

New Security Features in OpenSolaris and Beyond

Jan Pechanec
Sun Microsystems, Inc.
Jan.Pechanec@Sun.COM

Christoph Schuba
Sun Microsystems, Inc.
Christoph.Schuba@Sun.COM

Mark Phalan
Sun Microsystems, Inc.
Mark.Phalan@Sun.COM

1 Abstract

This paper examines several new security features and enhancements to existing security features that were introduced into the OpenSolaris Operating Environment in the time period of approximately mid 2006 through mid 2008. We focus on the following contributions, rather than present an exhaustive list: Solaris Trusted Extensions (the multi-level security features that is now an integral part of the Solaris architecture), the Key Management Framework (KMF - a unified set of interfaces for managing PKI objects), the OpenSSL PKCS#11 engine, and a number of functional enhancements to our Kerberos system.

Furthermore, we present work in progress on filesystem encryption (most notably ZFS encryption and the loopback file system encryption), PKCS#11 engine, SunSSH, and Kerberos, new security features that, as of mid 2008, are being actively developed and are scheduled to become part of future OpenSolaris versions and distributions

2 Overview of the OpenSolaris project

The OpenSolaris project[1] is an open source project sponsored by Sun Microsystems, Inc, that was initially based on a subset of the source code for the Solaris Operating System. It is a nexus for a community development effort where developers from Sun and elsewhere can collaborate on developing and improving operating system technology. The OpenSolaris source code will find a variety of uses, including being the basis for future versions of the Solaris OS product, other operating system projects, and third-party products and distributions.

Since its opening day launch in June 14, 2005, in which the bulk of the Solaris system code was released, a growing community of OpenSolaris users and contributors has emerged. There are already several distributions based on OpenSolaris code, including but not limited to SchilliX, Belenix, Nexenta, and Sun's official distribution called Opensolaris.

3 Overview

All the security features and enhancements mentioned in this paper were or are being developed internally at Sun. They will be or have been first introduced as a part of the Nevada distribution (also known as Solaris Express Community Edition or SXCE[2]). The code is available externally very shortly after it has been committed to the Nevada code base. Also, it is released as part of Sun's official OpenSolaris distribution, also known as project Indiana[3].

In this paper, we will examine two filesystem encryption options which are under development – lofi and ZFS crypto. Next the Trusted Extensions project will be introduced, which is a reimplementation of Trusted Solaris 8 based on new security features in Solaris 10. The Key Management Framework (KMF) is a unified set of interfaces (both programming API and administrative tools) for managing Public Key Infrastructure (PKI) objects in Solaris, and SunSSH is an example consumer that uses this API for implementation of X.509 authentication in the SSH2 protocol. The OpenSSL PKCS#11 engine is a way to access the Solaris Cryptographic Framework from OpenSSL. There is a project in progress to add new mechanisms to the engine, and the SunSSH project uses those enhancements to off-load cryptographic operations to cryptographic hardware providers, if available, and thus significantly speed up any bulk transfer of data. Validated execution is a way to verify the integrity of program and library objects at the time of execution. Finally, we examine some of the new features recently added to Kerberos or in development as of June 2008. **For features under development as of June 2008, the details are subject**

to change as work progresses. This document is not a feature commitment from Sun Microsystems for a future releases of Solaris.

4 Solaris Trusted Extensions

4.1 Overview

The Solaris Trusted Extensions project[6] has moved security technologies that constituted the difference between Solaris 8 and Trusted Solaris 8 into the base of Solaris 10 and OpenSolaris. In the process, most of these technologies have been expanded significantly in functionality, usability, and robustness. The Trusted Solaris products has been renamed because it is delivered as software that is part of the regular Solaris distributions that can be optionally enabled and configured.

4.2 How it works

Solaris Trusted Extensions is an optionally-enabled layer of secure labeling technology that allows data security policies to be separated from data ownership. Integration to OpenSolaris (and Solaris 10) allows the system to support both traditional Discretionary Access Control (DAC) policies based on ownership, as well as label-based Mandatory Access Control (MAC) policies.

Different objects of the system, including files, packets, devices, window management services or printers are assigned labels or range of labels. Those labels establish explicit relationships between objects. Authorization allows applications and users to read or write to the objects. When two labels are compared, the first label can be greater than, less than, equal to, or disjoint from the second label. Labels consist of hierarchical components called classifications (or levels) and a non-hierarchical components called compartments (or categories). Classifications are compared as integers, and compartments are compared as bit masks. Labels are disjoint when each contains at least one compartment bit which is not present in the other.

