Achieving CAPP/EAL3+ Security Certification for Linux

Kittur (Doc) S. Shankar
IBM Linux Technology Center
dshankar@us.ibm.com

Olaf Kirch
SUSE Linux AG
okir@suse.de

Emily Ratliff
IBM Linux Technology Center
emilyr@us.ibm.com

Abstract

As far as we know, no Open Source program has been certified for security—until now. Although some people believed that it was not possible for an Open Source program to receive a security certification, we have proven otherwise by obtaining a Common Criteria security certification for SuSE SLES 8 SP3. With the increasing use of Open Source in general and Linux in particular within government and commercial environments, security of Open Source products is of increasing importance and as a result the demand for the security evaluation of Linux is evident. It is also generally believed that security certifications are time consuming and can take years to accomplish. We were able to obtain the Common Criteria certification of Linux in a few months. The presentation will cover our experience and the technical challenges associated with this Linux evaluation. In particular, we will discuss the enhancements we made to SLES 8 SP3 including the Linux kernel to support CAPP audit requirements. In addition the business advantages of the evaluation for Open Source software will be covered.

1 Introduction

In promoting Linux to IBM's enterprise and government customers, the requirement for Common Criteria certification emerged as a barrier to entry. All of Linux's commercial competitors have the required level of certification. As Linux continues to be adopted by the enterprise market, many customers, especially those from the government sector, have raised concerns regarding Linux security and questioned whether Linux was capable of achieving certification. These customers view security certification as table stakes for proving a minimal level of operating system security. In order to increase Linux adoption by these customers, certification is required. The expense of achieving certification makes certification unobtainable by community projects without corporate or government sponsorship. For these reasons, and after a careful analysis, IBM decided to sponsor a Common Criteria (CC) security certification for Linux. SUSE agreed to partner with IBM to evaluate SUSE LINUX Enterprise Server 8 (SLES 8).

In this paper, we will begin with a brief overview of the Common Criteria standard. We will then describe our approach and expe-

rience during this certification effort. We will describe in detail the additional functionality that was needed in kernel and user space to fulfill the requirements of the certification. We will also describe the level of documentation and test needed to obtain the certification.

Throughout the paper, we use the pronoun 'we.' By 'we,' we mean individuals or subteams from the large team of people who contributed time and effort in achieving this evaluation, including:

- IBM, the evaluation sponsor
- SUSE, the developer
- atsec information security GmbH, the evaluator
- BSI, the German agency for information security, the evaluation body

2 Common Criteria Overview

Common Criteria (CC) is documented in the ISO standard 15408 for the security analysis of IT products. The governments of 18 nations have officially adopted the Common Criteria, including the United States, Canada, Germany, France, and the UK. The U.S. government has required a Common Criteria evaluation for all IT-products used for the processing of security-critical data since July 1, 2002.[1]

Common Criteria splits the requirements into two sets: functional and assurance. Functional requirements describe the security attributes of the product under evaluation. Assurance requirements describe the activities that must take place to increase the evaluator's confidence that the security attributes are present, effective, and are designed and implemented correctly. Examples of assurance activities include documentation of the developer's search for vulnerabilities and testing.

[1]The requirement is codified by NSTISSP No. 11.

2.1 Functional Requirements

The functional requirements desired by the customer are described in the Protection Profile (PP). Protection Profiles are targeted at specific types of systems. For example, there are unique protection profiles for operating systems, firewalls, databases and other complex or security sensitive products. Protection Profiles are often created by the product developer, standards bodies, or government agencies, rather than by the customer. To be officially recognized, the Protection Profile must itself be evaluated. Protection Profiles are intended to be reusable and thus typically define standard sets of security attributes that can be used to compare different implementations of a product type. The name of the Protection Profile is therefore often used as shorthand to describe the functional level of the evaluation.

The product being evaluated is known as the Target of Evaluation (TOE). The security policy used by the TOE is known as the TOE Security Policy (TSP) and the functionality that enforces the TSP is known as the TOE Security Functions (TSF). The TSP may be enforced by software, hardware or firmware, but no matter what the enforcement mechanism is, the enforcement functionality is included in the TSF. The TOE does not exist in a vacuum; external forces that act on the TOE are known as the TOE (security) environment. The TOE environment may consist of elements such as non-privileged processes running in an operating system and the network to which a system is attached. The main purpose of an evaluation is to determine whether or not the TSP is correctly enforced.

2.2 Assurance Requirements

Evaluated Assurance Levels (EALs) are defined on a scale of increasingly rigorous development methodologies. The Common Criteria defines multiple classes of assurance components with multiple levels of difficulty for each component. The assurance levels are then composed from these components. These components include items such as level of documentation and testing. The assurance components used for this evaluation are described in more detail in the EAL3 Overview Section. Each higher assurance level requires more proof that security was a fundamental element of the development process; therefore, each higher level is more difficult to achieve than the previous level. There are seven ordered EALs[2]:

- EAL1 – Functionally Tested
- EAL2 – Structurally Tested
- EAL3 – Methodically tested and checked
- EAL4 – Methodically designed, tested and reviewed
- EAL5 – Semiformally designed and tested
- EAL6 – Semiformally verified, designed and tested
- EAL7 – Formally verified, designed and tested

EAL1 is the entry level assurance level. EAL4 is the highest assurance level that any product is expected to be able to achieve without significant expense and rework if it had not been specifically developed with Common Criteria evaluation in mind.

