Symbian OS Internals/08. Platform Security

by Corinne Dive-Reclus

Computers are like Old Testament gods; lots of rules and no mercy.
Joseph Campbell

In this chapter I will introduce a new concept - that of platform security. I will not explore this subject in too great a depth here, as it will shortly be the subject of a book of its own (Platform Security for Symbian OS, by Craig Heath. Symbian Press). Instead, I will discuss how the core operating system components contribute to the implementation of platform security on Symbian OS.

Introduction

EKA2 was designed specifically for mobile phones, and so we had to meet the security requirements of both the mobile phone industry and the users of those mobile phones. This meant that we had to understand those requirements in detail, and in particular to understand the impact they would have on the essential components of the operating system, such as the kernel, the file server and the loader.

To decide how we could implement security in a mobile phone, it was important that we were aware of the user's perception of her phone, and the main differences between the well-known desktop environment and the mobile phone environment:

- For our typical end-users, mobile phones are **not** like computers: people expect them to be simple to use, reliable and predictable
- Mobile phones are personal: people do not share their mobile phones with others as they share their landline phone or family PC.

When we asked users what security they expect from a mobile phone, they responded:

1. I don't want nasty surprises when I receive my telephone bill
2. I want my private data to stay private!

So why not just re-use all the security measures from a desktop computer? The answer, of course, is that phones have specific characteristics, linked to the physical device itself and the way that it is used, that make them fundamentally different from the desktop environment:

1. There are the limitations imposed by the device itself - compared to a desktop, the screen is small and the keyboard is limited. This restricts the amount of information that can be displayed to the user (typically just one or two short sentences) and also the number and complexity of passwords the user is willing to enter
2. There is no IT support department to rely on: the user must not be asked questions that she cannot comprehend
3. The operating system is hidden: files and even processes are invisible. Let's take the case of the Sony Ericsson P900: when the user clicks on an application, she does not know (and should not need to know) whether this is starting a new process or re-using an existing one. Therefore the user should not be asked to make a security decision based on the name of a process or a file.

So the main goals of Symbian's platform security architecture were:

1. To protect the phone's integrity, because users want reliable phones
2. To protect the user's privacy
3. To control access to networks, because users care about their bills.

Our constraints on platform security were:

4. It must be easy for developers to use 5. It must be fast and lightweight 6. It must only expose security policy to users when they can understand it.

The platform security architecture of Symbian OS is based on three key concepts:

1. The OS process is the unit of trust
2. Capabilities are used to control access to sensitive resources
3. Data caging protects files against unauthorized access.

Unit of trust

We define a unit of trust as the smallest entity to which we grant a set of permissions.

Concept

A phone only has a single user, so the security policy is not about protecting different users from each other, as in a desktop environment. Instead, it is about controlling exactly what applications are allowed to do when they run. It was clear that we should choose the unit of trust to be the process, because this is already the fundamental unit of memory protection on Symbian OS. (As I've shown in previous chapters, the kernel cannot protect individual threads from each other if they are running in the same process, because they all have unpoliced access to the same memory.) We identified three main levels of trust, and these are shown in Figure 8.1.

The Trusted Computing Base (TCB)

The Trusted Computing Base (TCB) is responsible for maintaining the integrity of the device and for applying the fundamental rules of platform security. Very few components - the kernel, the file server and, on open devices, the software installer and its registry - are part of the TCB and have unrestricted access to the device's resources. The rest of the operating system implicitly trusts these to behave correctly; because of this, all TCB code is reviewed very strictly.

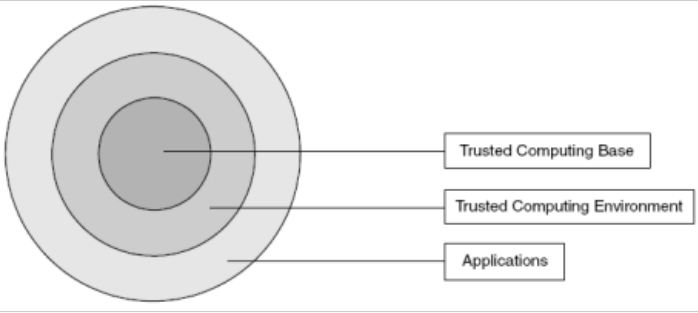

Figure 8.1 *Processes and trust*

The Trusted Computing Environment (TCE)

Beyond the core of the TCB, other system components require access to some, but not all, sensitive system resources. For example, the window server requires direct access to keyboard events, but not to the ETEL server. The *Trusted Computing Environment* is composed of these key Symbian OS components that protect the device resources from *misuse*. In Symbian OS, server programs are used to control access to shared resources, so we define the TCE as the set of all system servers.

