditing features are required. Mandatory access control means that the system administrators define the access control policy of the system, not the users. The intention is to prevent users from granting unauthorized rights that would compromise the secrecy or integrity of their objects accidentally. An audit system is intended to capture the relevant security-critical operations to prevent compromise and/or identify sources and means of attack (i.e., achieve the auditability goal). The ease in which access control and audit can be used to protect a system defines its ability to meet the manageability goal.

A minimized Trusted Computing Base (TCB) means that the amount of security-relevant code in a system is minimized. This benefits the system by reducing the number of ways in which a system can be attacked (e.g., by reducing the number of bugs). Since the TCB is a fundamental component of the system, its minimization achieves the robustness goal of the system as well as better ensuring secrecy and integrity.

Covert channels and structured security (i.e., protection domain boundaries within the kernel) are not practical requirements for Linux at this time, as they would require a significant redesign of the kernel.

Details of means for achieving security goals (e.g., authentication, access control) are discussed in the section on the Linux Security Architecture.

Inadequacies of Current Linux Mechanisms

While some people claim that the existing security features in Linux and other available operating systems are adequate to meet the demands of e-commerce security, there is clearly room for improvement. One popular argument is that servers are vulnerable only because lazy administrators fail to keep up to date with the latest security patches. The problem with this argument is that patches come out too frequently - seven per day, across all systems. Thus, an administrator is often faced with the choice of applying patches without adequate testing, and possibly breaking a critical application, or not applying the patch, and being hacked.

Further, the installation of new applications or patches often enables the current security policies to be modified. Historically, system services run as <root> and use their own access control files to control use. The installation of programs that may silently change the entries in these files presents a difficult access control problem. Once stronger access controls are in place, there is the possibility that effective updates are harder: they either break access control rules or subvert them.

A second common security misconception is to place too much trust in security tools, such as firewalls and intrusion detection systems (IDS). A firewall does protect against some attacks, but many other attacks, such as e-mail and web attacks, go right through. While an IDS can detect many intrusion attempts, it often notices the attack only after it has succeeded. This is little comfort after customer credit card numbers have been stolen.

Administrators need additional tools, beyond the standard operating systems, to manage the security of their systems. These tools must provide the administrators with easily managed security configurations, provide a coherent approach to software update, contain the effects of buggy software, and enable effective use of auditing and intrusion detection.

Summary

While Linux security is comparable to that of current enterprise operating systems, it clearly does not effectively protect such systems against current attacks, such as buggy system services and misconfiguration. A variety of security approaches are known that can improve the security of a system, but it is not clear how these approaches can be combined into a effective security system and how these approaches can be integrated into an existing Linux system in ways that will be acceptable to the Linux community.

A problem typically is that achieving the ultimate security goal in one step is too much of a change for the community to accept. Therefore, it is important to understand the current security tools available for Linux and what they can do now, where we ultimately want to go, and what steps can be taken to get there. Making incremental improvements of Linux security that lead to a sufficiently secure solution is our goal.

Applicability of Security Concepts for Enterprise Linux

In the following section, we review the basic security concepts of the Orange Book levels and discuss in detail whether and how they are applicable for secure enterprise Linux.

Minimized Trusted Computing Base (TCB)

The Orange Book defines the TCB as the collection of security relevant software on a system. The single best way to improve the security of a system is to minimize the amount of code that must be trusted to enforce a secure system. A recent study [19] has shown that the rate of significant security bugs is directly related to the number of lines of code in a system, with the typical rate being one security bug per 1000 lines of code. RedHat 6.2 was measured at 17MLines of code with roughly 1.5MLines in the kernel. Minimizing the TCB in Linux to at most encompass the kernel, would reduce 90% of the potential security bugs.

At the current time, Linux has an unbounded TCB forcing all 17MLines of code to be considered part of the TCB. One of the major reasons for this is that Linux follows the POSIX tradition in which applications are responsible for authentication and the kernel trusts the applications to do this properly and inform the kernel of the results via the setuid() system call. Consider that login() is application that runs at the same security level (root) as any average Linux system service. Therefore, any compromised Linux system service can replace login() with a rogue version.

Authentication

Authentication is the means by which the security-relevant identity of a user or client is established. Authentication requires that the users prove their identities by demonstrating knowledge of a secret (e.g., password or key). The system uses this authenticated identity to control the assignment of security types, and hence access rights, to the users processes. Further, authentication protocols are used to provide integrity and secrecy protection for network communications (e.g., SSL and IPSEC) and files (e.g., encrypted filesystems). While authentication is a fundamental TCB security function, there are several problems with its current implementation. Solving these problems depends on providing an approach we call *mandatory authentication*, where the authentication is required for all security changes and protected as part of the TCB. Ultimately, achieving our goals will require a staged approach as discussed in the Linux Security Architecture section.

Linux, like a typical UNIX system, depends on system services running outside the kernel (e.g., login()) to perform authentication via passwords and provides series of system calls (e.g., setuid()) that enable applications to set the security identity. There are several problems with this approach. First, Linux authentication services are run at the same security level as all Linux system services, many of which are not part of the TCB. As we have shown, bugs in Linux

services can lead to their compromise and as a result the compromise of the TCB. In general, any application can call setuid() to affect authentication decisions, so the TCB in Linux is actually unbounded. Second, passwords are a notoriously flawed way to provide strong authentication. Passwords do not scale in a distributed system, because a lack of trust between domains requires different passwords. Thus, users choose lots of easy-to-remember passwords that are often easily broken via dictionary attacks. Lastly, there is a significant amount of authentication being done by processes that will never be part of the TCB. For example, applications use SSL (now TLS) instead depending on IPSEC which runs in the kernel.

First, we need mandatory authentication, where the TCB performs all authentication operations upon which security decisions are based and all TCB authentication services are protected from modification. A small number of authentication services must be defined and protected. There are two approaches to development of these services: (1) at user-level, but protected by MAC policies and (2) in the kernel. Both approaches can be used to build secure systems, although the first approach depends on a strong MAC policy guarantees. The main advantage of the latter approach is that the integration of all authentication services may be easier in the kernel, but some additional system calls are necessary to implement it. Therefore, we recommend examining building secure systems with both approaches and helping the Linux community decide which is better.

To help address the password problem, the Pluggable Authentication Modules (PAM) framework was created. PAM modules are actually libraries that support an interface for authentication that PAMified services can use. The service calls pam_authenticate() with the identity of the client to be authenticated and the library uses whatever means that it desires to determine whether the authentication succeeds or fails. Thus, PAMified authentication services can use custom authentication techniques to verify the identity of users. Also, PAM is used by Argus PitBull and SELinux to pass their security information to the kernel on login. However, both still use passwords -- they use PAM to authenticate their security IDs. Also, such services still run at user-level, so a compromised Linux service can still replace the PAM authentication services and thus compromise the TCB.

As far as the choice of strong authentication technique, strong authentication requires the use of cryptographically strong keys. There are two types of commonly-used cryptosystems, symmetric (secret) key and asymmetric (public) key. Symmetric keys cryptosystems require the use of a common password, which makes it unsuitable in distributed systems due to the management overhead and the exposure to dictionary attacks. Asymmetric cryptosystems have one key that is public which is distributed widely (i.e., the public key) and one that is always kept secret (i.e., the private key). The private key need not be managed by each domain, so the same key can be used in multiple domains which provides better scalability. We have seen some advancement in the use of public key cryptosystems in authentication (e.g., SSH), but the infrastructure is not there for a typical enterprise user. Also, people have gotten used to building cryptosystems at user-level to get around the limitations of the operating system, such as the Kerberos symmetric cryptosystem and the user-level public key protocol SSL (now TLS). We think that it is a matter of building an acceptable kernel authentication service that can support public key cryptosystems

for authentication, however, such a system must enable the implementation of existing techniques to gain acceptance.