Users interact with labels as strings. Graphical user interfaces and command line interfaces present these strings. Human readable labels are classified at the label that they represent. Thus the string for a label A is only readable (viewable, translatable to or from human readable to an opaque label data type) by a subject whose label allows read (view) access to that label.

Applications do not need to be modified nor profiled to bring them into conformance with the MAC policy. Instead, the entire application environment is virtualized for each label through the use of Solaris Containers (zones), Solaris primary OS-level virtualization technology. The Solaris Containers facility provides an isolated environment for running applications. Processes running in a zone are prevented from monitoring or interfering with other activity in the system. Access to other processes, network interfaces, file systems, devices, and inter-process communication facilities are restricted to prevent interaction between processes in different zones. At the same time, each zone has access to its own network stack and name space, enabling per-zone network security enforcement, such as firewalling or IPsec.

All the zones are centrally administered from a special, protected global zone which manages the Trusted Computing Base (TCB) known as the Trusted Path. The zones share a single Lightweight Directory Access Protocol (LDAP) directory in which network-wide policy is defined, as well as a single name service cache daemon for synchronizing local databases. All labeling policy and account management is done from within the Trusted Path. MAC policy enforcement is automatic in labeled zones and applies to all their processes, even those running as root. Access to the Global Zone (and hence Trusted Path applications) is restricted to administrative roles.

Each zone is assigned a unique sensitivity label and can be customized with its own set of file systems and network resources. Each mounted file system is automatically labeled by the kernel when it is mounted. The file system label is derived from the label of the zone or host which is sharing it. All files and directories within the mounted file system have the same label as their mount point. No extensions to the file system structure is required. Processes are uniquely labeled according to the zone in which they are executing. All processes within a zone (and their descendants) must have the same label, and are completely isolated from processes in other zones.

4.3 Multilevel Desktop Sessions

Users can log in via the Trusted Path and can be authorized to select their multilevel desktop preference (Common Desktop Environment or Java Desktop System). Once authenticated they are presented with an option to select an explicit label or a range of labels within their clearance and the label range of their workstation or desktop unit. The window system initiates a user session in the zone whose label corresponds to the user's default or minimum label.

Attempts to cut and paste data, or drag and drop files between clients running in different zones are mediated by the Trusted Path. Specific authorizations are required for upgrading or downgrading selections and files, and are prohibited by default.

5 Filesystem Encryption

5.1 lofi driver encryption

5.1.1 Overview

The lofi file driver[4] exports a file as a block device. Reads and writes to the block device are translated to reads and writes on the underlying file. This is useful when the file contains a file system image. Exporting it as a block device through the lofi file driver allows normal system utilities to operate on the image through the block device (like `fstyp1M`), `fsck(1M)`, and `mount(1M)`. This functionality was originally created for mounting ISO CD images.

5.1.2 How it works

The lofi driver will gain the ability to encrypt/decrypt the raw blocks. The administration command `lofiadm` is extended to support requesting encryption and setting the key.

It is not desirable for lofi to have the encrypted data take up more space than the clear text data would. For this reason it is not possible to use a separate digest and HMAC algorithm as is commonly done in network protocols, for example AES for encryption and HMAC-SHA1 for integrity protection of the cipher text. Encrypted block device support on other platforms does not provide any integrity protection support. To ensure we have integrity protection in lofi a future case will introduce a cipher suite that provides a non-expanding ciphertext output that has integrity protection built in.

No algorithm or key data is written to disk, however some metadata space is reserved at the start of the file. That space is currently used to store a version number only. However, in future versions this space will most likely be used to store other types of data. If an incorrect key value is given then the `lofiadm` mapping will still succeed but any filesystem layered on it will fail to mount (since the data on the device will not appear to be a filesystem).

5.1.3 Example

```
# mkfile 35m /export/home/test
# lofiadm -a -c aes-256-cbc /home/secrets
Enter key: xxx
Re-enter key: xxx
/dev/lofi/1
# newfs /dev/rlofi/1
...
# lofiadm
Block Device      File              Options
/dev/lofi/1       /home/secrets    Encrypted
```

5.2 ZFS encryption

5.2.1 Overview

The first phase of the ZFS encryption project[5] covers the addition of encryption and decryption to the ZFS IO pipeline and key management for ZFS datasets and ZFS pools.