[2]Common Criteria Part 3 available from `http://csrc.nist.gov/cc/Documents/CC%20v2.1%20-%20HTML/CCCOVER.HTM`

2.3 Evaluation Approach

When the developer has decided on a Target of Evaluation and a Protection Profile, the first step towards evaluation is writing a Security Target (ST) which describes the security objectives of the TOE and how they meet the security requirements defined in the chosen PP. It is possible for an ST to claim conformance to multiple PPs or no PP at all. The claims in the security target determine the scope of the evaluation. Every facet of the evaluation is directly impacted by what is claimed in the Security Target. After the evaluation is completed, the Security Target is always made available for customer scrutiny so that the customer can understand exactly what was evaluated.

3 Description of the Evaluated TOE

Our target of evaluation (TOE) was the SUSE LINUX Enterprise Server 8 operating system with Service Pack 3 and the certification-sles-eal3.rpm package.

The SLES evaluation covers a distributed, but isolated, network of IBM® xSeries®, pSeries®, iSeries®, and zSeries® servers running the evaluated version of SLES. The hardware platforms selected for the evaluation consisted of commercially available machines from across the IBM product line.

The TOE Security Functions (TSF) consist of Linux kernel functions plus some trusted processes. These functions enforce the security policy as defined in the Security Target. The TOE includes standard networking applications, such as ftp, ssh, ssl, and xinetd. System administration tools include standard admin commands. Yast2 and several yast2 modules were also included in the package list that formed the TOE. The X Window System was

not included in the evaluated configuration.

The hardware and the system firmware are not considered to be part of the TOE but rather are a part of the TOE environment. The TOE environment also includes applications that are not evaluated, but are used as unprivileged tools to access public system services. For example, an HTTP server using a port above 1024 (e.g., on port 8080) can be used as a normal application running without root privileges on top of the TOE. The Security Guide provides guidance on how to set up a http server on the TOE without violating the evaluated configuration.

4 ST Description

The Security Target specifies that the evaluation covers the Controlled Access Protection Profile (CAPP) functionality at the EAL3 augmented assurance level.[3] The primary security features and assurance documentation are described below, along with how the requirements were satisfied. The key features are supported by domain separation and reference mediation, which ensure that the features are always invoked and cannot be bypassed. Most of the security and assurance features are included in the vanilla kernel (e.g., object reuse) or are standard to most Linux distributions (e.g., PAM, OpenSSH, OpenSSL) and were thus already present in SLES 8. A few, most notably audit, had to be added for the evaluation.

4.1 EAL3 Overview

EAL3 provides assurance by an analysis of the security functions, using its functional and interface specifications, guidance documentation, and the high-level design of the TOE to understand the security behavior. The EAL3 assurance requirements fall into the following seven categories:

- Configuration Management
- Delivery and Operations
- Development
- Guidance Documents
- Life Cycle Support
- Security Testing
- Vulnerability Assessment.

Many of the documents created to support the assurance requirements can be reviewed at `http://oss.software.ibm.com/linux/pubs/?topic_id=5`

4.1.1 Configuration Management

The Configuration Management assurance class specifies the means for establishing that the integrity of the TOE is preserved during development. The Configuration Management process must provide a mechanism for tracking changes and ensuring that all the changes are authorized.

Configuration management procedures within SUSE are highly automated using a process supported by the AutoBuild tool. Source code, generated binaries, documentation, test plan, test cases and test results are maintained under configuration management. Because of this, SUSE already exceeded the requirements for this evaluation, so we just had to document existing procedures to fulfill this requirement.

This assurance requirement is the one that was commonly expected to be the source of difficulty in achieving certification of code developed via the open source methodology. The

[3] The augmentation is the flaw remediation procedure.

key to meeting this assurance requirement is that every line of new code that comes into the SUSE AutoBuild environment is assigned to an owner within SUSE who becomes responsible for its integrity.

4.1.2 Delivery and Operations

The Delivery and Operations class provides requirements for the assurance that the TOE is not corrupted between the time the developer releases it and the customer fires it up.

SLES is delivered on CD/DVD in shrink-wrapped package to the customer. SUSE verifies the integrity of the production CDs and DVDs by checking a production sample. Service Pack 3, the certification-sles-eal3.rpm package, as well as other packages that contain fixes must be downloaded from the SUSE maintenance Web site. Because those packages are digitally signed, the user is both able to and required to verify the integrity and authenticity of those packages. Guidance for installation and system configuration is provided in the Security Guide.

Again, existing SUSE processes met the requirements for the EAL3 assurance level, so documenting existing procedures was sufficient for this evaluation.