Lastly, as much as possible, the authentication information computed by the system should be done in the TCB, so the TCB can leverage this information in security decisions. If applications use SSL rather than IPSEC, then the TCB still has no choice but to believe that the network data is untrusted. Thus, enforcing least privilege is not possible. The rights granted to the application must be the same regardless of the clients to which it communicates (unless upgraders are used, but these introduce other problems). Also, even the myriad of the authentication services in the TCB (IPSEC, signed executables, and user authentication) require integration before the authentication information can be used effectively.

Access Control

Access control or authorization determines whether a particular subject (e.g., user, process, etc.) can perform a particular operation (e.g., read, write, etc.) on a particular object (file, socket, etc.). Typically, there is a monitor that determines whether the subject is allowed to perform the requested operation based on an access control policy. While there are two different fundamental ways of expressing access control policy, capabilities and access control lists (ACLs), the monitor determines whether the capability is valid or the ACL permits access. The key difference is that the enforcement of an ACL policy requires that the identity of the subject be known, but the possession of a valid capability is sufficient for authorization for a capability-based policy.

A further distinction is the distinction between discretionary access control (DAC) and mandatory access control (MAC). We describe both in detail below, but the bottom line is that DAC as support historically by UNIX systems, such as Linux, is insufficient for providing protection of enterprise systems. However, work on MAC framework for the Linux kernel are underway and one, called the Linux Security Modules (LSM), enables a variety of MAC policies to be supported behind generic authorization hooks. As LSM has a good likelihood of being accepted into the Linux kernel, it provides a basis for implementation of successful MAC policies, we advocate building access control policy modules for LSM and the integration of authentication services with these LSM modules. Further, we advocate the development of simple policies focused on integrity protection between the different logical services in the system.

Discretionary Access Control

Discretionary Access Control (DAC) means that the management of permissions is at the discretion of the owners of the objects (i.e., the users). The user specifies who can do what on a particular object. It is possible to utilize DAC as a means to implement secure system, however since it is discretionary, it is also easy to do it wrong and to have attacker bypass the DACs. Owner of the object can give away privileges to anybody, thus preventing any containment.

Linux, as is typical of UNIX systems, provides DAC. It represents access control policy in a limited form of ACLs known as mode bits that are simpler and less expressive than ACLs are in general. A further limitation in UNIX access control model is that only access to files can be

controlled using mode bits. However, many other objects, such as sockets, IPC, and global system data are also relevant to security.

Linux also has POSIX capabilities which are not a true form of capabilities, but do enable further control over a variety of system operations. This model basically defines a variety of abilities and the kernel checks whether the process has these abilities set when an operation is run, similar to the mode bit permissions.

Linux typically associates processes with an identity based on the user which is believed to be running the process. We believe that identities based solely on users are not fine enough, because users assume different roles in different applications. Further, we specifically need to eliminate the <root> account, instead making all of its powers available through permissions.

Mandatory Access Control

Mandatory Access Control (MAC) means that the system (i.e., administrators) defines the access control policy for its objects. This is typically done by the assignment of security identifiers to subjects and objects. We will refer to these identifiers as *security types.* Users cannot change a MAC access control policy. Thus, it is practical to use MAC to build secure systems because users cannot give rights away inadvertently. However, it is still non-trivial to design and maintain MAC policies that provide the desired privileges and are *safe*, i.e., do not enable a user to obtain an unauthorized right.

An effective MAC policy protects the integrity of the TCB from non-TCB services. Fundamentally, bugs at a particular security level must be contained to that level. Thus, policy enforcement must ensure that a subject's privileges be strictly decreasing. A process can reduce its privileges at any time, but it can never increase privileges, even to former levels, without an explicit authentication. Thus, if a process is compromised and taken over, it cannot increase its access level.

Linux does not have MAC, but we need MAC policies to build a secure system. Linux variants, such as Argus PitBull [2], Immunix [21], and Security-Enhanced Linux (SELinux) [12], have modifications to the base kernel that enable the enforcement of MAC policies. However, the Linux Security Modules (LSM) project appears to be the approach that will become part of the mainline Linux distributions. LSM provides an interface of generic authorization hooks for the base kernel. This enables loadable modules to be created that can implement a variety of policies, including MAC. For example, the SELinux folks have already ported their policy module to LSM.

There are a variety of access control models for expressing MAC policies. There is a trade-off between the flexibility of the model and the complexity of using the model. In general, MAC policies must be simple or they cannot be maintained effectively by system administrators. Therefore, models that support enough policies as simply as possible are preferred. For example, a model that protects the integrity of processes, such as LOMAC[11], might be an interesting starting point. Perhaps four levels, TCB, system, applications, and untrusted, will be largely sufficient, but chances are that there are a significant number of exceptional cases in current

systems that must be handled gracefully. Example Linux policies in SELinux and Argus PitBull are quite complex, so it may be naive to assume that simple integrity levels will be sufficient.

Exceptional cases must be few and easily addressable to keep them from overwhelming the system administrators. Typically, exceptions are handled either by upgraders, using a more complex policy model to express exceptions, or using audit to track exceptional use. Each of these approaches has significant problems. Often upgraders become complex themselves, and a large number of them becomes unmanageable. Complex policy models make the system administrators ability to maintain configurations correctly difficult. SELinux addresses this problem by using multiple access control specifications: a coarse-grained Role-based Access Control (RBAC) [14] specification and a fine-grained Type Enforcement (TE) [4] policy. Currently, the RBAC policy does not limit the rights distribution in TE. According to the SELinux folks this proved to be to limiting. Lastly, auditing will not prevent actions, only detect them.

Audit

Linux does have a syslog and kernel log facility, but it is not heavily utilized (e.g. the kernel logs mostly activities at startup time). It needs to be able to log **all** security-related decisions. It should be configurable whether only mandatory or discretionary or success and/or faulty.

Linux currently contains some auditing facilities, but they are not comprehensive. Work is underway by SGI to determine the hooks necessary to support an Orange Book B1 level auditing ability for LSM. Such hooks will not be integrated with phase one of the LSM, and a debate is underway as to whether these hooks will be accepted by the Linux community. At a minimum, we expect that an LSM module can use the LSM authorization hooks to perform basic auditing functions. Some work will be necessary to determine how B1 audit can be supported, but SGI is already working with the community to define those hooks. The degree to which additional auditing may be necessary and how hooks for that can be added to Linux need further research.

Structured Security

Structured Security enables the kernel to isolate certain functionality from other parts of the kernel. Normally, this is used to isolate the security monitor and the loadable modules from each other. They are typically implemented through different address spaces and results in a micro kernel approach in the underlying OS design. We specifically do not set this as a requirement as we don't see this as a practical near term goal that can be realized in the Linux community.

Elimination of storage/timing covert channels

This is largely a secrecy issue that we do not think is important enough given the range of deployment scenarios. At this point we want to focus largely on integrity.

Proven security

Proven security requires a comprehensive formal description and evaluation. Given the way that Linux is developed, we do not believe that such an evaluation is possible.

However, some form of evaluation of the security of a Linux system should be performed. For example, we are working on analysis tools that determine whether all dangerous operations are invoked only when the required LSM authorizations have been performed.

