Mobile Simplified Security Framework

Dmitry Kasatkin
Nokia Corporation
`dmitry.kasatkin@nokia.com`

Abstract

Linux kernel has already several security frameworks such SELinux, AppArmor, Tomoyo and Smack. After some studies we found out that they are not very suitable for mobile consumer devices such as mobile phones. They either require too complicated administration or do not really provide any security API, which can be used by applications providing services to verify credentials of their clients, and then decide if a particular client can access the provided service or not.

In this paper we present a new platform security framework developed by the Maemo security team specifically for mobile devices. The key subsystem of the Mobile Simplified Security Framework is the Access Control framework, which is used to bind privileges (resource tokens) to the application when the application is starting. Using a special API, different entities are able to verify possession of those resource tokens and allow/disallow access to protected resources. If any of the applications require an access to protected resources, a Manifest file with the credential request should be included in the package providing the application. The Manifest file is also used to declare new credentials, which are provided by an application coming from the package.

1 Introduction

The scope of this paper is to describe Mobile Simplified Security Framework (MSSF). This security framework has been developed specifically to be suitable for mobile consumer electronics devices such as smartphones.

Smartphones usually have limited resources such as memory, CPU power, and power supply, but at the same time have a network connectivity and allows the user to download and install applications. Malicious applications can pose a threat to the security of the system.

Platform security mechanisms must provide following:

- Protect the user.
  It must not be possible for malicious application to corrupt or steal the device owner's personal data. Also malicious application must not be able to misuse the device and incur costs by sending sms to pay numbers. If device is stolen, it should not be possible to access the user's private data.

- Protect the device.
  Device functionality and reliability must satisfy specification requirements. It must not be possible to change critical device parameters. Changing RF, WiFi values can cause device malfunctioning and violate regulatory requirements.

- Protect the business.
  Phones are sold via different channels. Operators often subsidize devices. Breaking a SIM/Subsidy lock immediately mean lose of business. Operators want product customization - certain applications and service should only be available on its devices, and possibly limit what can be installed on the device.

- Enable new services.
  Providing services such as Music Store or Application Store requires device to support copy-protection. Services like mobile payments and billing requires secure handling of customer data.

Comparing to personal computers where users have more control over the device and often prepared to perform administrative tasks, mobile device users do not expect that they need to make complicated configuration of the device. That requires security framework to provide protection without additional maintenance effort and to allow applications coming from different sources to get access to protected resource in controlled manner.

2 Existing Linux Security Frameworks

Linux kernel has already several security frameworks such as SELinux, Smack, Tomoyo. While providing Mandatory Access Control (MAC) implementation, they do not provide end-to-end solution for security, taking into account also software distribution and developers ecosystem. Here we provide some reasons why we decided to develop our own security framework.

2.1 Traditional Unix DAC

Unix DAC is a classical access control model which is based on restricting access to objects based on identity of the subjects and groups to which they belong. The main difficulty to use Unix DAC is a lack of process based access control.

Mobile device is normally a single user device where all processes are either running under root account or the same user account. Processes with the same user ID have unlimited access to the resources of each other. This pose a threat that malicious application can corrupt or steal the data of other applications.

The way to work around this issue is to run processes under different users accounts and groups. But this approach is not generic and requires administrative work to maintain needed information.

2.2 SELinux

SELinux is LSM-based MAC implementation [1]. SELinux is very powerful, but requires very complex and centralized policy administration. That is not problem for the servers which are usually centrally administered by professional people. Required administration effort makes it very complicated to use in smartphone platform.

2.3 Smack

Simplified Mandatory Access Control Kernel (SMACK) is LSM-based, relatively simple MAC implementation as alternative to SELinux [2]. It's operational logic is simple: labels are attached to system components and access rules between the labels are defined by the system administrator. SMACK provides primitive security API, but doesn't provide the application enough granularity to provide detailed access control.

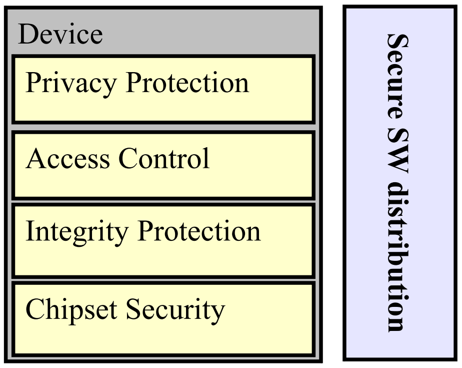

Figure 1: Security Frameworks Layers

2.4 Tomoyo

Tomoyo is a lightweight MAC implementation [3] . It performs pathname-based access control. TOMOYO utilizes "process invocation history" and requires administrative actions on the target system.