Applications

The final level of trust is ordinary applications. In most cases, these will not have any capabilities that can endanger the integrity of the phone, because they will access sensitive resources through components in the TCE and the TCB. In certain circumstances they may require capabilities - to access information private to the user, network services or local connectivity.

The kernel's role

As I mentioned before, the components in the TCB need to be entirely trustworthy - and because of this they are reviewed very carefully. The key component of the TCB is the main subject of this book, the kernel.

In a secure environment, the first duty of the kernel is to ensure the availability of hardware resources that critical processes require, and to provide access to those resources in a short, bounded period of time. I will discuss the real-time aspects of the kernel in Chapter 17, *Real Time*, and so will not dwell on them further here.

The second duty of the kernel is to provide strong protection of the process memory space (including its own) to guarantee that processes are protected against each other and therefore that the behavior of a trusted process cannot be compromised by another process. This aspect of the kernel is covered in Chapter 7, *Memory Models*, and again I will not discuss it further here. Instead I will proceed to enumerate the new security features of EKA2.

Access to user-mode memory

Any access to user-mode memory from within privileged code (that is, the kernel and device drivers) uses special accessor and copy methods to apply user-mode memory permissions to the access. This ensures that invalid data pointers passed to kernel functions by user processes do not cause the kernel to fail. See Section 5.2.1.5 for more details.

Thread stacks and heaps

Thread stacks and heaps are *private* chunks: they cannot be mapped and made accessible to processes other than their owner.

Process memory overwritten

When the kernel allocates memory to a process, it overwrites it with zeroes, to prevent any private data from the previous owner being accessible to the new process.

New IPC framework

We have replaced the existing Symbian OS inter-process communication (IPC) framework API (V1) with one designed for trusted IPC (V2). Please refer to Chapter 4, *Inter-thread Communication*, for more details. Here is an overview of the new features of the framework that provide increased security:

- The new server-side IPC classes have a *2* added to the EKA1 name to indicate that this is the version 2 of the IPC mechanism: for example, `CSession2`, `CServer2`, `RServer2`, `RMessage2`, `RMessagePtr2`
- We have removed old, insecure methods that allowed reading and writing arbitrary process data (for example, `<tt style="font-family:monospace;">RThread::ReadL()`)</tt> and we have replaced them with methods that only allow access to data areas specifically passed by the client in a message (for example, `<tt style="font-family:monospace;">RMessagePtr2::ReadL()`)</tt>. Now the server can only access data that is associated with a valid message and it must now use the `RMessagePtr2` class to do this. This also means that the data is not available to the server after it has processed the message
- Parameters passed in an IPC request are now typed for increased robustness. For extra security, their lengths are also specified, even in the case of a pointer, to ensure that the server will not read or write more than the client expected to disclose: any attempt to read before the pointer's address or beyond its length will fail.

ARM v6 never-execute bit

If the hardware supports it, then EKA2 will take advantage of the ARMv6 never-execute bit in the page permissions (see Chapter 7, *Memory Models*, for more on this). This is used to deny the execution of code from stacks, heaps and static data, with the aim of preventing buffer-overflow attacks. (These inject malicious code into stacks or heaps over the limit of an array passed as a function parameter to trick a process into executing that code.) EKA2 only allows execution from execute-in-place ROM images, software loaded by the Symbian OS loader and explicitly created *local code* chunks (for compiled Java code such as JIT or DAC).

Capability model

Concept

A capability is an authorization token that indicates that its owner has been trusted to not abuse resources protected by the token. This authorization token can grant access to sensitive APIs such as device driver APIs or to data such as system settings.

Capability rules

The following capability rules are fundamental to the understanding of platform security. They strongly re-enforce the first concept that the process is the unit of trust. They are worth reading carefully; they are somewhat less obvious than they may first appear.