Such evaluations should be performed over all facets of the system and the system as a whole. However, cost-effective, comprehensive techniques are not well-developed at present.

Additional Security Features for Enterprise Linux

In the following, we discuss other security features and how they can be enhance enterprise system security. Leveraging these features will take more work than the basic features discussed in the last section, and we briefly examine the work required.

IPSEC (Secure IP)

IPSEC provides authentication, confidentiality, and integrity of network data integrated with the network protocol stack. However, at the current time, IPSEC is not widely used. Instead, applications, such as web browsers, use SSL (Secure Socket Layer). The advantage of SSL is that it does not require any OS support, which as the same time is also its disadvantage from a security perspective. IPSEC, being part of the OS, can enforce network security across all applications independent of application-level support.

Currently, Linux does not provide any IPSEC support, although packages such as FreeSwan provide full IPSEC support. We anticipate that IPSEC will become more widely used in the future.

Signed Executables

A signed executable is effectively combination of a program and a certificate that can be used to verify its integrity. Thus, the TCB can verify that a program is what it claims to be before it is executed (i.e., authenticate programs). This can ensure that programs assigned to run at particular security levels have not been tampered with.

Prototype implementations of signed executable infrastructure for Linux have been done by IBM and Wirex. The signature embedded in the executable (e.g., as an ELF section) is checked by the kernel against a list of trusted public keys before running it. This work can be integrated via an LSM module that is aware of signed executables.

It is important to recognize some of the limitations of signed executables. In particular, this support does not prevent malicious shell scripts attacks or malicious data attacks, such as buffer overflows.

Encrypted Filesystems

Access control is limited to online systems. There is also a need to ensure confidentiality, integrity, and authenticity in case of an offline attack (e.g., disk stealing). Using an encrypted file system solves this problem because the thief cannot read or modify the files without detection without the key.

Due to performance requirements, bulk encryption of files must be done using symmetric keys. Per file keying enables a unique symmetric key for each file, and deleting the associated key

provides a strong means to eliminate the data. This also solves the disk scraping problem. Cryptographic integrity provides protection to offline integrity attacks (cryptographic checksum). Authenticity of the data is related to the metadata of the file, e.g., the ownership of the file. This requires integrity of the metadata.

Although there are encrypted filesystems available that can be dynamically loaded into the kernel, these are prototype systems and none achieves the full set of requirements. StegFS [10] provides an encrypted filesystem where even the presence of files can be hidden. However, access is managed by passwords, so its security is limited by the liabilities of passwords. However, some improved security can be gained using this filesystem, so it is worthwhile to examine the use of this and other similar systems and the development of better authentication mechanisms for it.

Note that there is a synergy between encrypted filesystems and signed executables. If the filesystem guarantees the authenticity and integrity of the executable, the signature on the executable need only be checked at install time, and the filesystem can inherently vouch for the executable everytime it is run.

Per Session /tmp Directory

A common attack is to misuse temporary files created by privileged applications in the /tmp directory. In this attack, the attacker creates a file known to be used by a privileged service and waits for the application to try to create the file as well. If the privileged service fails to check that it created this file as well, then it will use the version created by the attacker. The attacker can point this file to a symbolic link to modify a protected file or modify the data placed by the privileged service in the file.

The success of this attack depends upon a single shared /tmp directory across all applications. A simple method to defeat this attack is to create a unique logical /tmp directory for each authenticated session. In much the same way as `chroot()` syscall redirects references to "/" a newly created syscall `chtmp()` could transparently redirect all references to "/tmp" to the unique temp directory.

Kernel Crypto Support

Kernel Crypto support needs to provide symmetric and asymmetric crypto algorithms along with the associated hashing and MAC (i.e., keyed authenticity/integrity check) function. The Linux kernel has a standard crypto patch that provides basic symmetric cyphers, called the International CryptoAPI [7]. However, this patch does not provide public key operations such as signature verification (e.g., necessary for signed executable support) and public key encryption/decryption for key exchange. Such services must operate transparently regardless of whether there is hardware support for these operations. We expect that this would be achieved by using a standard interface, such as PKCS-11.

PKI Support

In addition to providing public key crypto support, the system needs to be able to find public key certificates needed by the authentication modules, IPSEC, and signed executables. All public key certificates must be obtainable both from local files and optionally from remote services, such as LDAP.

An idea is a configuration switch that lets you specify where to look for public keys (key/certificate server). This would work much the same way that resolv.conf directs hostname resolution on existing Linux systems.

Intrusion Detection

Effective intrusion detection requires the collection of relevant information, tools for analyzing this information, and mechanisms to respond to detected intrusions. Thus, intrusion detection is closely tied to audit and firewalls. Whether audit provides sufficient information depends on the analysis techniques, and the development of such techniques is still evolving. Further, as we mentioned before the introduction of additional hooks for audit into the Linux kernel, beyond those in LSM, will be controversial. Therefore, intrusion detection techniques based on the information that can be collected from the authorization hooks will be preferred.

Resource Control

Denial of service attacks are also important in the enterprise space. Presently, most preventative measures are taken at intermediate points in the network rather than at the victims themselves. However, authenticated communication in the kernel may provide the opportunity to manage network utilization better at the victims as well. This and other forms of resource control may prove useful although significant research is necessary.

Linux Security Directions

An important direction in Linux security is the development of the Linux Security Modules (LSM) framework [22]. LSM provides a basis from which MAC policies can be enforced by the kernel. It is envisioned that LSM will be adopted in Linux 2.5, and complete patches are available already for Linux 2.4.12. Thus, we believe that the time is ripe to evaluate the construction of comprehensive security solutions around LSM, so the enterprise system requirements outlined above can be met by future versions of Linux.

In this section, we describe the LSM project and examine how LSM is used in one of its first security modules, the SELinux module. We then discuss the issues of verifying the effectiveness and maintainability of LSM, and the needs beyond what LSM provides for developing a comprehensive Linux security solution.

Linux Security Modules Project

The Linux Security Modules (LSM) project aims to develop a generic interface behind which a wide variety of authorization mechanisms and policies can be implemented as loadable kernel modules in Linux. The LSM community is identifying all controlled operations in the Linux kernel. Prior to each controlled operation, LSM hooks in the form of function pointers are placed. Each hook passes the information deemed relevant to the authorization to the code that implements the function pointer. Modules that implement the LSM interface define the authorization behavior for each LSM function pointer. Typically, this behavior consists of the implementation of reference monitor-style authorization mechanism behind which a number of policies, including mandatory access control policies, can be implemented.

The LSM project was motivated by a variety of projects that aimed to improve Linux system security by authorizing client requests in the kernel. It has been known for a long time that authorization at the system call interface is insufficient because the mapping of file names to actual inodes can be changed between the time of authorization and time of use[3]. Therefore, effective authorization of system calls requires that the base kernel be modified to authorize at the locations of the security-relevant operations. Research projects, such as SELinux [12], RSBAC [15], and LIDS[8], and industry projects, such as Argus PitBull[2], Immunix[21], and HP Secure System Software for Linux[6], have all modified the base kernel in ad hoc ways in order to support their authorization mechanisms and policies. The SELinux project, in particular, was brought to the attention of leaders of the Linux community, as a demonstration of a complete and coherent Linux authorization framework. However, the authorization hooks in SELinux were dependent on the SELinux approach, so Linus proposed that a generic interface be authorization interface by developed for Linux 2.5. This proposal galvanized the community behind the LSM project which is led by Wirex, the makers of Immunix, NAI Labs (SELinux), and a number of independent developers.