3 Mobile Simplified Security Framework

Mobile simplified security framework (MSSF) is a set of mechanisms to protect entire platform consisting of following layers (Figure 1):

- Chipset security.
  Provides secure cryptographic services for OS level security.
- Integrity protection.
  Ensure protection of TCB, applications and data. Provides protection against offline attacks.
- Access Control.
  Limits application access to critical resources. Provides protection against runtime attacks.
- Application privacy protection. Provides integrity and confidentiality protection for applications and services. Provides protection against offline attacks.

Security Framework relies on the secure software distribution model. The goal of Secure SW distribution is to ensure the authentication of a SW's source.

4 Chipset Security

4.1 Features

Chipset security is the key subsystem on which whole security framework relies on. It provides tamper-resistant securure services and serves the same purpose as Trusted Platform Module (TPM) [4] or Mobile Trusted Module (MTM) [5]. It includes:

- Root symmetric devices specific key.
  The root symmetric device specific key is unique one-time programmable (OTP) key, which is used to derive keys used for local cryptography operations. It is also used to derive a unique public identifier of the device.
- Root public key.
  The root public key is OTP key and is used to verify that software components are coming from trusted source.
- Provides trusted boot (chain of trust).
  Root public key is used to verify integrity of the bootloader and SW image.
- Secure services.
  Secure key management and cryptography services.
- Provides Secure Execution Environment (SEE).
  SEE consists of secure ROM and RAM which is isolated from reset of the system. It allows execution of integrity protected applications, which can utilize secret device keys and provide specific secure services for the OS. Protected applications are needed by Protected Storage and DRM framework.

4.2 Operation modes

Nokia MeeGo 1.0 N device will have two operation modes: normal and open mode.

Devices shipped by Nokia come with original Nokia SW having device configured for normal mode Security functionality such as access control and integrity protection are enabled and enforced. Applications and services are able to use device keys and cryptographic services. In normal mode authorized applications are given access to copy protected content. Unauthorized modification of the security policy is impossible.

Developers who want to have unrestricted access to the platform resource, might turn the device into open mode by flashing an ”open mode” image.

Open mode is mostly needed only for low-level development and deep device customization. Ordinary application developers test their applications with normal mode as well. Open mode provides the same functionality as in normal mode except that there is no access to copy protected content. In open mode image security is enforced but allows developers to modify the policy and to allow their applications to access more device resources without the need of application certification process. In open mode it is possible to use own kernel. However in open mode, chipset security generates different keys which are incompatible with normal mode keys. This makes it impossible to get an access to copy protected content.

4.3 Boot process

Boot process is shown on figure 2.

Bootloader image is verified with Root Public key.

If bootloader verification fails, device is automatically resetted.

In the case when kernel verification fails, device either restarted or booted to open mode. If device is SIM locked, it is not allowed to boot into open mode unless explicitly allowed by the device customization.

5 Integrity Protection

MSSF has an integrity protection subsystem called Validator, which protects the integrity of kernel modules, executables and libraries and data files. The primary goal is to protect integrity of SW components which belongs to Trusted Computing Base (TCB). Trusted Computing Base includes all hardware, firmware and software components that are critical to the security of the entire platform.

The integrity subsystem is shown on Figure 3.

Validator is implemented as LSM kernel module and is based on the DigSig project [7]. The difference is that