4.1.3 Development

The Development class encompasses requirements for documenting the TSF at various levels of abstraction, from the functional interface to the implementation representation. For EAL3, we needed a functional specification and a high-level design. In addition, the correspondence between the security functionality, the functional specification, and the high level design had to be documented.

The functional specification for SLES consists of the man pages that describe the system calls, the trusted commands, and a description of the security-relevant configuration files. A spreadsheet tracks all system calls, trusted commands, and security-relevant configuration files with a mapping (correspondence) to their description in the high-level design and man page(s). The high-level design of the security functions of SLES provides an overview of the implementation of the security functions within the subsystems of SLES, and points to other existing documents for further details where appropriate.

To fulfill this requirement, the functional specification spreadsheet, correspondence, and high-level design were written. Additionally, several new man pages were created for undocumented system calls, PAM modules and utilities, and many man pages required minor corrections.

4.1.4 Guidance Documents

The Guidance Documents class provides the requirements for user and administrator guidance documentation. A security guide is also necessary to fulfill the requirements of this class at EAL3.

SLES 8 already shipped with User and Administrator Guides. The Security Guide and a special README file were created that contain the specifics for the secure administration and usage of the evaluated configuration. The Security Guide explicitly documents setting up and maintaining the system in an evaluated configuration.

4.1.5 Life Cycle Support

The Life Cycle Support assurance requirement includes requirements for processes that deal with vulnerabilities found after release of the product, as well as the physical security of the developer's lab.

The SUSE security procedures are defined and described in documents in the SUSE intranet. The defect handling procedure SUSE has in place for the development of SLES requires the description of the defect with its effects, security implications, fixes and required verification steps.

Again, existing (and previously planned updates to) SUSE procedures met the requirements for this class and were merely required to be documented to fulfill the assurance requirements.

4.1.6 Security Testing

The emphasis of the Security Testing class is on the confirmation that the TSF operates according to its specification. This testing provides assurance that the TOE satisfies the security functionality requirements. Coverage (completeness) and depth (level of detail) are separated for flexibility.

A detailed test plan was produced to test the functions of SLES on each evaluated platform. The test plan includes an analysis of the test coverage, an analysis of the functional interfaces tested, and an analysis of the testing against the high level design. Test coverage of internal interfaces was defined and described in the test plan documents and the test case descriptions. The tests were executed on every platform. The test results are documented so that the tests can be repeated and the results independently confirmed.

Although, SUSE has an excellent test infrastructure for regression testing already in place, additional tests were required to test new functionality, such as audit, and ensure coverage of security relevant events. The Linux Test Project provided an excellent base for the test suite needed for EAL3. It already contained almost all of the necessary test cases for every system call. In some cases, we had to add tests of expected failure cases to ensure that the security was being correctly enforced. We added some test cases for security-relevant programs, such as su, cron, at, and ssh. We created tests to ensure that the system was configured in the evaluated manner. We also created many tests for correct ACL behavior. Many of the system call and security-relevant program test cases were created during the course of the EAL2 evaluation and then reused during the EAL3 evaluation. The largest class of new test cases for EAL3 was tests of the new audit system. Testing the audit subsystem required showing that all security-relevant system calls are logged correctly, all trusted programs (including PAM) correctly logged security-relevant events, the audit userspace tools contained correct functionality, and that audit exhibits Controlled Access Protection File (CAPP)-compliant behavior during threshold and failure events (for example, low disk space). Gcov was used to show test coverage of the kernel internal interfaces. Writing, documenting, and running these test cases on all of the evaluated platforms was a significant portion of the evaluation effort.

4.1.7 Vulnerability Assessment

The Vulnerability Assessment class defines requirements for evidence that the developer looked for vulnerabilities that might arise during development and use of the TOE.

Our search for vulnerabilities was documented

in the Vulnerability Assessment document. This assessment included TOE misuse analysis and a password strength of function analysis. The analysis also describes the approach used to identify vulnerabilities of SLES and the results of the findings.

The Vulnerability Assessment was performed and written as part of this evaluation.

4.2 CAPP Overview

The Controlled Access Protection Profile (CAPP) is based on the C2 class of the "Department of Defense Trusted Computer Systems Evaluation Criteria" (DoD 5200.28 – STD) colloquially known as the "Orange Book." CAPP requires that the operating system implement the Discretionary Access Control (DAC) security policy. DAC allows the information owner to control who is allowed to access the information.

The CAPP functional requirements fall in the following five broad categories:

- Identification and Authentication
- User Data Protection
- Security Management
- Protection of the TSF
- Security Audit.

4.2.1 Identification and Authentication

Identification and Authentication include the functionality required to uniquely identify the user.

SLES provides identification and authentication using pluggable authentication modules (PAM) based upon user passwords. Other authentication methods (e.g., Kerberos authentication, token based authentication) that are supported by SLES as pluggable authentication modules are not part of the evaluated configuration. PAM was configured to ensure medium password strength, to ensure password quality to limit the use of the su command, and to restrict root login to specific terminals.