LSM Use and Policy Issues

To demonstrate how to use LSM we examine the implementation of a particular module, the SELinux module. SELinux implements a reference monitor mechanism that implements a mandatory access control policy expressed in a variant of the Type Enforcement (TE) model[4]. The reference monitor mechanism used by SELinux has been evolved over several years of application to other systems, such as Mach and Fluke[13][18]. Its main feature is the separation between the policy (in TE) and the authorization mechanism.

TE consists of the following core concepts as implemented in SELinux: subject types (domains), object types, and permissions. Subject types are assigned permissions to the object types that they can operate upon. More complex concepts in TE include domain transition rules and assertions. Domain transition rules define when a process may change its security type and to what security types it may transition. Assertion are used to express consistency and safety constraints on a policy.

SELinux comes with a default policy written in a syntactically-sugared language for TE, but this policy is still quite complex. Therefore, it is difficult for a system administrator to determine what rights are available to a particular subject. More importantly in a MAC policy, a system administrator must be able to determine whether a subject will ever gain an unauthorized right. Assertions is the concept used to express these requirements, but assertions more complex concepts than permission assignments (rules, in general) and are not fail-safe. If an assertion is not specified, then unauthorized rights may become available. SELinux actually uses a second RBAC policy specification that bounds the TE specification to prevent errors from compromising core services.

These problems are typical of a general MAC model: the flexibility of the model makes it too complex to ensure that unauthorized rights are not accidentally available. Policies must be designed that demonstrate effective security and manageability. Further, tools are necessary to help keep the policy simple and examine the implications of policy changes.

LSM Verification

As the LSM project target is the definition of an acceptable generic interface, there are questions about whether the LSM interface is correct and maintainable. Using the historical definition of a reference monitor[1], the LSM interface enables implementation of a correct reference monitor if it monitors all the controlled operations in Linux. We are in the process of defining static and dynamic evaluation tools that identify controlled operations in the Linux kernel and verify whether they are run only if the expected authorizations are performed. Analysis of correctness in the presence of loadable modules is possible if we can define the authorization requirements at the function pointers which call the modules and the modules load do need only those authorizations.

The LSM interface is maintainable if it sufficiently easy to maintain the relationship between the location of the interface hooks and the controlled operations. We are also in the process of

implementing a tool that identifies interface hooks that may be difficult to maintain. It uses simple heuristics to estimate the controlled operations from the LSM interface and its placement. If the result is ambiguous or the controlled operation is far from the hook, then the tool identifies these hooks as having potential for improvement.

Comprehensive Security Around LSM

Lastly and most importantly, the LSM framework is initially only targeted at authorization. In order to build a secure system we need other features: authentication, audit, intrusion detection, and management tools to use them effectively. Further, we need the ability to leverage other security services, such as signed executables, encrypted file systems, and IPSEC.

At present, no plans exist for providing a trusted path for strong cryptographic authentication of subjects or programs for LSM, however. The default approach would depend on the Pluggable Authentication Modules (PAM) approach to provide subject identities much like Argus PitBull does now. While MAC can protect such services in general, it is not clear if the MAC policy will be simple enough to be trusted to protect the TCB. However, using MAC policies to protect these modules is the obvious default.

The simpler alternative, from a protection perspective, is to place the authentication services in the kernel. The kernel/user boundary would then protect the authentication service, and the TCB would be clearly limited to the kernel. Two things are needed to make this possible: (1) the addition of a authentication system call API and (2) the creation of an interface for the placement of authentication modules in the kernel. As the first will require changes to the base kernel system call interface, it will be the more difficult and controversial. Also, since most authentication protocols are complex and stateful, this API will have to be fairly complex. The addition of authentication module hooks should be fairly simple given the system calls and the current LSM authorization hooks.

Regardless of whether authentication is in the kernel or not, the semantics of setuid() must be changed. Only TCB processes are allowed to modify the subject identity of a process, so while setuid() must remain for backward compatibility, it will be “tamed.”

The LSM community envisions the addition of audit hooks in phase 2 of project. SGI is currently leading the LSM audit work based on the audit hooks placed in Trusted IRIX. It is not clear if the Linux community will accept audit-specific hooks for the base kernel. Thus, the default action should be to base audit on the LSM authorization hooks as much as possible. Additional audit hooks should identified based on requirements, such as intrusion detection systems.

Fortunately, most of the other security features can either be added as kernel modules or policies. The additional security feature that may require some integration are the signed executables support. Every effort should be made to hide this service behind the LSM authorization hooks, but work remains to verify that this is sufficient.

Linux Security Architecture

Given the stated requirements, we now outline the Linux system security architecture. First, we outline the major security decisions that must be made in the system. We then look at how each of the decisions can be implemented assuming that a Linux LSM framework is present. We then detail the major implementation stages.

Major Security Decisions

In Figure 1, an abstract system consisting of applications and a TCB that enforces comprehensive security is shown. We assume that an initial application, called the authentication application, and all remote data are assumed to be of the lowest security type (recall that a security type is a MAC security identifier for subjects and objects) initially. The following security decisions must be made by this system:

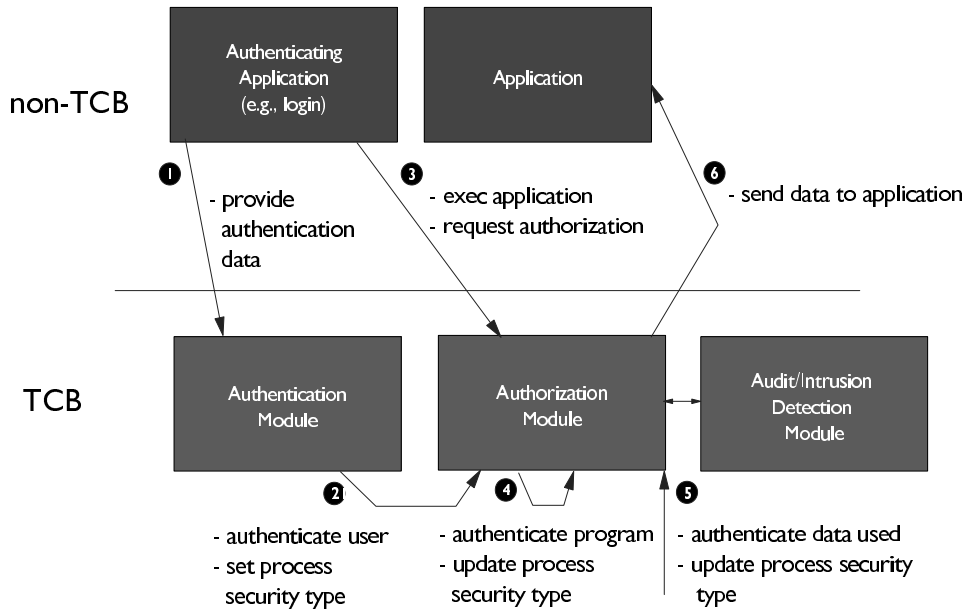

Figure 1. The primitive security operations of the TCB.

(1) can authenticating application authenticate as a particular subject?
(2) can the authenticating application have its security type increased to that of the subject?

(3) can the authenticating application load an executable into a process for that subject (e.g., a shell)? Or, more generally, can an application perform a particular system request? Further, can the resource utilization requested by the application be granted?
(4) what is the security type of the resultant process?
(5) what is the security type of network data objects? Or other objects used by a process?
(6) what is the security type of a process after accessing a particular object?
(7) can we detect intrusions attempted on the system?