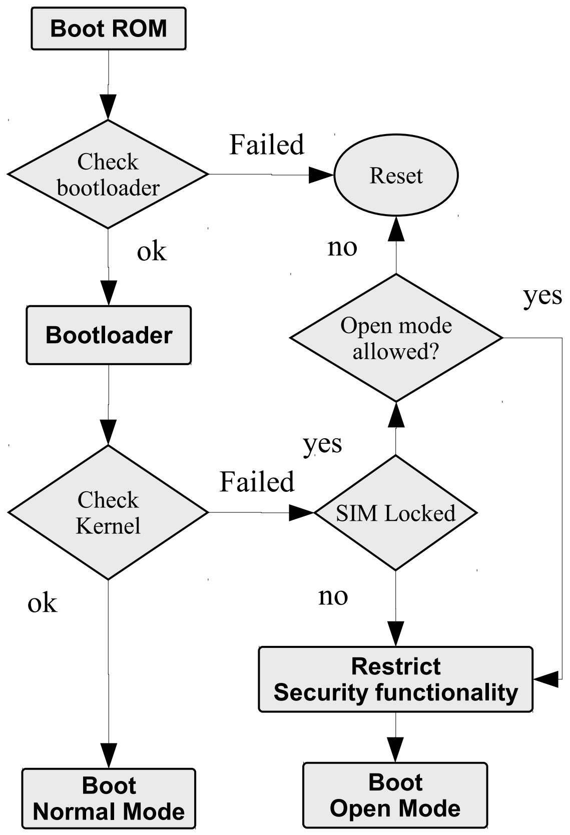

Figure 2: Boot process

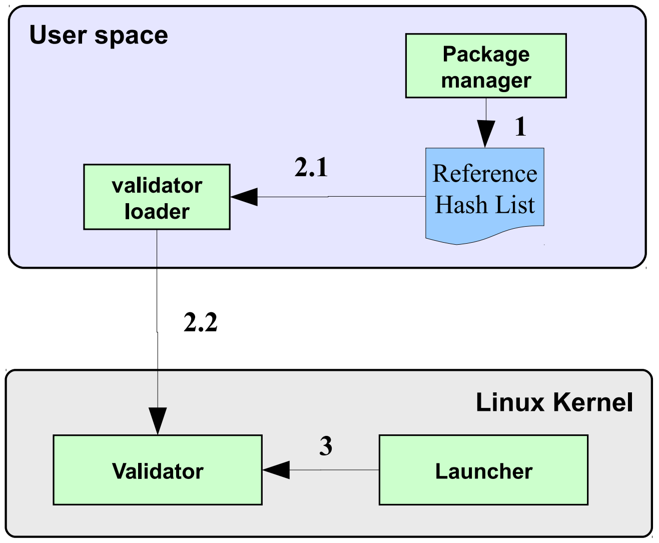

Figure 3: Integrity Protection Subsystem

instead of using ELF header, device maintains a reference hash list (`/var/lib/mssf/refhashlist`) of all protected binaries. Integrity of the reference hash list is also protected by the device signature.

Reference hash list includes SHA1 hash of the file, file attributes, and AC related data.

Debian package contains sha1 hashes of all executables and important data files. Package manager updates reference hash list upon package installation, removal or upgrade.

Use of single reference hash list instead of ELF signed binaries and EA has certain advantages. It allows to have protection for scripts and data files, which do not have ELF header. It does not require to verify integrity of EA itself for every file using digital signatures. Integrity verification of reference hash list is more mobile device friendly. Also EA is a subject to offline removal attack, which cannot be detected.

Installation includes following steps:

1. Package Manager installs new binaries and updates reference hash list.
2. Validator loader loads new or updated hash list into the kernel.
3. Validator calculates and compares hash and file attributes upon execve() call. It also verifies hashes of shared libraries upon mmap() call.

Integrity protection policy defines action when integrity verification fails. Currently it only blocks the execution.

Validator also has a support for integrity protection of non-modifiable data files. That is used for protection of critical configuration files.

Source code of this module is located in:
linux/security/mssf/validator

6 Access Control

6.1 Introduction

Access Control framework provides runtime protection.

We had following design goals for access control:

- Process-based access control to protected resources.
- Minimal changes to the default Linux model.
- No need for centralized security policy administration.

Access control framework includes following components:

- Manifest file.
  Manifest file is included to the package and contains a list of executables and its credentials.
- Device Security Policy.
  Located on the device and defines repository trust level and credentials, which can be granted to packages coming from that repository.
- Credentials Policy.
  It is a file which contains mapping of credentials to executables. Package Manager updates this file when packages are installed, upgraded or removed.
- Package Manager.
  In addition to installing the application, Package Manager updates Credentials Policy database.
- Credentials Policy loader.
  It is called during boot to read and import credentials policy into the kernel.
- Credentials Manager.
  Provides credentials management and assignment to the process. It is implemented as a set of kernel modules (see Implementation Details).