As of June 2008 the phase one code development is complete and under code review. While code integrations into OpenSolaris is planned for later in the year, we cannot give any committed dates at this time. This is the second filesystem encryption project for OpenSolaris under development as of June 2008, see Subsection 5.1 on lofi driver encryption above. However, the lofi project has important limitations -- no data integrity protection, for example. The ZFS encryption project aims at seamless integration of dataset encryption/decryption and key management in the frame of ZFS technology[13].

5.2.2 Goals of the project

The project has several phases, with the first one, as of June 2008, is development complete. The project will provide per file system encryption with randomly generated keys. Those keys are stored on disk with the dataset in wrapped form. It is wrapped by a master per pool key or per dataset key that the user or the administrator provides. While other filesystem encryption technologies often offer per file or per directory encryption, ZFS datasets (datasets are either file systems or ZVOLs) are relatively very cheap compared with traditional filesystems so using the dataset in ZFS provides equivalent functionality. All data and file system metadata, such as file owner Access Control List (ACL) size etc, must be encrypted when on disk. For ZVOLs all data is encrypted on disk; encryption support for ZVOLs allows, for example, encrypted swap areas and databases which directly use raw devices. For encryption, CCM mode of operation was chosen because it provides data integrity, too. The algorithm mode and key length are fixed at dataset creation time. There is no direct use of asymmetric cryptography in the file system. A software only solution will be provided but whenever appropriately supported hardware is present, this hardware can be used, for example, to store the master key or to accelerate cryptographic operations. The SCA-6000 is an example of a device where the master key can be secured in a hardware token. The features we have just mentioned are all part of the first phase of the project.

Future project phases are likely to include features such as support for an encrypted root filesystem, remote key manager (here, asymmetric cryptography might be used in remote key manager protocols), and secure deletion.

5.2.3 Example usage

The following example shows how to set the master per pool key via a passphrase for a ZFS pool. An encryption key is generated from the user supplied passphrase. Raw key data can also be supplied via a file, smart card, or from standard input.

```
# zpool set -o keysource=passphrase,prompt encryption=aes-256-ccm tank
Enter passphrase for 'tank': ****
Enter again: ****
```

After creation before any encrypted dataset can be mounted, the pool key must be loaded:

```
# zpool key -l tank
Enter passphrase for 'tank': ****
Enter again: ****
```

The key status of the pool can be displayed as follows:

```
# zpool get keystatus tank
NAME  PROPERTY  VALUE           SOURCE
tank  keystatus available     local
```

6 Key Management Framework (KMF)

6.1 Overview

The goal of the project[7] was to provide a unified set of interfaces (both programming APIs and administrative tools) for managing PKI objects in Solaris. Before KMF, there were several different "keystore systems" that developers and administrators could choose from when designing systems that employed PKI technologies – Network Security Services (NSS), OpenSSL, and, PKCS#11 were the three main choices for Solaris users. Each of these systems presents very different programming APIs and administrative tools and none of them has any sort of concept of a PKI policy enforcement system.

Not having a unified infrastructure for PKI-enabled applications means that every application must choose between the existing systems, which makes interoperability more difficult and limits the use of the application. KMF provides utilities and an API for managing public key objects in a format-neutral manner and bridges the gap between the existing PKI technologies currently used in Solaris. It also introduces new features not available in existing technologies, for example policy enforcement.

6.2 How it works

KMF provides a consistent management interface through the `pktool` utility. `pktool` was originally developed to manage PKCS#11 keystores, the KMF project enhanced it so that it could serve as a generic PKI tool. This allows an administrator to use `pktool` to administer all three keystore systems, whereas previously, he would have had to use `openssl` command to work with OpenSSL key files, `certutil` to work with NSS databases, and `pktool` to work with PKCS#11 keystores.

From the developer's perspective, the KMF API hides the details of the underlying keystores, and provides information necessary to identify those keystores and stored objects. For example, for OpenSSL files, a filename must be provided, but the developer can use a single KMF API call to access the keystore in a format-neutral manner. The KMF API is also pluggable so that third party vendors can introduce, for example, proprietary or legacy implementations while benefiting from KMF independence. For certificate validation[14], Online Certificate Status Protocol (OCSP), and Certificate Revocation List (CRL) checking is provided, with Public Key Infrastructure (PKIX) path validation planned for future version of OpenSolaris.