Rule 1. Every process has a set of capabilities and its capabilities never change during its lifetime.

To build a Symbian OS executable, you have to create a project file (MMP). The Symbian OS build tools now require an additional line that defines what capabilities should be given to the executable in question. The build tools will read the MMP file and write the desired capabilities into the resulting binary, just as they do with UIDs.

MMP files are used to associate capabilities with every sort of executable code. Programs (.EXE), shared libraries (.DLL) all have their capabilities defined in this way. When it creates a process, the kernel reads the capabilities from the header of the program file on disk and associates those capabilities with the process for the remainder of its lifetime. The kernel stores the capabilities in memory, which is inaccessible to user-mode processes, to prevent tampering. It provides user-side APIs to allow user-side code to check that a process requesting a service has a specific set of capabilities.

In the following example, litotes.exe has been assigned two capabilities, ReadUserData and WriteUserData. These grant the application access to APIs that access and modify the user's private data, such as contacts.

```
// litotes.mmp
TARGET litotes.exe
TARGETTYPE exe
UID 0x00000000 0x00000123
SOURCEPATH ..\litsource
SOURCE litotes.cpp
```

```
USERINCLUDE ..\include
SYSTEMINCLUDE \epoc32\include
...
CAPABILITY ReadUserData WriteUserData
```

Symbian OS provides three main methods for programs to use services provided by other executables:

1. Loading a DLL and calling its API
2. Making requests of server programs
3. Loading and calling a device driver.

In each case, the caller's capabilities must be checked. I will explain how this is done, taking each of these cases in turn.

Loading a DLL

As mentioned earlier, a DLL is executable code and its binary will contain a set of capabilities. But a DLL must always execute within the context of a process - the process that loads the DLL - and rule 1 stated that the capabilities of a process can never change. Together, these statements imply that the DLL code must run at the same capability level as the loading process. This leads to the principle of DLL loading, rule 2:

Rule 2. A process cannot load a DLL that has a smaller set of capabilities than it has itself''.

The need for this constraint follows from the fact that all DLL code runs at the capability level of the loading process. This means that DLL capabilities are different to process capabilities, in that they don't actually authorize anything; they only reflect the confidence placed in them to not abuse or compromise a host process with those capabilities.

DLL capabilities are policed by the loader, and so are checked only at load time. After that, all the code contained in the DLL will run with the same capabilities as the code directly owned by the process. This means that it will be subject to the same capability checking when it accesses services provided by other processes.

Rule 2b. A DLL cannot statically link to a DLL that has a smaller set of capabilities than it has itself''.

Rule 2b is a corollary of rule 2, as you can see if you replace the word *process* in rule 2 with the words *loading executable* - but it is clearer to restate it as I have just done. The Symbian OS loader directly resolves static linkage between the loading executable (program or library) and the shared library. So, when a library is linked to another library, the loader does not take into account the top-level program. To see why, let's look at an example. Assume that a DLL, METAPHOR, with capability ReadUserData, wants to statically link to a DLL without this capability, SIMILE. When the linker resolves this static direct linkage, it does not know whether METAPHOR will be statically linked to a program with or without ReadUserData. If the final program EPITHET does have ReadUserData, and if METAPHOR were statically linked to SIMILE, then this would be unsafe as SIMILE might compromise EPITHET's use of ReadUserData capability.

Hence the secure route is to always reject the linkage when the linked DLL has a smaller set of capabilities than the linking one. This is illustrated in Figure 8.2