6.2 Credentials

Access control in Linux is based on credentials. Conventional credentials in Linux are UIDs, GIDs and POSIX capabilities.

MSSF access control framework uses term protected resource to denote any virtual object which represents some functionality or data, such as tasks, files, sockets, devices.

MSSF access control framework extends credential set with **resource tokens** and **application identifier**.

6.2.1 Resource Tokens

Each protected resource is assigned a resource token, which is just a string representation of the resource, for example Cellular, UserData. Other security frameworks often use the term ”label”.

Applications or services need to declare requested or provided resources. For that purpose, package, which hosts those executables, must include the Manifest file.

Resource tokens can either be global or package specific. Global tokens comes from special package. Package specific tokens are declared as pkgname::token.

Access Control model does not define subject labels, object labels and access control rules, such as ’subjectlabel objectlabel access’, but resource tokens play the role of both subject and object labels. Resource tokens are assigned to the process subjective context upon startup. Enforcement mechanism just needs check if a process possess a token.

Rules are not enforced by MSSF access control model automatically, but processes need to perform enforcement manually when task is being accessed by retrieving client tokens. Currently supported access method is via Unix domain sockets.

API is provided to get credentials of the process.

6.2.2 Application identifier

Application identifier is used to derive application specific keys (see Privacy Protection). Application identifier is defined as:

AppID =
{SourceID, Package Name, Application Name}

for example {ovi.com, CoolTools, AddressBookPlugIn}.

Application name is defined in manifest file.

Application identifier has following properties:

- Unforgeable & Trustworthy.
  SourceID is defined in Device Security Policy and protected by repository keys.

- Unique.
  System can only have one package with the same name.
- Persistant.
  It remains the same between reboots, application updates, and for different instances of the same application.

6.3 Device Security Policy

Software packages are distributed via Debian-like repositories. Repository contains a package list, which is signed with repository private key and verified by the package manager using repository public key. Package based signing using PGP and X.509 is also supported.

The purpose of the Device Security Policy is to define repository trust level and credentials, which can be granted to packages coming from that repository.

Device Security policy contains entries which have following format:
{SourceID : Trust Level : Public Key : Allowed credentials}, where:

SourceID is a meaningful name for the origins of the repository, for example in a form of domain name.

Trust Level is an ordinal number and defines repository ranking. During update, package can only be updated from repository which has the same or higher trust level. It will prevent possibility for some 3rd party repositories by mistake or on purpose to replace trusted package with untrusted one.

Allowed credentials is a list of credentials, which can be granted by this repository.

Public Key is a repository public key which is used to verify repository package list.

Example of policy entry can look like:
{nokia.com : 1 : ABCDEF : UserData, Cellular}.

Package Manager uses device security policy when packages are installed or upgraded. Granted credentials, which is added to the Credentials Policy, are the result of 'intersection' operation over credentials set from Manifest file and security policy. Only allowed credentials are added to the Credentials Policy.

Figure 4 shows the concept of distribution model.

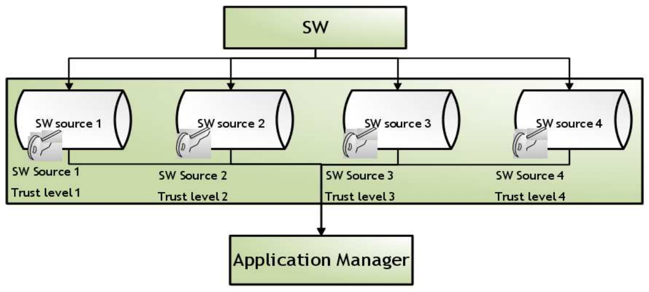

Figure 4: SW Distribution Model

6.4 Manifest File

If an application requests or provides some credentials, the package is expected to ship with the Manifest file `<package>.mssf` with description of credentials.

Package manager updates Credentials Policy based on the manifest file and constraints from the device security policy as described in the previous section.