There are also questions about who can determine and update the answers to these questions. This is particularly relevant given the problem of maintaining systems in a dynamic environment. However, logically this is the same problem as (3), authorizing a given operation.

Upon an authentication request (1), the authenticating application forwards the request to an authentication module in the TCB that determines whether the request is granted (i.e., the client can be authenticated as the requested identity). That is, the application must present information that proves that the subject knows the required secret for authenticating to it identity.

Upon a successful authentication, the authentication application can increase its security type based on the authenticated identity (2). The authorization module manages the assignment of processes to security types. However, the possible security types that a process can assume depend on the authenticated identity of the process's owner. The authenticated identity must be provided to the authorization module via a trusted and/or authenticated path.

The authentication application then creates a process (e.g., forks a shell or logically creates a new process by execing a shell) running at this security type for the user. The authorization module determines whether the authentication application can access and load this executable (3). Other system call requests are authorized in the same manner. Further, resource control may be performed for some requests, such as receiving a network packet. If these properties impact the current security type, then the authorization module may reduce the security type of the process (i.e., reduce the rights of the process by changing the security type).

Then, the integrity and any other security relevant properties of the program being loaded into the process are verified. The security type of the resultant application process is determined by the authorization module based on the security properties of the executable and the subject (4).

When untrusted network data is sent to the application, the kernel receives the data and can determine the security type of the data (5). For example, a data authentication module can verify the source and integrity of data (e.g., IPSEC). An encrypted filesystem can do the same for file data it delivers to the application.

Any time the kernel delivers data to an application it asks the authorization module whether the data can be delivered, as a typical authorization (4). Further, the authorization module can lower the security type of the application based on the security type of the data that it receives (6). However, it is possible for the authorization module to upgrade data prior to its delivery to

applications based on system policy. Research is necessary to determine the ultimate policy model and policies needed for the various enterprise application requirements (e.g., web hosting).

Of course, all interactions between the applications and TCB modules are targets for auditing and intrusion detection (7). Note that each of the decisions are based on the security policy of the system. Therefore, the policy modules for (1) through (7) must also be discussed.

Architectural Decisions

Below, we identify the key modules and policies required for a secure enterprise system. However, we must consider the current Linux architecture and available technologies in determining how to construct such a system. Below, we list the current status of the individual modules and policies and the desired status.

User authentication module:

Current: User-level PAM authentication via passwords
Desired: Protected, mandatory authentication services that use strong cryptography

The state of the art in Linux is PAMified authentication services that use setuid() to tell the kernel the security type (i.e., userid) of the client. The main improvements are to: (1) require TCB authentication of users; (2) provide mandatory protection of the authentication services; and (3) enable the use of strong cryptography for authentication. MAC protection can be enforced at user-level via a MAC policy or placing authentication services entirely in the kernel. It is not clear which will prove to be the best practical solution for Linux. The authentication services must support both symmetric key (e.g., Kerberos) and asymmetric key authentication protocols. A cryptolibrary with the necessary functionality must be made available.

User authentication policy:

Current: User-level password files
Desired: Trust management policies, such as SDSI[9] or KeyNote[17]

Currently, password authentication is used to set the user identity in Linux or security type in SELinux. Trust management protocols enable flexible authentication, in particular support for multiple authentication protocols. It must be determined how much flexibility is necessary for enterprise applications. Also, the trust model for modifying or extending this policy must be determined.

Authorization module:

Current: Kernel-specific, DAC mechanisms
Desired: Kernel-independent, kernel internal, Linux LSM module

The LSM framework enables the construction of kernel-level authorization modules. A few are under development, including a version of SELinux. The basic SELinux module has a number

of suitable features (e.g., caching, consistency management, etc.), but has some performance issues in a enterprise environment (e.g., multiprocessor) that need to be addressed.

Authorization policy:

Current: No particular MAC model has been adopted, UNIX DAC model is standard
Desired: MAC policy, aimed at simple containment of applications

A variety of MAC models exist, but no single model or policy expressed using those models is dominant. We expect that SELinux's use of two models, roles for coarse-grained safety protection of core services and subject types for fine-grained access control, will be the starting point. Collection of the various ad hoc policies into a single effective policy (e.g., with new ideas, such as multiple /tmp directories) in the face of updates is necessary. Also, research of tools to aid system administrators in managing policy is underway.

Executable authentication module:

Current: Kernel-level, signed executable frameworks in prototype stage
Desired: Mature, kernel-level, signed executable framework

Prototype executable authentication modules, including IBM Research's signed executable framework and Wirex's Cryptomark, are under development. In fact, integration of the two approaches has been initiated. Integration with LSM is necessary and quite likely, given IBM and Wirex's support of LSM.

Executable authentication policy:

Current: None
Desired: Integrity verification based on digitally-signed digests, including source and freshness

In Linux, no executable authentication is done at present. Initial prototypes have been developed to authenticate the integrity of programs. Further, the source and/or freshness of a program may have an impact on access control decisions, particularly for mobile code. Determining the breadth of requirements that must be met is important to designing effective policy support.

Data authentication module:

Current: Data-specific, such as SSL (user-level) and IPSEC (kernel-level) for network data, some cryptographic file system prototypes for files
Desired: Comprehensive, kernel-level approach consolidating into as few approaches as possible

Currently, most data authentication is done for network data in applications using SSL. In order to implement MAC policies that provide comprehensive secrecy and integrity protection, such protection must be done in the TCB, preferably via IPSEC. Protection of the secrecy and integrity of files is also desirable, although such systems are less mature. Data authentication that covers all forms of data delivered between processes in a complete and comprehensive way would remove the ad hoc nature of such services, but is not yet underway.

Data authentication policy:

Current: Ad hoc, if present
Desired: Must determine how the use of data impacts the security type of the user of that data

Currently, policies are specific to the service, such as IPSEC. Also, integration of the data authentication with other forms of authentication would enable least privilege control, by associating a system-wide authenticated identity with all users, data, and programs.

Audit/Intrusion detection module:

Current: Syslog and kernel logging; ad hoc intrusion detection
Desired: Ability for complete audit, B1-level auditing

Currently, it is possible to log operations, but this is done in an ad hoc way in typical UNIX systems, if at all. The goal is to enable audit of any security decision. Thus, integration of auditing with the LSM authorization module is a first step. SGI has taken the lead in the LSM community, although significant work remains to get audit into LSM. Further, integration with the other modules listed above is necessary for sufficient auditing. Intrusion detection mechanisms are quite ad hoc at present, but we envision that these will be able to use the generated audit logs.

Audit/Intrusion detection policy:

Current: Ad hoc
Desired: Comprehensive policy integrated with authorization

Currently, when to audit is usually hard-coded into the operating system or application. Some applications use audit as a means to track operations that are not normally authorized. In these cases, audit policy will be integrated with authorization. In other cases, additional audit policy specifications may be expressed. The nature of this kind of policy is not well-understood.

As is apparent from the list, very few of the desired architectural modules or policies are currently supported. The LSM framework is the only one that appears to be maturing quickly. Others, such as executable authentication, have been prototyped and are evolving into mature frameworks. Further, mandatory policies for these frameworks are an ongoing topic of research. Thus, while we would like to have the desired Linux security services, we must take a pragmatic approach in which achievable security targets improve Linux incrementally towards the the ultimate goals. We describe the incremental targets in each area below.