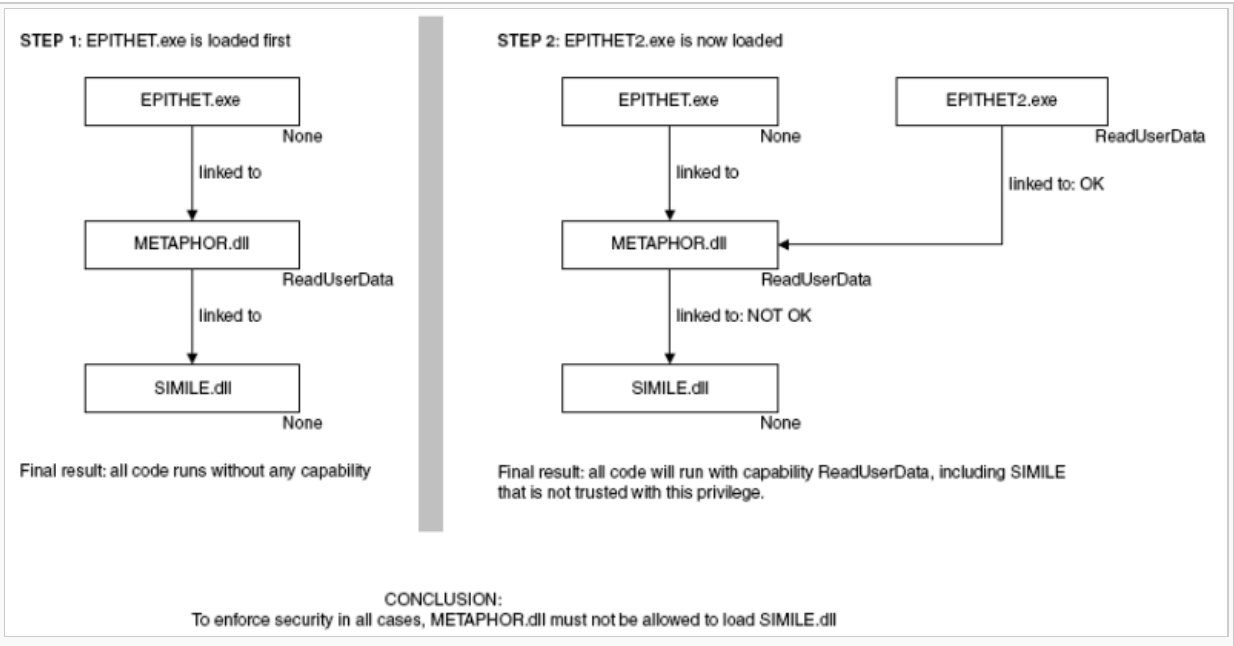

Figure 8.2 *Capability rules and DLL linking*

Rule 2 and its corollary prevent malicious or un-trusted code being loaded into sensitive processes, for example a plug-in into a system server. The rules also encourage the encapsulation of sensitive code inside well-known processes. The loader provides this security mechanism for all processes; relieving them of the burden of identifying which DLLs they can safely load or link to.

The following examples show how these rules are applied in the cases of statically and dynamically loaded DLLs respectively.

Examples for statically linked DLLs

Assume that the program PLOT.EXE is statically linked to the library RHYME.DLL and the library RHYME.DLL is statically linked to the library REASON.DLL.

Example 1 Also assume that:

1. PLOT.EXE holds Cap1 and Cap2
2. RHYME.DLL holds Cap1, Cap2 and Cap3
3. REASON.DLL holds Cap1 and Cap2.

Then the load fails because RHYME.DLL cannot load REASON.DLL according to rule 2b.

Example 2 Also assume that:

1. PLOT.EXE holds Cap1 and Cap2
2. RHYME.DLL holds Cap1, Cap2 and Cap3
3. REASON.DLL holds Cap1, Cap2, Cap3 and Cap4.

Then the load succeeds; however RHYME.DLL cannot acquire the Cap4 capability held by REASON.DLL, and PLOT.EXE cannot acquire the Cap3 capability held by RHYME.DLL due to rule 1.

Examples for dynamically loaded DLLs

Assume that the program PLOT.EXE dynamically loads the library RHYME.DLL and the library RHYME.DLL then dynamically loads the library REASON.DLL.

Example 1 Also assume that:

1. PLOT.EXE holds Cap1 and Cap2
2. RHYME.DLL holds Cap1, Cap2 and Cap3
3. REASON.DLL holds Cap1 and Cap2.

The load succeeds because PLOT.EXE can load RHYME.DLL and REASON.DLL. You should note that the loading executable is the process PLOT.EXE and **not** the library RHYME.DLL, because the `<tt style="font-family:monospace;">RLibrary::Load()</tt>` request that the loader processes is sent by the process PLOT.EXE. The fact that the call is within RHYME.DLL is irrelevant: once loaded the code from RHYME.DLL is run with the same capability set as PLOT.EXE (rule 1).