Manifest file is written in XML and defines following tags:

- **<request>**
  requested credentials
- **<provide>**
  provided credentials
- **<credential name="credential name">**
  credential name
- **<for path="path">**
  absolute path to the program executable
- **<dbus name="dbus service name">**
  D-bus service name
- **<bus="bus type">**
  D-bus type (system or session)
- **<own="credential name">**
  Credential to bind to a specific d-bus service name
- **<interface name="interface name">**
  D-Bus interface name

6.4.1 Manifest file for client-server example

In the example bellow, server defines resource token **UserData**, which is needed by the client to access the server.

```
<mssf>
  <provide>
    <credential name="UserData"/>
  </provide>
</mssf>
```

In the example bellow client declares that it requires a token **UserData** and **Cellular**

```
<mssf>
  <request>
    <credential name="UserData"/>
    <credential name="Cellular"/>
    <for path="/usr/bin/userdatamanager">
    <for path="/usr/bin/userdataclient">
  </request>
</mssf>
```

In this example both applications **userdatamanager** and **userdataclient** will get the same credentials.

6.4.2 Manifest file for traditional credentials

Manifest file can be used also to assign conventional credentials such as UID, GID and POSIX capabilities.

```
<mssf>
  <request>
    <credential name="UID::email"/>
    <credential name="GID::email"/>
    <credential name="CAP::cap_sys_rawio"/>
    <for path="/usr/bin/mssf-dbus-server"/>
  </request>
</mssf>
```

6.4.3 Manifest file for policy update

Manifest file is also used to update device security policy. Policy update is done via the special authorized package.

```
<mssf>
  <domain name="MyDomain" rank="30">
    <allow>
      <credential match="*"/>
      <deny>
          <credential name="drm"/>
      </deny>
    </allow>
    <origin>
      <keyinfo>
        mQGiBE...O6XB
     </keyinfo>
    </origin>
  </domain>
</mssf>
```

6.4.4 Manifest file for DBUS

We implemented a DBUS extension which uses credentials API to verify client credentials. Manifest file may have dbus specific tags, which are used by the Package manager to generate DBUS policy.

Manifest file for DBUS-server is shown on Figure 5.

Generated DBUS policy file is shown on Figure 6.

DBUS client uses the same manifest file as with peer-to-peer access control (Figure 7).

6.5 Package Installation

Installation of new applications and services is done via packages (Figure 8).

Package installation includes following steps:

1. Package arrives to the Package Manager together with Manifest file.
2. Package Manager checks the Device Security policy for the information.
3. Package Manager updates the Credentials Policy according to the ”Intersection rule”.
4. Package Manager possibly updates D-Bus policy.
5. Package Manager updates runtime credentials policy in the kernel.

6.6 Startup

Startup process is shown on Figure 9.

1. At a boot, Credentials Policy loader reads Credentials Policy and loads it into the kernel.

```
<mssf>
<provide>
     <credential name="access"/>
     <dbus name="com.meego.mssf.example" own="mssf-dbus-server" bus="session">
         <node name="/">
             <interface name="mssf.Example">
                  <annotation name="com.meego.secure.Access" value="access"/>
             </interface>
       </node>
     </dbus>
</provide>
<request>
     <for path="/usr/bin/mssf-dbus-server"/>
</request>
</mssf>
```