Security Targets

We describe incremental system security targets for the individual decision points listed above. For each decision point, we describe three successive targets: (1) near-term target that provides significant, but basic, security improvements; (2) a mid-term target that provides the features we

expect to require in a general way for the enterprise; and (3) an ultimate target that may or may not be achievable, but that indicates the vision for comprehensive enterprise security in Linux.

User authentication Module:

PAMified, MAC-protected authentication service (target 1):

A PAMified MAC-protected authentication service is used for user authentication (e.g., login()). This service runs in an authentication TCB security type which permits it to change to another security type upon authentication (i.e., when the service believes that the authentication is successful). Remote login programs (e.g., SSH) are run the same way. In this case, a new interface is needed to set the uid of another process and select the security type for this identity for the allowed types. Only the TCB service can use these two interfaces. Setuid() has no effect on the security type.

Kernel authentication module (target 2):

As an alternative, we develop a kernel-level authentication mechanism. To keep the task simple initially, a password-based module called from the appropriate PAMified service sends the password data to the kernel. The kernel authentication module verifies the password and caches the authenticated identity for the process. The process then uses setuid() to set the identity of the process. Further, security types can be selected using the interface defined in target 1.

An additional system call is necessary to transfer the authentication information to the kernel. We should also examine whether this interface is sufficient for strong cryptographic protocols, such as Kerberos and public key protocols, like SSL. If so, then a generic kernel interface to authentication can be defined. Other features necessary to implement strong cryptography effectively, such as crypto libraries and PKI, need to be explored at this time.

Comprehensive authentication (target 3):

Based on the results of the first two targets, an architecture for comprehensive authentication based on a set of user, executable, and data authentication modules could be designed and prototyped.

User authentication Policy:

Password authentication and beyond (target 1):

Initial PAM modules will continue to use password authentication, but prototypes should be developed that use strong cryptography, preferably public key cryptography.

The requirements for authentication of different key applications should be determined. The exact requirements for authentication policy depend on the types of authentication required. Since we aim at the enterprise for scalability, a few distributed applications should be chosen as representatives, and policies should be derived based on their needs.

Flexible choice of authentication policies (target 2):

A policy model supporting the authentication requirements found above should be prototyped and evaluated. As a starting point, trust management policy languages, such as SDSI and KeyNote should be examined, but we tend to believe that a simpler approach will suffice.

Comprehensive authentication policies (target 3):

A policy model that supports user, executable, and data authentication policies and their impact on security types should be examined. Convergence to a single policy framework would have some benefits if the complexity can be kept manageable. However, it is unclear whether this is the case.

Authorization module:

Use and improve existing modules and hooks (target 1):

First, the authorization requirements of the key applications should be identified to determine whether these requirements can be met using the current LSM hooks. The most mature comprehensive module is the SELinux module. We envision using this module as a starting point. Given the NSA's interests in comprehensive security, we recommend working with the NSA on any shortcomings in the module that may be found.

Verification of the correctness and suitability of the LSM hooks is also necessary. We recommend verification that the LSM hooks enable the desired authorizations at the various dangerous operations in the kernel and meet basic maintainability requirements. The research of automated tools for these purposes is underway.

Module/hook revision (target 2):

Any module or hook revisions that are found to be necessary will be undertaken for target 2. Also, performance testing and system hardening using the resultant LSM system is necessary.

Other tools necessary to aid the kernel developers in the management of the LSM hooks should be investigated.

System acceptance and maturation (target 3):

At this point, the LSM acceptance and use in a broad sense should be demonstrated and promoted.

Authorization Policy:

Simple policies and requirements (target 1):

SELinux also has the most mature MAC policy for Linux. However, this policy is rather complex, so we consider effective MAC policies for Linux as an open research issue. We recommend working with the NSA to develop their policy and management tools for it.

We also recommend examining the use of simple, integrity policies, such as can be expressed using LOMAC [11] or a simple RBAC [14] model, for the coarse-grained policy. The policies to be explored should be derived from the enterprise system examples and take into account system updates. The goal at this stage is to provide strong integrity guarantees for the TCB and critical applications. The safety of our applications is not required, although least privilege permissions are preferred.

Comparison of alternatives (target 2):

At this point, it will be useful to express authorization policies for our example applications in a variety of models and evaluate the effectiveness of each. Also, policy tools for managing updates and verifying the security properties, such as safety, should be prototyped at this stage.

Comprehensive policy and management (target 3):

Ultimately, we expect that comprehensive models will be developed that indicate how coarse-grained safety, fine-grained least-privilege, and auditing/intrusion detection policies work together. The policy development environment should enable expression, querying, and analysis of policies.

Executable Authentication Module:

Prototype module (target 1):

The combination of IBM's signed executables and Wirex's Cryptomark frameworks should be developed that is integrated with LSM. Collaboration with Wirex is being initiated here.

Prototype experience (target 2):

Experience with the prototype and application requirements should be collected to determine which security properties need to be verified in practice. Enhancements to the framework necessary to support these requirements should be developed.

Integration (target 3):

Executable authentication should become part of a comprehensive authentication framework. In particular, this will mean: (1) integration with data authentication to verify the security properties of executables; (2) integration with encrypted filesystems to store authenticated files in a way that preserves their integrity; and (3) integration with the authorization modules to impact the security types to which this executable may be assigned.

Executable Authentication Policy:

Integrity only (target 1):

Ensure that the integrity of the code executed by the system is verified using the initial prototype.

Other properties (target 2):

Determine the other security properties that may require verification for the key applications, and examine policy models that can enable the expression of such requirements. For some applications, more complex requirements, approaching that of mobile code, may be required. In particular, we may need to verify the source, and perhaps even the freshness, of the program in order to determine the security type that it should be assigned.

At present, little or no work has been done in this area, so existing policy approaches may not be sufficient.

Comprehensive policies (target 3):

A comprehensive policy store executable authentication, data authentication, and user authentication requirements in a single place. These requirements state what properties must be verified to achieve a particular security type and how the use of that type affects the processes' security types.

Data Authentication Module:

Individual modules and use (target 1):

Initially, data authentication will be done in an ad hoc way by a variety of tools. We must identify acceptable, although not necessarily the most secure, solutions for the enterprise that require a small number of modifications. This will require gathering the basic data authentication requirements for the key applications, and analysis of the ability of existing approaches to achieve these requirements.

At least the authentication of file and network data should be examined. For example, different prototype systems for encrypted file systems should be evaluated and their strengths and weaknesses identified. For network data, the main problem is to convert user-level data authentication using SSL to kernel-level using IPSEC. Therefore, a simple IPSEC-based version of a key application should be prototyped.

Encourage secure solutions (target 2):

Ultimately, applications should be modified to use IPSEC for network data and an encrypted filesystem for file data. Knowledge learned in the development of the first target will provide input to the development of a strategy to encourage the use of strong, mandatory data authentication.

Comprehensive data authentication (target 3):

Further, consolidation of various data authentication schemes into a single comprehensive approach may help keep management and implementation simpler. This is an open research topic.

Data Authentication Policy:

Ad hoc policies (target 1):

Policies that ensure the secrecy and integrity of different types of application data should be developed as the first step. A few simple policy options will be supported initially for a few different data types.

File and network policies (target 2):

Next, the requirements of the key applications will drive the policy model requirements. Again such policies are similar to trust management policies, so that may be a starting point for models.

Also, we would like to consolidate objects into a small number of types to limit the ad hoc nature of data authentication. Probably, only network and file object types need to be used.

Comprehensive data authentication (target 3):

In this phase, we explore integration of the various authentication policies and the impact that may have on simplifying management. By this time, the results of data authentication should be usable for making authorization decisions in integrity policies, like LOMAC.