Meeting the CAPP requirements for Identification and Authentication involved changing the default PAM configuration for SLES 8. The new configuration is documented by the Security Guide.

4.2.2 User Data Protection

User Data Protection specifies the functionality that protects data from unauthorized access and modification—the enforcement of the Discretionary Access Control policy. In addition, deleted information must not be accessible and newly created objects must not contain residual information.

The Discretionary Access Control policy restricts access to file system objects based on Access Control Lists (ACLs) that include the standard UNIX® permissions for user, group, and others. Access control mechanisms also protect IPC objects from unauthorized access.

The evaluated configuration used the ACL support in the ext3 file system. The vanilla kernel already clears file system, memory and IPC objects before they can be reused by a process belonging to a different user. Thus, the User Data Protection functionality requirements were already being met by SLES 8.

4.2.3 Security Management

The Security Management class specifies how security attributes, security data and security functions are managed by the TOE. Security Management includes management of groups and roles, separation of capability, and management of audit data.

Management of the security critical parameters of the TOE is performed by administrative users. Commands that require root privileges, such as useradd and groupdel, are used for system management. Security parameters are stored in specific files that are protected by the access control mechanisms of the TOE against unauthorized access by non-administrative users.

Other than the audit data management commands (which are described in the Security Audit section below) all security management functionality was provided by standard functionality already included in SLES 8.

4.2.4 TSF Protection

Protection of the TSF specifies the requirements for maintaining the integrity of the TSF and its data, particularly the protection of configuration data. The TSF will need to perform the appropriate testing to demonstrate the security assumptions about the underlying abstract machine upon which the TSF relies. In addition, the TSF must be demonstrated to be complete and tamperproof.

While in operation, the kernel software and data are protected by the hardware memory protection mechanisms. The memory and process management components of the kernel ensure that user processes cannot access kernel storage or storage belonging to other processes. Non-kernel TSF software and data are protected by DAC and process isolation mechanisms. In the evaluated configuration, the root user owns the directories and files that define the TSF configuration. Files and directories containing internal TSF data (e.g., configuration files, batch job queues) are also protected by DAC permissions.

The TOE and the hardware and firmware components are required to be physically protected from unauthorized access. The system kernel mediates all access to the hardware mechanisms themselves, other than program visible CPU instruction functions.

4.2.5 Abstract Machine Test Utility (AMTU)

To completely fulfill the TSF Protection requirement, we had to produce a tool to test the underlying abstract machine: "The TSF shall run a suite of tests [selection: during initial start-up, periodically during normal operation, or at the request of an authorized administrator] to demonstrate the correct operation of the security assumptions provided by the abstract machine that underlies the TSF."[4] This requirement is sometimes fulfilled by Power-On Self Test (POST) procedures, but given the diversity of platforms that were included in the certification, we decided that a userspace administrative tool, AMTU, would be the simpler approach. AMTU can be run by an administrator at any time and ensures that the hardware enforced security protection is still in effect. To this end, the tool runs a simple check for memory errors, checks for enforcement of memory separation, checks the correct operation of network and disk I/O controllers, and verifies

[4] Controlled Access Protection Profile available from `http://www.radium.csc.mil/tpep/library/protection_profiles/CAPP-1.d.pdf`

that privileged instructions cannot be executed when the hardware is in user mode.

The source code for AMTU is available at `http://www-124.ibm.com/developerworks/projects/amtu`.

4.2.6 Security Audit

Auditing systems collect information about events related to security-relevant activities. Security-relevant activities are defined as those events that are governed by the security policy. The resulting audit records can be examined to determine which security-relevant activity took place and which user is responsible for them. No fully CAPP-compliant audit subsystem was available for Linux, so we implemented this feature to achieve the certification. The audit subsystem developed for the evaluation is called Linux Audit System or LAuS.[5]

LAuS Conceptual Overview The Linux Audit System (LAuS) consists of three primary components: a kernel module responsible for intercepting system calls and recording relevant events, an audit daemon (auditd) that retrieves the records generated by the kernel and writes them to disk, and a number of command line utilities for displaying, querying and archiving the audit trail. See Figure 1.

The interface between kernel and user space uses a character device named /dev/audit. The audit daemon uses I/O Control operations (ioctls) on this device to configure the audit module, and it retrieves audit records from it using the read() system call.

To improve performance, filtering of audit events is performed at the kernel level. Unlike some existing implementations, the audit daemon does not perform any filtering itself. This eliminates a serious performance bottleneck.

The set of filter primitives provided by LAuS is fairly rich, and primitives can be combined using boolean operations. For instance, it is possible to audit open(2) calls made by a setuid application, while ignoring all other open(2) calls, or to restrict auditing to certain files. The eal3-certification RPM contains the evaluated audit configuration files.

At startup, auditd reads its configuration and the set of filter expressions from one or more files, loads the filters to the kernel, and starts auditing.