KMF API operations include all the commonly needed routines to work with keys, certificates, and Certificate Signing Requests (CSR), including but not limited to creating, deleting, searching, importing, and exporting such objects. All common cryptographic operations are provided, including signing, verifying, encrypting, and decrypting using keys and certificates. The KMF API also provides access to complex objects like X.509 attributes and extensions in human readable formats.

Security policy is defined in a system wide configuration file in XML format, and a new `kmfcfg` tool was introduced to administer that file. Any application can use a different policy file if it chooses to. KMF policy is a set of parameters that controls the use of X.509 certificates by an application. The application must use the KMF API to make use of the policy system.

Any applications that work with certificates are potential consumers of KMF, and we will examine SunSSH as one of them in the next paragraph. One enhancement of KMF in the design phase as of June 2008, is certificate to name mapping. This pluggable mapping system implements a policy which controls which username or hostname the particular certificate should be mapped to.

6.3 X.509 support for SunSSH

6.3.1 Overview

Using X.509 certificates[14] for SSH authentication is already included in the SSH2 protocol as part of `pubkey` authentication but not fully defined. Having the option to use X.509 certificates means here that there is no need to verify public keys using a secondary information channel or blindly accepting those keys during the first session within which a man-in-the-middle attack can be mounted. There are several ways of implementing it – for example, using NSS, OpenSSL or the PKCS#11 API. The last one would

also allow us to use hardware keystores, and sign data without exposing private keys from the tokens. Another approach, as already suggested, would be to use the public KMF API¹.

There are two different IETF drafts on X.509 authentication in the SSH2 protocol. One documents a desirable way of using such an authentication method[15] while the second, which is an informational draft only, documents the way it is implemented in existing SSH implementations. The informational one is the draft that we must follow for compatibility with SSH implementations that already supports X.509 authentication. To our best knowledge, no SSH implementation in production supports [15].

KMF is used for X.509 authentication only. The existing manner of working with public and private keys in files is unchanged. The existing `HostKey` and `IdentityFile` options keywords are overloaded so that a PKCS#11 URI² can be used. The URI specifies a private key in the token. A Solaris soft token keystore can be used as well, making easy to try it out. An example of this URI when used with the `IdentityFile` option follows:

```
-o IdentityFile="pkcs11:token=Sun Software PKCS#11 softtoken;object=user"
```

6.3.2 The use of KMF policy in SunSSH

One goal when adding KMF support to SunSSH[17] was to leverage the existing policy file configuration method rather than adding extra options to SunSSH. Note that a user can always use a specific policy file which will define the policy on the user's side of the connection. Policy files should never be edited manually, `kmfcfg` tool should be always used. The main thing we need the policy file for is to define a Trusted Anchor (TA), i.e. the certificate of a certification authority. There is no separate option to define a different TA that SunSSH should use. So, the only two new options SunSSH needs is the policy file name and a policy name within that file.

7 OpenSSL PKCS#11 engine

7.1 Overview

OpenSSL provides an ENGINE API[10], which allows the creation, manipulation, and use of cryptographic modules in the form of ENGINE objects. These objects act as containers for implementations of cryptographic algorithms. With the standard OpenSSL distribution, users can write their own engine implementations and use the dynamic loading capability of OpenSSL to load and use those engines on the fly. Depending on engine support, various operations including but not limited to RSA, DSA, DH, symmetric ciphers, digests, and random number generation can be offloaded to the engine. For operations that are not available in the engine, native OpenSSL code is used³.

OpenSSL in OpenSolaris is shipped with an internally (in Sun) developed PKCS#11 engine which enables cryptographic operations to be offloaded to the Solaris Cryptographic Framework (CF)[18]. Due to import restrictions in some countries, OpenSSL in Solaris doesn't support dynamic engine loading, and no other engines are shipped as any hardware cryptographic provider should be plugged into the CF and used through the PKCS#11 engine. The main reason the PKCS#11 engine was developed was to offload RSA/DSA operations to the ncp cryptographic provider on the Niagara-1 platform when using the Apache web server. However, with introduction of the n2cp driver for Niagara-2 which is capable of hardware accelerated symmetric cryptographic and digest mechanisms, there are many more applications now that could make use of the PKCS#11 engine.