Audit/Intrusion Detection Modules:

No audit/intrusion detection work is part of target 1.

Initial authorization audit (target 2):

Apply LSM authorization hooks for audit purposes and identify addition audit hook requirements. SGI is leading the effort to expand LSM to meet audit requirements.

Authentication and beyond audit (target 3):

Implement additional authorization audit hooks and determine the requirements for audit in kernel-level authentication. Initial prototype of authentication audit hooks.

Audit/Intrusion Detection Policy:

No audit/intrusion detection work is part of target 1.

Initial audit/intrusion detection policies(target 2):

Develop an approach for the expression of audit based on the use of LSM authorization hooks. Basic intrusion detection using such logs should be examined.

Also, some intrusion detection techniques should be expressed as a policy that generates audit logs and performs analysis.

Authentication audit and intrusion detection policies(target 3):

Revise and extend initial policies from target 2 based on analysis of this prototype and the availability of new hooks. Develop policies that can use authentication audit hooks.

Architecture Summary

This architecture summary shows the evolution from an LSM-based Linux kernel to that which can provide the enterprise level security that we desire based on the targets described above.

First System Target

For this target, the key steps are providing mandatory access control and mandatory authentication in the most direct manner. This target contains a user-level authentication service protected by an LSM module's MAC policy, either SELinux (from NAI Labs) or a simple MAC module for LOMAC. The signed executable prototype is introduced at this stage (probably from Wirex), but only to provide integrity verification for modules. Data authentication is little changed, often SSL is used by applications, rather than IPSEC. Thus, the MAC module must support upgrading untrusted network data to applications that it believes can handle this data effectively.

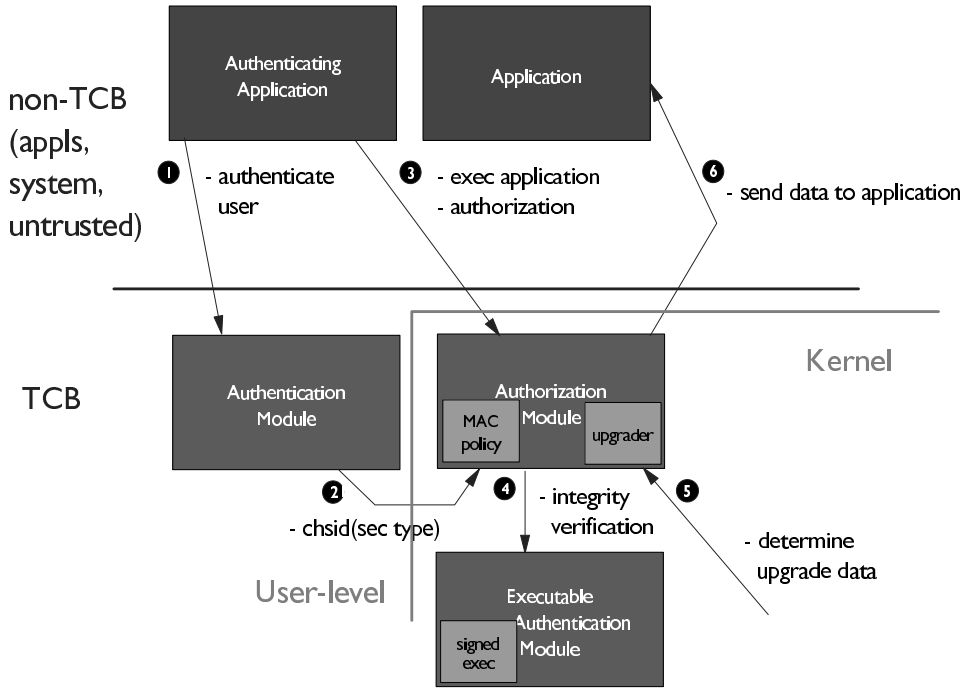

Figure 2. First security target architecture: (1) authentication outside kernel; (2) signed executables added; and (3) support for MAC policies.

Figure 2 shows how the first target for the Linux security architecture implements the security decisions. The authentication modules are some PAMified services (e.g., ssh, login) that enable a user to authenticate to the TCB (message 1). These services run at user-level, but are still part of the TCB. SELinux and Argus PitBull demonstrate how such modules may be implemented.

Further, improvements, such as the use of strong cryptography, should be explored. The authentication service uses a new system call to change the authenticated identity of processes, chsid(), which is only usable by TCB services (message 2). The security type for the process can be set by an additional system call, chtype() or via a "tamed" setuid(). The security types are limited by the authenticated identity set using the chsid() inputs. Then, an LSM MAC module will implement authorization according to a simple policy aimed at protecting system integrity (message 3). As we stated above, signed executable prototype will be available and integrated with LSM to ensure the integrity of executables (message 4). Changing the process's security type based on the programs loaded will not be implemented at this time. Lastly, a key problem is verifying that untrusted data coming from the Internet can be delivered to the web server application (which has access to secret data, such as credit card numbers) in a secure fashion (message 6). Work on effective upgraders will be necessary.

Second System Target

In the second system target, we aim to get initial versions of all security services into the kernel. This has a minor impact for security improvements, but will make it much easier to collect all security relevant information into one place for future enhancements.

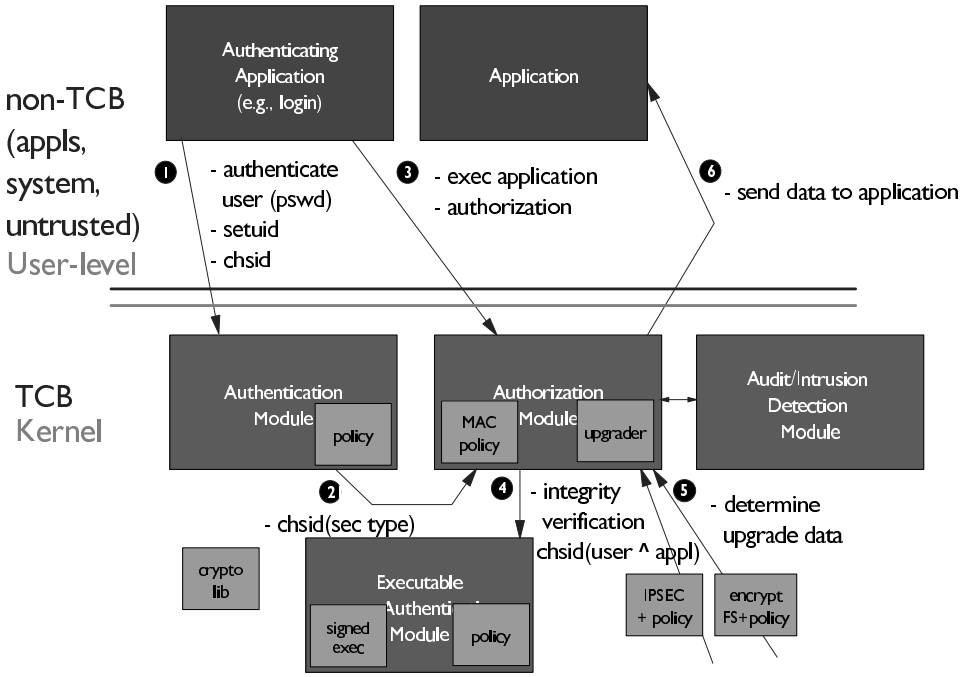

Figure 3. Second system target includes: (1) kernel-level authentication and crypto library; (2) application-driven security policies, including initial resource control; (3) initial audit and intrusion detection; and (4) cryptographic file system.