Figure 5: DBUS server manifest

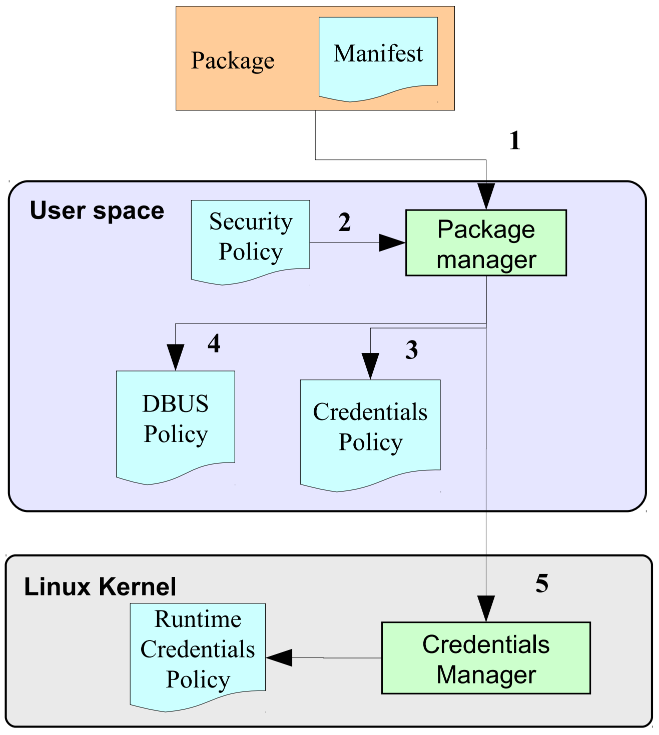

Figure 8: Package instllation

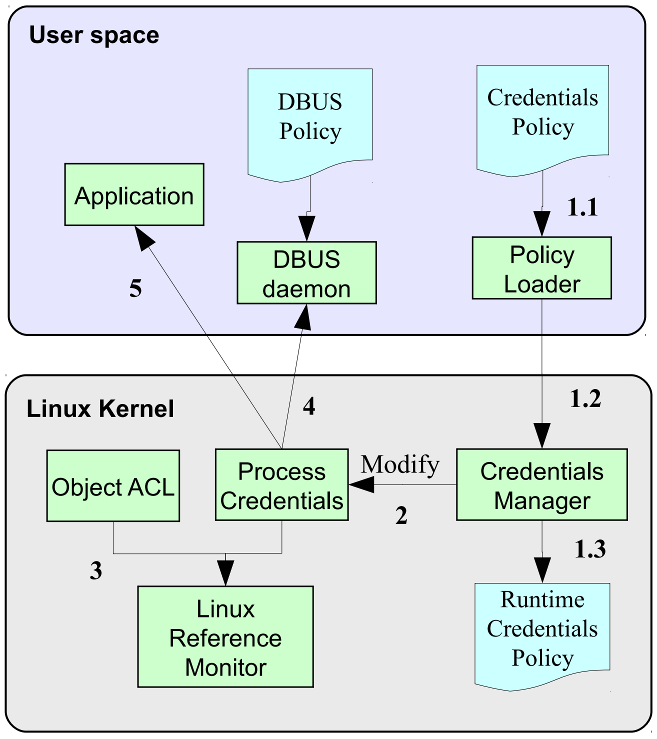

Figure 9: Startup

```
<busconfig>
  <policy context="default">
    <deny own="com.meego.mssf.example"/>
  </policy>
  <policy creds="mssf-dbus-server::mssf-dbus-server">
    <allow own="com.meego.mssf.example"/>
  </policy>
  <policy context="default">
    <deny send_destination="com.meego.mssf.example" send_interface="mssf.Example"/>
    <deny receive_sender="com.meego.mssf.example" receive_interface="mssf.Example"/>
  </policy>
  <policy creds="mssf-dbus-server::access">
    <allow send_destination="com.meego.mssf.example" send_interface="mssf.Example"/>
    <allow receive_sender="com.meego.mssf.example" receive_interface="mssf.Example"/>
  </policy>
</busconfig>
```

Figure 6: DBUS policy

```
<mssf>
  <request>
     <credential name="mssf-dbus-server::access"/>
     <for path="/usr/bin/mssf-dbus-client"/>
   </request>
</mssf>
```

Figure 7: DBUS client manifest

2. Upon application startup, Policy Manager modifies process' credentials according to the received credentials.

3. File AC.
   Validator checks process credentials using kernel API.

4. D-Bus.
   D-Bus daemon checks client credentials using libcreds (see DBUS Integration).

5. Client-server.
   Application checks client credentials using libcreds.

6.7 Credentials APIs

When a client issues a request to a server, the server may wish to check whether the client is authorized for the requested operation. It is done using the **libcreds** library, which gives the server a way to read the credentials of the client process and to permform the desired credential checks.

Kernel credentials API is also available.

Example of using API is shown on Figure 10.

creds_str2creds() converts token string to internal format.

creds_getpeer() retrieves credentials of the client process.

creds_have_p() checks if the client process has required credential.

6.8 DBUS support

6.9 File System Access Control

Debian packages often contain installation scripts which runs under the root. It allows them to modify any files

```
creds_value_t value;
creds_type_t type;
require_type = creds_str2creds("UserData", &require_value);
fd = accept(sockfd, &cli_addr, &clilen);
ccreds = creds_getpeer(fd);
allow = creds_have_p(ccreds, require_type, require_value);
if (allow)
   write(fd, MESSAGE("GRANTED\n"));
else
   write(fd, MESSAGE("DENIED\n"));
```

Figure 10: Code example

on the system which can make device unusable. In order to prevent that is necessary to protect access to certain files and folders.