¹ as of June 2008, the new KMF API is available in OpenSolaris only. The one present in Solaris 10 is a different private API and should not be used by third party applications.

² format of the PKCS#11 URI is not standardized yet so the current format used is subject to change

³ during the initialization of the engine, a set of mechanisms that the engine supports is returned to OpenSSL through the engine initialization function. OpenSSL then offloads only mechanisms that are part of the set.

7.2 The current version of the engine

As of June 2008, the version of the engine consumes the PKCS#11 API from the `libpkcs11` library. This library provides a special slot⁴ called the metaslot. The metaslot provides a virtual union of capabilities of all other slots. When available, the metaslot is always the first slot provided by `libpkcs11`. The problem with this approach is that all mechanisms for which hardware acceleration is not present on the machine used are offloaded to the softtoken, which is a software implementation of a PKCS#11 token. The inherent overhead of the CF then causes operations performed by softtoken to be performed more slowly than the native OpenSSL code. One possible solution may be to use the `pkcs11_kernel` library directly⁵. This library provides the same PKCS#11 API but offers only those slots that have hardware capabilities. Such a solution is needed if we want applications to use the engine by default where no regression is acceptable on platforms without hardware acceleration for cryptographic operations. Using the `pkcs11_kernel` library directly would mean no speed regression on machines without supported cryptographic hardware accelerators as the native OpenSSL cryptographic operations will be used.

Another limitation of the current engine implementation is that it doesn't offer many mechanisms:

```
$ openssl engine -vvv -t -c
(pkcs11) PKCS #11 engine support
[RSA, DSA, DH, RAND, DES-CBC, DES-EDE3-CBC, AES-128-CBC, RC4, MD5, SHA1]
```

7.3 New code in progress in the engine

Given the current cryptographic accelerator hardware that is available for and integrated into Sun machines, the decision was made to operate with two provider slots. The first slot is used for the most appropriate RSA/DSA/DH/RAND instance, and the second slot is used for symmetric cryptographic algorithms and digests. The issue that needed to be solved to implement this change in the engine code was that the ncp hardware provider presents key usage limitations for RSA (2048 bits), DSA (1024) and DH (2048). For all operations that use longer keys we must call back into the native OpenSSL code. Note that with the `libpkcs11` library and the metaslot no code changes were required since operations using such keys were automatically offloaded to the `pkcs11_softtoken` library. When the project integrates to OpenSolaris, applications using OpenSSL will be able to load the PKCS#11 engine and transparently experience significant performance improvements when an appropriate hardware accelerators is present in the system, while avoiding the overhead of the Cryptographic Framework for mechanisms for which no hardware accelerator is present.

The current engine prototype on Niagara2 now shows these mechanisms:

```
$ openssl engine -vvv -t -c
(pkcs11) PKCS #11 engine support
[RSA, DSA, DH, DES-CBC, DES-EDE3-CBC, DES-ECB, DES-EDE3, RC4, AES-128-CBC,
AES-192-CBC, AES-256-CBC, AES-128-ECB, AES-192-ECB, AES-256-ECB,
AES-128-CTR, AES-192-CTR, AES-256-CTR, MD5, SHA1, SHA256]
```

7.4 SunSSH with OpenSSL PKCS#11 engine support

7.4.1 Defining the problem

SunSSH is a good example of an application that could make use of hardware acceleration for symmetric cryptographic algorithms. Recently, we have been receiving many requests for this enhancement which reflects the number of Niagara T2 machines sold. Sun servers that are based on the Niagara T2 CPU chipsets and configured with many virtual CPU's present an ideal computing platform for multithreaded applications or workloads where many requests need to be served in parallel. However, applications that are CPU intensive and run in a single thread exhibits apparent slow-down in comparison to common 1-2 CPU machines. The problem is that all the CPU intensive part is now done on one virtual CPU which is slower in comparison to CPU's shipped with other workstations with one or a few processors.