Example 2 Also assume that:

1. PLOT.EXE holds Cap1 and Cap2

2. RHYME.DLL holds Cap1, Cap2 and Cap3

3. REASON.DLL holds Cap1, Cap2 and Cap4.

Then the load succeeds because PLOT.EXE can load RHYME.DLL and REASON.DLL. Because of rule 1, the code loaded from RHYME.DLL and REASON.DLL will be run with the same capability set as PLOT.EXE - that is Cap1 and Cap2.

Client-server

The servers that make up the TCE police incoming requests from their clients to ensure that those clients hold the required capabilities. For example, if a client asks the ETEL server to make a phone call, ETEL checks that the client has the *network services* capability.

An alternative approach to restricting access that is sometimes used in security architectures is a check based on the caller's identity, using, for example, an access control list. The approach used by Symbian OS platform security is preferable, because adding a new requester (for example a new client of ETEL) does not impact the security policy of the enforcer, ETEL.

A server typically offers many functions to its clients, and each of these functions can require a different set of different capabilities, or none at all. The capabilities required may also differ according to the data the client passes to the server - for example, the file server's function to open a file does not require any capabilities if the client wants to open its own files, but requires a system capability if the client wants to open a file that it does not own.

This security check does not affect the way in which IPC messages are formatted. It would be pointless to require a client to pass its capabilities to the server directly, since a badly behaved client could lie about them. Instead, the kernel adds client capabilities to the IPC message as it transfers the call from the client to the server side.

Figure 8.3 shows an application ODE.EXE that wants to dial a phone number by invoking `RCall::Dial()`. This method is just a front-end to a client-server IPC communication.

The kernel receives the IPC message and adds ODE's capabilities to it before dispatching it to the ETEL server. ETEL can trust it to do this, since the kernel is part of the TCB.

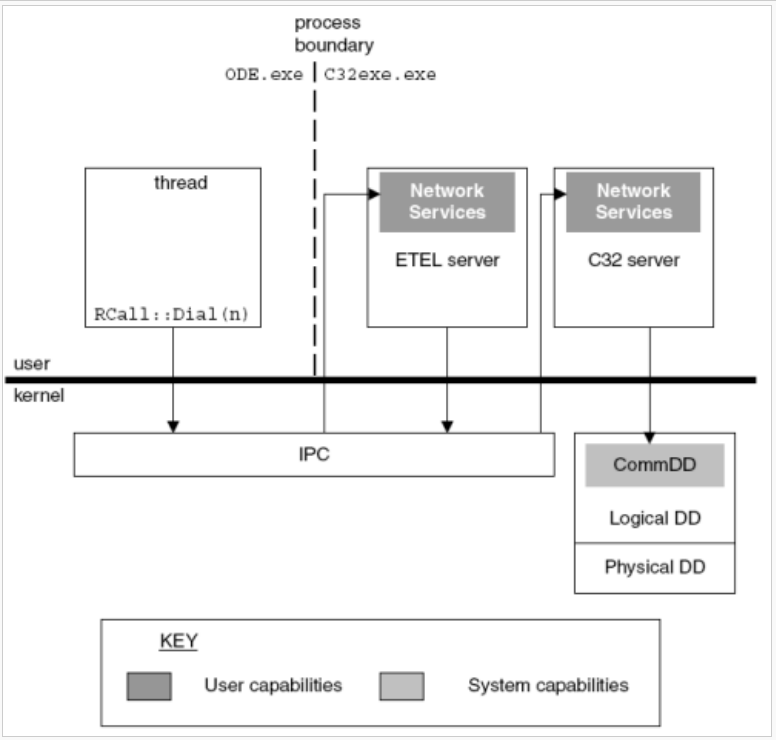

Figure 8.3 *Runtime capability checking*

ETEL can access ODE's capabilities via the `RMessage2` class. It verifies that ODE has the network services capability before accepting the request. Several classes exist to help servers check access to their services:

- The server should derive from `CPolicyServer`
- Mapping of IPC messages and security policies are defined by using `CPolicyServer::TPolicy`
- Security policies are defined by using `TPolicyElement` which provides verification services such as:

```
IMPORT_C TBool CheckPolicy (RMessagePtr2 a MsgPtr, const char* aDiagnostic=0) const
```

These classes hide the format used to store capabilities. They are a useful abstraction guaranteeing binary compatibility to developers by isolating them from changes in the underlying internal format.