Validator reference hash list also contains list of tokens, required to access files and folders. Validator uses resource tokens kernel API to verify process's permission to access the file.

6.10 Kernel implementation details

Credentials Manager is implemented as set of kernel modules: **restok**, **credp**, and **creds**.

6.10.1 restok

restok module provides a persistent mapping of strings to unique dynamically assigned identifier numbers.

The generated identifiers are used as supplementary group numbers in the task structure and provide additional, dynamically configured credentials for processes. An access to service is protected by requiring a presence of specific credential in the task context (supplementary groups).

Although these numbers are used as supplementary groups, they are not persistent and cannot be used as file system groups in permanent storage.

Once the string has been assigned an identifier, this assignment cannot be changed while restok module is loaded. If the module is compiled into the kernel, the assignments are permanent until the next boot.

To provide different name spaces, the strings form a forest of trees. The string corresponding the identifier, is the path from the ground up to the node that defines the identifier. Within the path "::" is used as a separator.

The module creates a special default tree with an empty string as a name of the root. A string without any "::" is assumed to be a direct child of this default root. For any other identifiers, the string must be a full path from one of the roots to the defining node.

The above rule would make it impossible to address any other root nodes. Thus, the module implements a special case, where a string containing "name::name" collapses into "name". Some examples:

foo -> "::foo" (symbol under default root)
foo::foo -> "foo" (root level symbol, different from previous)
foo::foo::foo -> "foo" (a repeated name is reduced to single instance)
::foo -> "::foo"
:: -> "" (= default root)

The purpose of the "default root" is to provide applications a place to define simple symbols, which do not conflict with the root names, which are used for identifying different name spaces.

The string used in resolving an identifier (function 'restok_locate') is always a full path or a string under the default root.

Strings are defined one level at a time (function 'restok_define'). The identifier of the parent must be supplied. A zero as parent creates a new root.

A malicious application could create a huge number of mapped strings. This is the only reason for limiting the capability of creating new mappings.

For debugging purposes, restok can be compiled as a module, but real usage requires that it is built in.

Source code of this module is located in:
linux/security/mssf/restok

6.10.2 credp

credp module provides credentials management and assignment to the process.

This module maintains a runtime credentials policy, which is a mapping of credentials to an executable or identifier. The module provides a user space API via securityfs entry **/sys/kernel/security/credp/policy**, which is used by tools to add and remove rules from the runtime policy. When adding a new rule, kernel performs translation of resource token strings to identifiers using kernel API provided by the **restok** module.

Credentials Policy database is located in the **/var/lib/mssf/restok/restok.conf** file. During boot, the policy loader **mssf-loader** reads rules from the policy database and imports them into the kernel. Upon installing a new package, package manager, in addition to updating the policy database, imports new rules into the kernel.

To perform credentials assignment, the credp module registers a hook that is called when new executable is about to be started. To achieve that, we implemented a small patch for **security/commoncap.c:cap_bprm_set_creds()**, which allows modules to register credentials assigner operations.

Operations has **apply()** function, which is called from cap_bprm_set_creds() upon executables startup via **execve**.

Source code of this module is located in:
linux/security/mssf/credp

6.10.3 creds

creds module provides an API for user space access control in client/server architecture. The module provides a user space API via securityfs entry **/sys/kernel/security/creds/read**, which is used by **libcreds** library.

This module gives the server a way to read the credentials of the client process and to perform the desired credential checks. Because this is targeted for access control, the returned credentials are the *effective* credentials.

Without this service, getting information about the credentials of another process, is only possible by parsing the "/proc/<pid>/status" content, which is fragile to format changes and it only provides maximum of 32 supplementary groups.

In addition to credentials retrieval, this also provides translations between string and numeric values of credentials. Currently only capabilities names need to be provided and handled by the kernel.

If a companion module 'restok' is compiled, this provides a gateway for translations of symbols defined there. The restok defined symbols are currently mapped into credentials via use of supplementary groups. Other mappings, like defining a totally new credential type for those, are possible in future.

Source code of this module is located in:
linux/security/mssf/creds

7 Privacy Protection

7.1 Protected Storage

Protected Storage provides protection against offline attacks.