⁴ for a definition of a *slot*, please see the PKCS#11 standard[12]

⁵ that is what `libpkcs11` library itself uses

7.4.2 Hardware cryptographic support

To remedy the situation, the n2cp hardware provider together with SunSSH using the PKCS#11 engine presents an attractive combination with one condition for successful use of the engine. The data blocks to process must be of a reasonable size. There is a significant overhead present when data is offloaded to the hardware, so offloading data blocks of, say 128 bytes, would actually slow down the transfer significantly. However, with bulk data transfer⁶, that's not the case and with current version of SunSSH blocks of around 8KB are processed. Very small SSH packets during interactive sessions are not of concern here, because human beings are not capable of typing fast enough to notice the slow down caused by offloading small data blocks to the hardware.

The changes needed for SunSSH to use the PKCS#11 engine are not small. This is in part due to how privilege separation is done in SunSSH. SunSSH forks after user authentication, with the parent becoming a privileged monitor responsible for allocating pty's, auditing logins and logouts, working with utmpx/wtmpx databases, and using the host's private keys and Generic Security Services API (GSS-API) credentials. The child drops privileges and continues to process the packets, including encryption and decryption. The problem with using the engine is that the child process needs to use the cryptographic contexts after the parent process has forked.

7.4.3 Using the PKCS#11 engine from SunSSH

The PKCS#11 standard[12] forbids the use of existing PKCS#11 sessions after a fork which means that all contexts must be cleaned up, the engine closed and reinitialized. However, encrypted data sent in one direction is considered a single data stream, with initialization vectors passed from the end of one packet to the beginning of the next packet. Reinitializing the engine without other actions means that we loose the context and can't properly process the packets any more.

There are several solutions to the situation:

- the monitor process could be pre-forked before setting up the first cryptographic contexts
- a new key re-exchange process could be initialized, finish the engine and clear all contexts, fork, initialize the engine again and set up all contexts with the new keys after the fork,
- the child could assume a role of the monitor while the parent would continue using the current engine session.

From those three options, the third option seems to be the simplest to implement. The problem with it is that we would be changing the way an unprivileged child and monitor currently work together. While this is private to SunSSH, it is assumed that there are users that might depend on the current behavior. Debugging is an example of this. Clearly, this isn't the best way to go.

For the server, the first option was chosen, to avoid the second rekeying shortly after the first one for every SSH connection. Also, let's not forget that the SSH1 protocol doesn't support rekeying at all. Given the fact that the monitor then doesn't know all the needed information, for example that the user authenticated. A new privilege separation message was introduced and the privilege separation code was rearranged.

For the client, usage of `-f` option and `~&` escape sequence must have been resolved since forking of the ssh client was involved. To solve that, the second key re-exchange right before daemonizing was used. Since the SSH protocol 1 is used mostly by old implementations and probably older appliances, we think that there is no problem not to offer hardware acceleration on client side for SSH protocol 1.

7.4.4 Results using the SunSSH prototype

Every SSH packet is processed twice. Slightly simplified, a HMAC is computed over the plain data, the data is encrypted, and the HMAC is appended. We got significant gains from offloading symmetric cryptographic, and not much for offloading digest operations. With the prototype we have (subject to change) with the default AES-128-CTR encryption mode, we see roughly a 2.5x speedup on the Niagara T2 platform when the engine is used. We also provide a new `UseOpenSSL` option that might be used to switch off the default loading of the engine, in case of a faulty driver of the 3rd party accelerator, for example.

⁶ the speedup gained covers the SCP and SFTP protocols as well since those protocols work above the SSH protocol and rely on the SSH protocol to securely transfer the data they need

Note that we do not get much speed up from offloading RSA/DSA to the engine. This is a direct consequence of how SSH implementations are commonly used. In contrast to web servers, the speed up we ask for is for bulk data transfer, i.e. for sessions that last some time. Usually, machines do not accept hundreds of SSH connections in parallel, and RSA/DSA operations are used only during the initialization and key re-exchange (which is every 1-4GB of data by default, depending on the algorithm used).

8 Validated Execution

The Validated Execution project[9] provides a way to verify the integrity of program and library objects at the time of execution. Such verification provides assurance that the executable has not been altered, either accidentally or deliberately, since it was released by its publisher (which may be Sun, a third-party ISV, or the end customer).

There have been other attempts to solve this problem, such as Tripwire[22] or Basic Audit Reporting Tool (BART)[23], by comparing files against a manifest of known valid files. However, there is no protection against programs that are modified after the comparison but before they are executed. Performing the verification at the time of use avoids the race condition inherent in solutions that periodically compare files against a known reference.