Now suppose that ODE.EXE does not have the network services capability. What happens if ODE decides to bypass ETEL by talking to C32 directly? This will not work - C32, knowing that some requests can come from external processes, does its own policing.

What will happen if ODE decided to talk to the base band device drivers directly? This still will not work: these device drivers cannot be used without the system capability CommDD.

Or perhaps you might think that you could write your own device driver to access the hardware. That will not work either - device drivers run within the same process space as the kernel, and the rules described in Section 8.3.1 would require this device driver implementation to have at least the same process set of capabilities as the kernel itself.

The key points that you should bear in mind from this discussion are:

- Capabilities are **only worth checking** when a process boundary could be **crossed**
- The kernel is the **trusted intermediary** between a client and a server.

Device drivers

Device drivers run inside the kernel and so have access to all the resources of the mobile phone, without restriction. They are

implicitly part of the TCB. Because of this, it is very important that drivers check the capabilities of user-side processes and that they protect themselves against bad parameters passed in by a client. For example, if a malicious client passes a pointer into another process or even the kernel, and the device driver does not check that the client has the correct permissions to access that memory, then the client is able to bypass platform security and fool the driver into returning secret data.

The general rules for device driver writers are:

1. Check that the client has sufficient capabilities to access the functionality provided by the driver
2. Do not trust any parameters passed in by the client; check that all parameters are valid
3. Do not use pointers passed by the client directly - always use the kernel functions that access the memory using the client's permissions (kumemget(), kumemput(), `Kern::ThreadDesRead()` and so on)
4. Device driver channels cannot be passed between processes.

For device drivers that are split into LDD and PDD parts, the recommended practice is to perform all client capability checking at the LDD level. This ensures the policy is provided consistently and securely across all physical driver implementations.

Data caging

Concept

The capability model is the most fundamental concept of platform security. Data caging, another key concept, is much more specific: it is about file access control. As I said at the beginning of this chapter, phones are single-user devices, so the emphasis is not on controlling the user's access to files, but on controlling the file access of the various processes running on her phone.

One point to bear in mind is that, compared to a typical desktop environment, phones are resource-constrained - they may have limited storage space, slow processors and limited battery life. It is vital that platform security does not reduce the speed of file access, or increase power consumption or the size of file system itself.

The solution that we came up with was to move away from traditional access control lists, and to implement a fixed access control policy that can be fully described by the following sentence:

Rule 3. The access rules of a file are entirely determined by its directory path, regardless of the drive.

We identified four different sets of access rules, which we represent by four separate directory hierarchies under the root \.

sys

Only TCB processes can read and write files under this directory. This directory tree contains data vital to the integrity of the platform, such as executables.

resource

All processes can read files in this directory, but only TCB processes can write to them. This directory mostly contains bitmaps, fonts and help files that are not expected to change once installed.

private

The file server provides all programs with a private sub-directory under \private, regardless of their level of trust. Only the appropriate process, TCB processes and the backup server can read and write files in a process's private directory. Other processes may neither read nor write.

All other root files and directories

There is no restriction on what a process can do to a file stored in any other directory, including the root directory itself. These are all completely public spaces.

Implementation

Program SID

Each program must be assigned a unique identifier, called the secure identifier or SID, so that the kernel can provide a private space for each program. At ROM-build time and install time, the system will guarantee that each SID is uniquely associated with one executable. SIDs are assigned at compile time and specified in MMP files. If the MMP file does not define a SID, then the tool chain will use UID3 as the SID. If the program does not have a UID3, then the value of `KNullUid` is assigned. As a consequence, all processes with no SID and a null UID will share the same private directory.

Capabilities and file access

As well as the SID, there are two capabilities that we created to control access to data-caged directories:

1. TCB - grants write access to executables and shared read-only resources
2. AllFiles - grants read access to the entire file system; grants write access to other processes' private directories.