Auditd then proceeds to listen for audit events generated by the kernel. It retrieves and writes all records directly to disk. Because of the CAPP requirement that audit records must never be lost, this process is more complex than it might seem. auditd constantly monitors disk usage and can be configured to respond in different ways if free disk space drops below certain thresholds. Possible reactions to low disk space include notifying the administrator, suspending all audited processes, or shutting down the system immediately. Both the thresholds and auditd's reactions can be configured by the administrator.

LAuS supports different output modes to provide a flexible way to configure data collection. The simplest approach simply writes the audit trail to a single file in append mode, similar to the way syslogd works.

In "bin mode," audit writes data to a number of fixed sized files (bins), switches to the next file when the current one fills up, and invokes an external command to archive the full bin. Finally, there is a so-called "stream mode" that lets you pipe the audit trail directly into an ex-

[5] The LAuS Design Document is available at `ftp://ftp.suse.com/pub/projects/security/laus/doc/LAuS-Design.pdf`

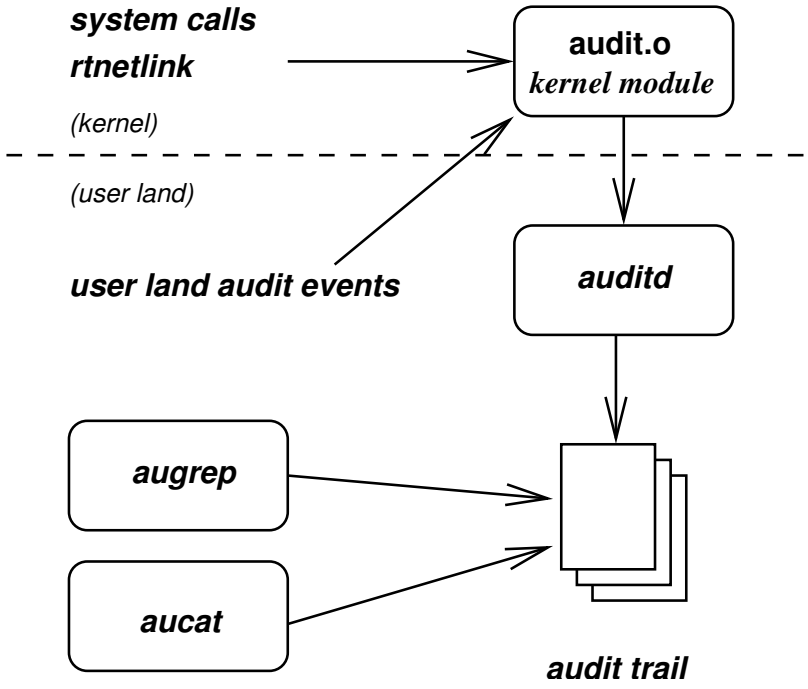

Figure 1: LAuS Conceptual Overview

ternal command; this can be useful if you want to forward the trail to a central storage server.

Auditing can be enabled globally or on a per-process basis; in the latter case, all the child processes are audited as well. The only processes always exempt from auditing are init and the audit daemon itself.

User land utilities were created to parse and read the audit log files. aucat 'cats' the file, transforming all of the audit records to a human readable format. augrep 'greps' the audit records and allows the administrator to selectively review the records. augrep allows the administrator to select audit records based on type, time (range), user, syscall, program (by name or PID) that generated the event, or any combination of these attributes.

Even though the user land utilities are far from trivial, the kernel portion of LAuS proved far more complex; in fact, the kernel portion of the LAuS is a lot more complex than we had initially anticipated. The rest of this section deals with the questions surrounding the audit kernel module.

Additional Design Constraints In addition to making our audit implementation compliant with the CAPP requirements, we had to deal with several constraints which are worth noting.

One was to minimize performance overhead. In the case where auditing was compiled into the kernel, but not configured by the administrator, we wanted it to have zero performance impact if possible. Our kernel developers spent quite a lot of time on additional kernel tuning, making sure the kernel performed and scaled well. Breaking this was not an option.

We also wanted to have a performance overhead as small as possible for the audited case, even though this wasn't as high on our agenda. This definitely took second place to correctness

and CAPP compliance.

A third objective was entirely non-technical, but played a crucial role in choosing an approach to intercepting system calls. We wanted our modifications to the core kernel as small as possible; most of the code should be inside a loadable module.

The rationale behind this was to minimize the probability of introducing bugs (except, of course, bugs in the audit code itself), and to ease maintenance.

The latter point was a fairly important item in the context of the SLES 8 kernel, which includes well above 1,500 additional patches applied on top of the mainline kernel. Updating SLES 8 to a new mainline kernel version was a bit of an adventure, so we wanted to avoid adding audit patches to the kernel that changed lots of files all over the place.

Where to intercept system calls There are basically three ways to intercept system calls on a 2.4 Linux kernel.

The first approach is to create wrappers for those system calls you wish to track, and replace the original function pointers in the system call table with those of the new wrapper functions. This sounds simple enough, and would also satisfy our requirements for zero performance impact in the non-audit case, and a minimally intrusive kernel patch. Unfortunately, this approach doesn't work on all architectures.