In Figure 3, we show the second target for the Linux security architecture. For this target, we do the following: (1) add the authentication module to the kernel, including the necessary crypto library support; (2) add initial audit/intrusion detection support; (3) develop application MAC policy requirements more comprehensively, including initial resource control; and (4) add prototype systems for data authentication, such as encrypted file systems, and increase the use of kernel data authentication.

We envision that the password data from the first target's PAMified service will be forwarded to the kernel service first (message 1). This service will implement the authentication as before to verify the password and set the user's identity (message 2). This enables the process to set security types or use the "tamed" setuid() as before. The authorization mechanism will remain basically unchanged except for any discovered improvements (message 3). Authorization policies will become more comprehensive and management tools should make their first appearance. Next, initial audit and intrusion detection support should be added (all messages). Hooks to achieve B1 level audit should be the goal, although at this point we expect that only LSM authorization hooks will be available. Also, we expect that some experience with the application requirements will have been gained, such that comprehensive policy modeling requirements can be determined. Policies for authentication will go beyond simple integrity checking to determining whether rights may be modified based on source and freshness of data and programs (message 4). Lastly, we envision that data authentication should be made more comprehensive and concentrated further into the kernel. We expect that a prototype encrypted file system should be functional. Also, transfer of some user-level data authentication to IPSEC should be prototyped (message 5).

Third System Target

The main goal of the third target for the Linux security architecture is the integration of authentication and authorization modules into a comprehensive security framework in the kernel. User, data, and executable authentication should be performed in an integrated fashion regardless of the source of the authentication information. For example, it should be possible for users establishing VPNs using IPSEC to be managed using a security type understood by the authorization system. Also, policies and policy management tools for supporting our key applications should be mature at this point.

In Figure 4, authentication is still performed in the kernel, but a variety of means for transferring authentication data to the authentication module are possible, such as IPSEC, local applications, and other remote clients (message 1). Authentication proceeds sets the authenticated identity as before (message 2), but mandatory authentication is now possible for all authentication services that the system provides. It may be possible to have a single authentication policy for all. Authorization also remains the same (message 3), but the policies and policy management infrastructure should be mature enough to make strong security claims for safety. Executable authentication is integrated with the secure network and file access (message 4), so the authentication services are common. Lastly, the use of network and file data impacts the permissions of the process that uses this data in reasonable ways based on our experience with applications (message 5). Dynamic restriction of permissions based on the access of low

integrity files or network communications is possible. All operations can be monitored, audited, and analyzed for intrusion detection according to the system policy.

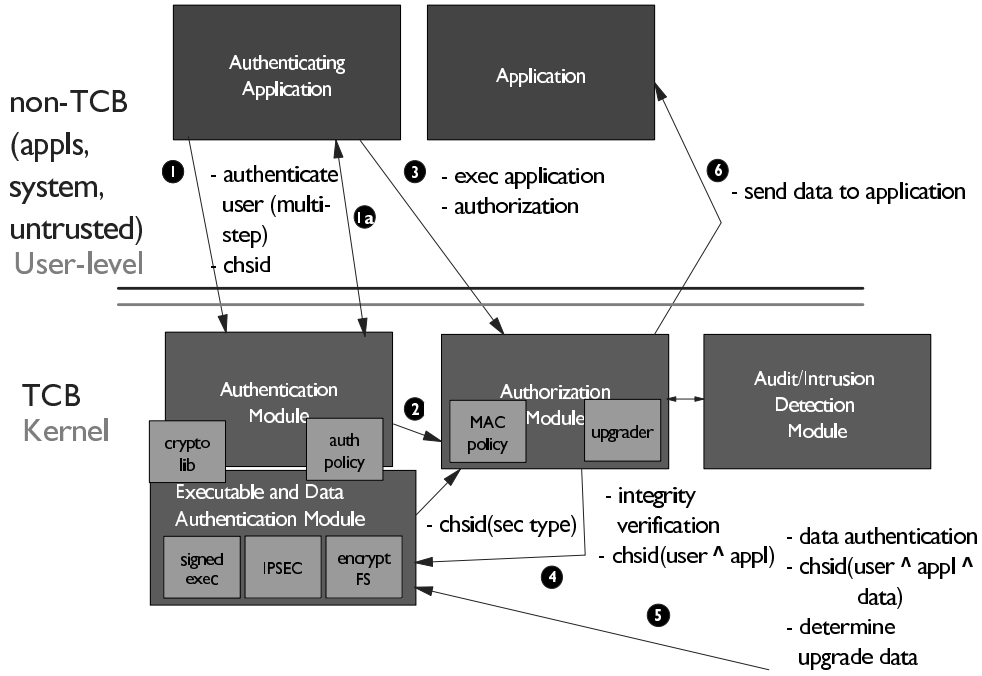

Figure 4. Third system target includes: (1) integrated and comprehensive data authentication; (2) mature, application-driven security policies; (3) scalable authentication of users regardless of location.

References

1. James P. Anderson. Computer Security Technology Planning Study. Tech report ESD-TR-73-51, Electronic Systems Division, Hanscom AFB. 1972.
2. Argus Systems. Argus PitBull LX. At http://www.argus-systems.com.
3. Matt Bishop and Michael Dilger, UC Davis. Checking for race conditions in file accesses. At http://nob.cs.ucdavis.edu/~bishop/papers/Pdf/racecond.pdf.
4. W. Boebert and R. Kain. A practical alternative to hierarchical integrity policies. Proceedings of the 8th National Computer Security Conference. 1985.
5. Leendert van Doorn et al. Signed executables for Linux. University of Maryland tech report CS-TR-4259. 2001.
6. Hewlett-Packard. HP secure OS software for Linux. At http://www.hp.com/security/products/linux/.
7. International Kernel Crypto API for GNU/Linux. At http://cryptoapi.sourceforge.net.
8. LIDS organization. Linux Intrusion Detection System. At www.lids.org.
9. Massachusetts Institute of Technology. Cryptography and Information Security Group Research Project: A Simple Distributed Security Infrastructure (SDSI). At http://theory.lcs.mit.edu/~cis/sdsi.html.
10. Andrew McDonald. StegFS: A steganographic file system for Linux. At http://www.mcdonald.org.uk/StegFS.
11. NAI Labs. LOMAC: MAC you can live with. At opensource.nailabs.com/lomac.
12. NSA. Security Enhanced Linux (SELinux). At http://www.nsa.gov/selinux.
13. NSA and Secure Computing Corp. Distributed Trusted OS (DTOS). At http://www.securecomputing.com/randt/HTML/dtos.html
14. Ravi Sandhu et al. Role-based access control models. IEEE Computer, February 1996.
15. Rule Set-based Access Control (RSBAC) for Linux. At www.rsbac.org.
16. P. Samarati and R. Sandhu. Access control: Principles and practice. IEEE Communications, September 1994.
17. University of Pennsylvania. The KeyNote Trust-Management System. At http://www.cis.upenn.edu/~keynote/.
18. University of Utah. Flask: Flux advanced security kernel. At http://www.cs.utah.edu/flux/fluke/html/flask.html.
19. David Wheeler. Counting Source Lines of Code (SLOC). At http://www.dwheeler.com/sloc.
20. Wirex Corp. Cryptomark. At http://www .immunix.org/cryptomark.html.
21. Wirex Corp. Immunix Security Technology. At http://www.immunix.com/Immunix/index.html.
22. Wirex Corp. Linux Security Modules (LSM). At lsm.immunix.org.