Each executable file object is protected by a digital signature, which is constructed from a one-way hash of the file contents encrypted with a private RSA key known only to the publisher of the software. The corresponding public key can be used to decrypt the hash value and verify that it matches the file contents. Most ELF files in Solaris already contain an embedded digital signature; for those that do not and for non-ELF objects such as shell scripts, we provide a separate manifest listing the signatures for each file. This mechanism extends easily to allow third-party software vendors and end customers to create their own manifests of file signatures.

The general strategy employed is for entities that cause code to be loaded and executed, such as the kernel and run-time loader, to verify such code before execution. This approach establishes a chain of trust where each code component is validated by a previously validated component. To validate the earliest executing code in the system and initialize this process, we rely on a hardware component called a Trusted Platform Module (TPM). For systems without a TPM, all of the functionality described herein is still available, but the user must implicitly trust that the earliest code has not been modified.

This feature provides some degree of protection even against an attacker who is able to run processes with all privileges. We cannot protect in general against such a process, because it is able to alter any part of the file system or memory. However, the privileged process cannot alter the information stored in the TPM without knowledge of the TPM's owner password. Even if the running system is compromised, any alteration to the file system will be detected the next time the system is booted. Any executable files modified by the attacker will fail to start when the system reboots. Therefore, if an attacker gains privilege by exploiting a vulnerability, he must do so every time the system is booted rather than leaving behind a back door that can be used in the future to gain privileged access to the system.

9 Kerberos

Kerberos[19] is a computer network authentication protocol, and the Solaris implementation[21] is based on the MIT Kerberos implementation. We tightly cooperate with MIT on the development.

9.1 Pluggable Kerberos DB backends and LDAP

The Kerberos database of principal and policy information is traditionally stored in a db2 file-based database. LDAP support was recently added allowing for the principal and policy records to be stored in a directory server instead of the traditional db2 store. A new plugin interface was created to allow future backends to be implemented with little or no impact on the existing code. Both the new LDAP support and the older db2 support are now implemented as plugins. The db2 plugin is still the default. There are a number of advantages to using the LDAP plugin including simplified administration. The LDAP plugin opens the possibility of having multi-master Key Distribution Centers (KDC).

9.2 Zero-Conf Kerberos clients

In older versions of Kerberos, clients had to be configured in order to operate in a Kerberos realm. At a minimum a default realm, host-to-realm mapping, and realm-to-kdc mapping had to be provided. The default configuration file was shipped in an essentially misconfigured state so it could not be used without modification. A number of small changes were made to remedy this situation:

- by default Kerberos clients will look up DNS to locate a KDC for a given realm. This is equivalent to having the "dns_lookup_kdc" option set to "true".
- the Kerberos realm for a given host (or the local hostname for the default realm) is determined by looking at the DNS domain name. If a KDC can't be found for a realm corresponding to the DNS domain name (in caps) then a component of the domain name is dropped and the search begins again. This procedure is repeated until there are only two components left in the potential realm name or a KDC is found.
- krb5.conf has been modified so that it isn't mis-configured out of the box. If certain conditions are true then krb5.conf may need no modifications for a working client.
- Limited client-side referral support. By default now all Ticket Granting Service (TGS) requests ask for tickets with the referral bit turned on. This allows for the KDC to inform the client what realm the host is in. Microsoft's Active Directory product supports referrals.

The result of these changes is that in some suitably configured environments (KDC information in DNS, realm names matching DNS names, DNS configured on the client) no further configuration is necessary.

9.3 Public Key Cryptography for Initial Authentication in Kerberos (PKINIT)

Traditionally a KDC will supply a Ticket Granting Ticket (TGT) to anybody who asks for one. The TGT is encrypted with the principal's long term key. As only the KDC and the user know the user's secret key only that user can use the TGT. The key used to encrypt the TGT is a symmetric key. In order to reduce the possibility of an off-line brute-force attack on the user's long term key a pre-authentication scheme is used. The default pre-authentication scheme uses a timestamp. For PKINIT[20] a new pre-authentication scheme is introduced whereby the client sends the KDC pre-authentication information containing the public key of the client. The KDC then encrypts the reply with either a symmetric key (signed by KDC, encrypted with public key of client) or performs a Diffie-Hellman exchange to establish a shared key which is then used to encrypt the reply. The MIT Kerberos implementation relies on OpenSSL for certificate processing. The plan for OpenSolaris is that KMF will be used. The PKINIT functionality is implemented via a plugin. New infrastructure was added so that pre-authentication methods may be implemented as plugins. A pam module using PKINIT to authenticate to a KDC would allow for smart-card based authentication. There are plans to implement this but as a follow-on project and not part of the initial integration.