The next approach is to add hooks to all kernel functions that must be audited. The major drawback to this approach is that the kernel patch would touch lots of files in the kernel, which we wanted to avoid.

The third approach, which we chose, was to hook into the code path that intercepts system calls for ptrace. This intercept happens very early in the platform-specific assembler code, before the system-call function itself is invoked. The assembly code retrieves a set of flags associated with the calling process, and checks the `PT_TRACESYSbit`. If that bit is set, it jumps to a separate code branch dealing with ptracing. The same test is performed when returning from a system call.

In our audit implementation, we simply defined an additional task flag named PT_AUDITED, and extended the bit test in the system-call entry and exit code to test for both bits at the same time. This gave us system-call intercept with zero performance overhead in the normal, non-audited code path.

See Figure 2 for a picture showing the flow of control when auditing a system call.

Defining which system calls to audit By far, the most important part of auditing concerns system calls. As mentioned above, CAPP requires auditing all security-relevant system calls. We needed to determine which system calls are security relevant and which aren't.

The obvious ones are those that change the state of a process, the file system, or other system resources. These includes calls such as setuid, open, close, and setting the system's host name or clock. An audit implementation also needs to cover less obvious operations, such as binding a socket to a port, attaching shared memory segments, and performing ioctls.

Most system calls are fairly straightforward to handle, and much of the information on system calls and the arguments they take can be encoded statically in tables. Some calls, such as msgrcv, which comes in two versions on the i386 platform for historical reasons, were difficult to handle. 64-bit platforms usually require an additional table as they support a 32-bit sys-

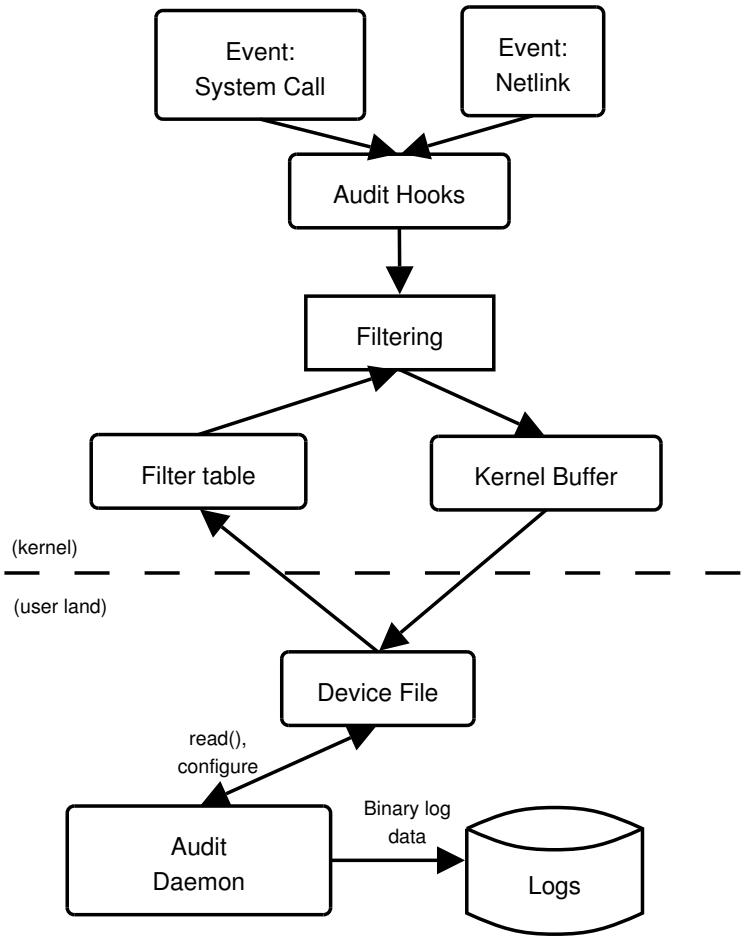

Figure 2: Auditing a System Call

tem call interface in addition to their native 64-bit interface.

However, some of the operations we wanted to audit proved a little more elusive; these were the ioctl system call and network configuration changes.

Auditing ioctls The ioctl system call is the dirty little back alley of UNIX-like operating systems. If a driver for a piece of hardware, a network protocol or a file system needs to expose some driver-specific mechanism or tunable parameters to user-space applications, the most common method for doing so is to define one or more ioctls.

The ioctl system call takes an open file descriptor, which must refer to something controlled by the driver (for example, a terminal, a device file, or a socket), an integer number specifying the request, and an opaque pointer to some chunk of memory. Exactly what to do with this piece of memory depends on the driver that is being talked to, and the integer passed as the request ID.

Unfortunately for us, the Linux kernel supports well over a thousand ioctls, and while many of them are rather obscure, they do change the system's state and are thus subject to auditing. It is obvious that compiling and maintaining a list of 1000 ioctls and their arguments was not an option.