Mobile device can be lost or stolen. For that reason it may be a good idea to store sensitive data such as contacts in encrypted form.

Also some security related configuration data such as security policies, credentials policy, reference hash list, certificates, and other configuration data requiring protection against unauthorized modifications.

For that purpose MSSF provides Protected Storage service. Protected storage can be global (G), private (P) or shared between applications (S). It can be used for integrity protection (s) or also for confidentiality (e) protection.

Protected storage implementation uses chipset cryptographic services, and is based on application id and resource tokens.

	Global	Private	Shared
Signed	Gs	Ps	Ss
Encrypted	Ge	Pe	Se

Table 1: Protected Storage types

Private storage uses an application specific key, which is derived from an application id: **K(device key, AppID)**. Global and Shared storages use a shared key, which is derived from a resource token: **K(device key, Resource Token)**. Keys are device specific and ensure copy protection.

If the protected storage is based on a resource token, only those applications that have the resource token can manipulate the store. If the protected storage is based on an application id, only those binaries that share the same application id can manipulate the store.

Applications need to use special API in order to use protected storage.

7.2 Security FS

In order to provide easy-to-use protected storage for such applications, where it doesn't make sense to use proprietary API, MSSF provides a FUSE-based user space file system for similar functionality through the normal POSIX file handling API.

This means in practice that the encryption is transparent for each application, in a similar way with a normal block-device (disk partition) encryption. But unlike with partition-wide encryption, applications cannot see and/or decrypt each other's files unless they have proper credentials, regardless of whether they are running in the same or different user-id.

Manifest file is extended to describe mount points and type of the storage.

Security FS is under heavy development now.

8 Performance

MSSF has slight effect on system performance. Boot time, application startup, runtime performance are affected.

8.1 Integrity protection

Integrity protection (Validator) has most significant implication to system performance. Binary startup time increases, because Validator needs to calculate the SHA1 hash. Verification is done only when binary is loaded for the first time.

Performance of Validator is heavily depends on use and performance of SHA1 HW accelerator. Nokia MeeGo 1.0 N device has SHA1 HW accelerator which is, according to our measurements, quite CPU and power efficient. In our case, application startup time increases by 5 to 10%, and total boot time by 2 to 3%.

8.2 Access Control

Performance penalties given by Access Control framework is insignificant.

Credentials Policy is loaded during boot and requires time insignificant to the total boot time.

Access Control affects application startup time, because Credentials Manager needs to find a policy, and to assign credentials if one exists. It increases startup time by 2.5% (or 6ms in our case).

9 Conclusions and Future work

Mobile Simplified Security Framework is a comprehensive, light-weight alternative to heavy security frameworks for mobile devices. Secure SW distribution model is a important part of MSSF end-to-end security model.

Latest Linux kernel provides integrity subsystem called IMA [6], but verification module (EVM) has not been integrated yet. We will consider possibility to use it when all components are available in the kernel.

MSSF Access Control has similarities to SMACK and we currently investigating possibility for co-operation.

MSSF project can be found on [8]. There libraries, tools and patches for the Linux kernel can be found.

10 Acknowledgements

I would like to thank my Nokia colleagues Ilhan Gurel, Vesa Jaaskelainen, Janne Karhunen, Markku Kylanpaa, Juhani Makela, Janne Mantyla, Elena Reshetova, and Markku Savela for some material, suggestions and review of the article.

References

[1] SELinux - Security Enhanced Linux, `http://www.nsa.gov/research/selinux/index.shtml`

[2] SMACK - Symplified Mandatory Access Control Kernel for Linux, `http://schaufler-ca.com/`

[3] TOMOTYO Linux - MAC implementation for Linux, `http://tomoyo.sourceforge.net/`

[4] Trusted Computing Group, TPM main specification, `https://www.trustedcomputinggroup.org/specs/TPM/`

[5] TGG Mobile Trusted Module specification, 2006, `https://www.trustedcomputinggroup.org/specs/mobilephone/`

[6] Linux kernel integrity subsystem, `http://linux-ima.sourceforge.net/`

[7] DigSig project, `http://disec.sourceforge.net/`

[8] Mobile Simplified Security Framework Project, `http://meego.gitorious.org/meego-platform-security`

Proceedings of the Linux Symposium

July 13th–16th, 2010
Ottawa, Ontario
Canada