The access rules are summarized in the following table:

	Capability required to:	
	Read	Write
\resource	none	TCB
\sys	AllFiles	TCB
\private\<ownSID>	none	none
\private\<other>	AllFiles	AllFiles
\<other>	none	none

It may sound strange that TCB processes also need the AllFiles capability to be able to read under \sys. This decision was made for the following reasons:

- Capabilities should be fully orthogonal to each other: one should not be a subset of another. This is to eliminate the use of OR in capability checking, which creates ambiguity about what a capability really grants
- We identified a need to provide a capability to grant read access to the entire file system without implying that write access to \sys had to be granted: this reflects the distinction that has been made between TCB and TCE processes.

The role of the file server and the loader

All executables are now stored under \sys\bin. The file server enforces the rule that only TCB processes can write into this directory, to ensure that executables cannot be modified. Non-TCB processes cannot inject new code into existing executables and cannot change the capability set. This is a powerful way of ensuring the integrity of executables without requiring that each executable has to be signed and its signature verified every time it is loaded. If executables are installed in directories other than \sys\bin, then this is harmless, as the loader will refuse to load them.

This means that normal processes can no longer scan for executables, as they will not have the permissions to do so. The path of an executable is therefore irrelevant, and only the drive, name and type of file are important. This gives Symbian OS the flexibility to organize the storage of its executables at it sees fit without breaking backwards compatibility. In Symbian OS v9, all executables are stored in \sys\bin; there are no subdirectories. It is even more important than before to choose a *good* name for your executables, for example by prefixing them with your company name. If any clash is identified at install time, the software installer will indicate the conflict to the user and cancel the installation. This may seem harsh, but it is a very effective measure against Trojan horse attacks in which a malicious file tries to be seen as a legitimate one. Another side effect of this decision is that loading is faster, since we have reduced the number of directories to scan and simplified the algorithm used to decide which file to load. See Chapter 10, *The Loader*, if you want to find out more about file loading.

What about installed binaries on removable media - including those that are subst-ed as internal drives? In this case the path <drive>:\sys\bin cannot be deemed to be secure, because the removable medium might have been removed and altered elsewhere. Some tamper evidence is needed to detect whether the contents of a binary file have been changed since they were known to be safe - which was when they were installed. To do this when it first installs an executable onto a removable medium, the Symbian OS installer takes the additional step of computing a secure hash of the binary and storing this in the tamper-proof \sys directory on the internal drive.

Subsequent attempts to load the binary from that medium will fail if a second hash computed at load time does not match the one stored in the internal \sys directory, or if the internal hash does not exist at all.

Sharing files between processes

Symbian OS has a variety of ways of allowing processes to share data, including publish and subscribe, DBMS (Symbian's relational database) and the central repository (Symbian's service to share persistent settings). These methods, and others, have their uses, but nevertheless we still need a means for one process to share a file with another process under controlled conditions. For example, a messaging application might wish to launch a viewer application on a message attachment, but without revealing all other attachments to that viewer process, or revealing the attachment to all other process. EKA2 gives us the basic building blocks, by providing a way of sharing handles across processes, and the file server supports this feature for file

handles.

The process that owns the file opens it in a mode that cannot be changed by the process receiving the file handle without the file server rechecking that file's access policy against the receiving process's credentials.

The receiving process gets access to a shared session and file handles, but it does not get access to the shared file's parent directory or to other files in the same parent directory.

If you want to share files like this, you should be aware that, in the owning process you should open a file server session specifically for file sharing, open any files you wish to share and no others, and then close the session as soon as it is no longer needed (that is, once the receiving process has done what it needs to with the shared files). This is because the session handle is shared, along with the file handle, which means that any other files opened by the owning process in that session may be accessible to the receiving process, which could then increment the file handle numbers and gain access to other files. **Not conforming to this rule is a security hole.**

For more information on shared file handles, please refer to Chapter 9, *The File Server*.

Summary

In this chapter, I have introduced you to the key concepts in the Symbian OS implementation of platform security. Although I have not fully explored the subject, I hope that I have demonstrated how the kernel, loader and file server have been designed to play their part in making Symbian OS a secure platform. I have shown that this support is provided independently of cryptographic and authentication mechanisms to reduce the impact on the performance of the system and dependency upon those mechanisms:

- Capabilities are used to associate permissions to a program independent of the origin of the program
- Capabilities are used to prevent a program from loading a library that could compromise it.

Finally, I have discussed the file server and its role in data caging. I have shown that data caging provides safe storage for binaries and sensitive data, thus keeping them out of the reach of badly written or malicious code.