9.4 Kerberos master-key encryption type migration

The Kerberos principal and policy information is encrypted with a key generated at the same time the Kerberos database is initialized. This key cannot be changed once set. By default a single DES key is used. In order to decrypt the principal or policy information a password is provided to the KDC on startup - either interactively or via a "stash" file. The project aims to remove this limitation and allow for the key to be changed and stronger encryption types be used (e.g., AES) without re-creating the Kerberos database. Instead of storing the secret information in a stash file a keytab will be used. This approach provides for key versioning among other things. An interesting side-note to this project: This work is being done by Sun but instead of integrating first into Solaris and contributing those changes back to MIT, we're working directly with MIT and the first commit of the code changes or contributions will be to the MIT codebase. Recently, MIT set up a new Kerberos Consortium which allows interested parties to have more influence on the direction that Kerberos is taking. Sun is a part of that consortium.

9.5 Improvements to kclient

kclient is a utility used to simplify Kerberos client configuration. It aims to make client configuration simpler and less error prone. It can be run in interactive or non-interactive modes. Recently kclient was

enhanced to support more client configuration scenarios. Significant additions include Microsoft AD support - kclient can now perform a Active Directory join. It also now supports non-Solaris KDCs such as MIT, Heimdal, and Shishi. Unfortunately there is no standardized Kerberos administration protocol - all Kerberos implementations use their own which precludes interoperability. kclient is a part of the solution. Other new features of kclient include better support for dynamic clients and clusters.

10 Conclusions

This paper presented security technologies that are or are going to be part of the Solaris and OpenSolaris operating systems. Some of these technologies are enhancements to existing security features, some are new security features, and some are part of ongoing projects. The paper focuses on these technologies in the timeframe of mid 2006 through mid 2008. With technologies such as multi-level security included in the operating system and future use of validated execution, the OpenSolaris operating system continues to be a leading contributor of technology innovation.

11 References

- [1] OpenSolaris project, <http://www.opensolaris.org>
- [2] Solaris Express Community Edition, <http://www.opensolaris.org/os/downloads/>
- [3] Project Indiana, <http://opensolaris.org/os/project/indiana/>
- [4] lofi(7D) driver, <http://www.opensolaris.org/os/project/loficc>
- [5] ZFS Crypto, <http://www.opensolaris.org/os/project/zfs-crypto>
- [6] Trusted Extensions, <http://www.opensolaris.org/os/community/security/projects/tx/>
- [7] Key Management Framework, <http://www.opensolaris.org/os/project/kmf/>
- [8] <http://www.opensolaris.org/os/community/security/projects/SSH/ssh-x509v3-design.html>
- [9] Validated Execution project, <http://www.opensolaris.org/os/project/valex>
- [10] engine(3), manual page for ENGINE cryptographic module support
- [11] n2cp(7D), manual page for Ultra-SPARC T2 cryptographic provider device driver
- [12] PKCS#11 standard, version 2.20, <http://www.rsa.com/rsalabs/pkcs/>
- [13] ZFS, <http://www.opensolaris.org/os/community/zfs/>
- [14] RFC 3280: Internet X.509 Public Key Infrastructure, Certificate and Certificate Revocation List (CRL) Profile
- [15] X.509 authentication in SSH, `draft-ietf-secsh-x509-03.txt`
- [16] X.509 authentication in SSH, `draft-saarenmaa-ssh-x509-00.txt`
- [17] SunSSH project, <http://www.opensolaris.org/os/community/security/projects/SSH>
- [18] Solaris Cryptographic Framework, <http://opensolaris.org/os/project/crypto/>
- [19] RFC 4120: The Kerberos Network Authentication Service (V5)
- [20] RFC 4556: Public Key Cryptography for Initial Authentication in Kerberos
- [21] Kerberos project, <http://www.opensolaris.org/os/project/kerberos>
- [22] Tripwire Enterprise, <http://www.tripwire.com/products/enterprise/index.cfm>
- [23] bart(1M), manual page for Basic Audit Reporting Tool