Most ioctl numbers nowadays encode sufficient information on whether the operation passes data into the kernel, retrieves data, or both, and the size of the argument. Therefore, writing audit records for these is straightforward. However, there is still a fair number of ioctls that do not follow this convention.

What is far worse is that ioctl numbers are not unique—frequent users of strace will probably know that the TCGETS ioctl uses the same number as some obscure sound card operation. But this is not the only conflict.

However, the most difficult aspect of auditing ioctls is that it isn't sufficient to simply generate audit records for these calls; you must also be able to display the information of each audit record to the user.

The way we solved this problem was entirely non-technical. Our target of evaluation clearly stated that the super user account remains special. The super user can do everything, from loading unsupported modules not covered by the certification, to disabling the audit subsystem altogether.

Instead of trying to handle each and every ioctl in the audit module, we went through all ioctls available in our to-be-certified configuration and categorized them. The ones we needed to audit were those that were security relevant in some way, but did not require administrative privilege; the list we came up with this way was much smaller than the original list and more manageable.

Auditing network configuration Another aspect that proved to be a challenge was tracking network configuration changes, because only a fraction of those are done through ioctl calls. Most network configuration changes are performed by passing data to a netlink socket. These changes can be audited by simply recording all sendmsg and recvmsg calls on netlink sockets, but that is far from optimal. On the one hand, a send or receive operation on a netlink socket can include more than one request. On the other hand, the outcome of a netlink call is not returned through the system-call return value, but in a separate netlink message generated by the kernel and queued to that socket. Simply logging the raw netlink data sent and received would require quite a bit of built-in intelligence on the part of the user land applications that are supposed to display this data.

So instead, we decided to tap into the netlink layer directly, where a data blob sent to a netlink socket is broken up into separate requests, and each request is processed in return. This allowed us to record each netlink request separately, and place the outcome of the operation into the same audit record as the original request.

The Login User ID An aspect of auditing that is worth mentioning is how to deal with the CAPP requirement that each record identifies the user performing the operation.

The obvious solution (which would be to use the real user ID associated with the calling process) is not sufficient, as setuid applications can change these IDs at will. Tracking all uid changes, and thereby allowing the audit utilities to piece together the original user ID from this mosaic, is not practical either. It is not uncommon for some processes on a dedicated server to run for hundreds of days, so the amount of data to look at would be prohibitive.

The only viable solution in this case is to attach a "login uid" to each process. The login uid remained constant across all other changes of real, effective, and saved user IDs, and was inherited by all child processes. Of course, this required changes to PAM so that this uid would

be set on login.

Nightmare on Audit Street There is one major problem with the approach to system call intercept that we chose and which in hindsight made it a less than optimal choice. The problem is that our approach requires data to be copied twice. To understand why this is a problem, let's look at the open(2) system call, which takes a path name as an argument. This path name is passed into the kernel function as a pointer to a string (essentially a chunk of memory) in the address space of the user process. In order to operate on this string, the kernel must copy it to a buffer in the kernel address space, possibly paging in memory as it goes.

When entering the kernel, the audit module retrieves the path name from user space, and decides whether to create an audit record for this call. If it does decide to create an audit record, it sets up an audit record containing the system call number and a copy of all arguments, including the path.

The system call proceeds as normal, and the kernel function `sys_open` retrieves the path name from user space a second time, and carries out the requested operation based on this data.

The problem is that the memory in user space may have changed in the meantime, so that the record written by the audit module does not correspond to what was actually performed by the operating system.

There are several ways this can happen. Of course, the calling process itself cannot modify this memory, as it is currently executing the system call. However, memory can be shared between processes in a variety of ways. Threads can share the entire address space; processes can attach to the same shared memory segment; memory can be mapped from a file, which can be mapped by other processes as well.

Such an attack on the audit module is not really practical, because proper timing is probably quite hard, and any attempt to perform this attack would most likely leave a trail in the audit file. But even the theoretical possibility of circumventing the audit subsystem is unacceptable in terms of CAPP compliance.

The cases described above can be detected and dealt with by the audit module. Dealing with these problems, however, incurs additional complexity and performance loss (especially in the case of multithreaded applications). Needless to say, the added complexity engendered a considerable number of bugs. For this reason, these additional checks can be turned off by the administrator. These checks are turned off in the evaluated configuration of audit and the associated risk, considered minimal, is documented in the Vulnerability Assessment.

SUSE Linux Server 9 SUSE Linux Server 9 will include an updated version of the LAuS kernel patches. In many respects, the updated LAuS module will work in the same way as the SLES 8 version did, with the major exception being the way system calls are intercepted.

When planning audit for SUSE Linux Server 9, we considered two options.

The first option was a solution we had already looked at for SLES 8 and abandoned, namely adding hooks to all system call functions relevant for CAPP. This is the approach we chose for SUSE Linux Server 9, mainly in order to avoid having to jump through all those extra hoops in an attempt to prevent the race conditions described in the previous section. One pleasant side effect of this approach is that it also eliminated a lot of platform-specific code.

The second option we considered was to add audit support as an LSM module, or extending an existing LSM module such as SELinux. The security framework in the 2.6 kernel goes a long way toward intercepting all security-relevant operations. Adding audit hooks in this place is appealing, because it would mean no additional performance cost (the security hooks do come with a certain performance penalty already) and no additional maintenance problems (because the audit patch would not have to touch multiple kernel files).

The main reason why we did not choose this approach was that the security hooks provide a more abstracted view than we had chosen for LAuS in SLES 8. Security hooks do not correspond directly to system calls, but rather represent the security check necessary to validate whether an operation is permitted. There is a fine distinction between "user X attempted to perform operation Y, and the outcome was Z" and "user X attempted an operation on object A that caused us to perform security check B, and the outcome of this check was C." In particular, we are neither aware of the operation that triggered the security check, nor of its final outcome, because the operation can still fail even if security clearance is given.

Moreover, a single system call may require several security checks, such as renaming a file, where we need permission to remove the file from the source directory and permission to add it to the destination directory.

Changing LAuS to use the security hooks would have meant rewriting much more code than we wanted to, including the filtering code and much of the user-land applications. We also would have had to modify considerable parts of the documentation required for recertification.

Future Directions This is not to say that it is not possible to write an audit implementation leveraging some features of the LSM framework. In fact, we hope to have a common audit implementation in the mainstream kernel one day. It would greatly help acceptance by the kernel community if that solution did not add another set of hooks into many performance-critical functions.

5 Evaluation Roadmap

Performing a security evaluation should never be a one-time accomplishment. To maintain the security level achieved, the security certificate must be maintained. In the case of Linux, the intent is to go a step further: to increase, step-by-step, the assurance level and the security functionality until Linux achieves the highest assurance level of any commercial operating system product, while offering the richest set of security functions. The first step was accomplished in July 2003, when we obtained an EAL2+ evaluation for SUSE SLES 8 as-is. This paper documented the results of the second step, where we obtained a CAPP/EAL3+ certification for SLES 8 SP3 in January 2004. Linux, like its commercial competitors, has now been successfully evaluated for compliance with the requirements of the US government-defined CAPP. As a further step, Linux is currently in evaluation for compliance with the requirements of the EAL4 level. This includes the development of a low-level design of the Linux kernel (the evaluation will be based on the 2.6 version of the kernel) as well as a more sophisticated vulnerability analysis being performed. The experience gathered in the EAL2 and the EAL3 evaluations have given us the confidence that compliance with EAL4 can be achieved in fairly short order.

6 Value of Certification

The value of certification can be considered from two perspectives: business and technical.

In order for Linux to be adopted by the commercial and government markets, it faces stiff competition from entrenched incumbents. All of the incumbent products have been evaluated using the Common Criteria. In addition, the U.S government instituted a national security community policy against procuring unevaluated products (NSTISSP No. 11). There is a high probability that other governments and commercial entities will do the same.

While there is much skepticism surrounding the technical value of certification, certification is very much in line with the "many eyes" philosophy. For commercial products, certification is often the only time the code is reviewed by people outside of the development team. The assurance requirements of Common Criteria add to the number of trained eyes looking at the design and source of a project using defined and rigorous procedures. During the course of the EAL3 evaluation, we found and fixed several bugs, created lots of documentation, and shipped an integrated CAPP-compliant audit system. We noticed an anomaly on the iSeries platform while testing the Abstract Machine Testing Utility. Analysis of this anomaly by the ppc64 development team led to the discovery of a memory separation bug on the iSeries platform.[6] Many PAM module bugs were identified and fixed in SLES 8, including a double free bug in `pam_pwcheck`.[7] Man pages were created for several undocumented system calls, PAM modules and admin utilities, including `io_setup`, `readahead`, `set_thread_area`, `pam_wheel`, `pam_securetty`, and others.

[6] Paul Mackerras fix to "Make kernel RAM user-inaccessible on iSeries" `http://www.kernel.org/diff/diffview.cgi?file=/pub/linux/kernel/v2.4/patch-2.4.23.bz2;z=290`

[7] `http://www.atsec.com/01/index.php?id=03-0002-01&news=28` Patches are available from klaus@atsec.com

7 Conclusion

Achieving the EAL2 and EAL3/CAPP certifications was significant because it proved that Linux is indeed certifiable. The certification opened the market up to include U.S. government agencies and commercial entities that require certification. Future evaluations of Linux distributions can be made easier by Linux adoption of a CAPP-compliant audit subsystem.

8 Legal Statement

Proceedings of the Linux Symposium

Volume Two

July 21st–24th, 2004
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Review Committee

Jes Sorensen, *Wild Open Source, Inc.*
Matt Domsch, *Dell*
Gerrit Huizenga, *IBM*
Matthew Wilcox, *Hewlett-Packard*
Dirk Hohndel, *Intel*
Val Henson, *Sun Microsystems*
Jamal Hadi Salimi, *Znyx*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*