Chapter 12
Security Middleware for Mobile Applications

Bart De Win, ***DistriNet, Katholieke Universiteit Leuven, Belgium***
Tom Goovaerts, ***DistriNet, Katholieke Universiteit Leuven, Belgium***
Wouter Joosen, ***DistriNet, Katholieke Universiteit Leuven, Belgium***
Pieter Philippaerts, ***DistriNet, Katholieke Universiteit Leuven, Belgium***
Frank Piessens, ***DistriNet, Katholieke Universiteit Leuven, Belgium***
Yves Younan, ***DistriNet, Katholieke Universiteit Leuven, Belgium***

12.1 Introduction

Over the last decade the popularity of mobile devices has increased enormously. Initially, personal managers and mobile phones were designed as closed, dedicated devices. More and more, these devices have evolved into general purpose instruments that can be extended at user's will (a.o. via proper software development kits). This has lead to the current generation of smartphones and full-blown personal information management systems. At the same time, the information managed by the devices has evolved from limited and personal to general purpose and business-centric and, consequently, they constitute a core component of daily life.

These evolutions have had a significant impact on the security and privacy features of these devices. While rather simple, low-protected security models were provided initially, the current devices have evolved into natural extensions of personal computing platforms offering advanced, fine-grained data and software protection. Compared to personal desktops, however, the big challenge is due to the limited hardware protection models (such as data protection in memory) and dito computational resources that are available for the software security measures to build on. Consequently, the protection models of these devices have always been more restricted and targeted towards a specific setting. In this context, security middleware is to be interpreted as a broad category of security enhancements for applications (and their data) on mobile devices with limited capabilities.

In this chapter, the state-of-the-art in software protection for mobile devices is discussed. First, the security characteristics of mobile devices (as opposed to regular desktop systems) are elaborated upon by eliciting and illustrating specific types of threats. This should help in understanding and appreciating the particular difficulties of these platforms and allow one to draw connections between related problems on specific devices. Second, in the wide range of protective measures for mobile devices, we focus on two recent security enhancements that address some of these threats and, hence, improve the protection of applications on these devices: execution memory protection and security by contract. These techniques are comple-

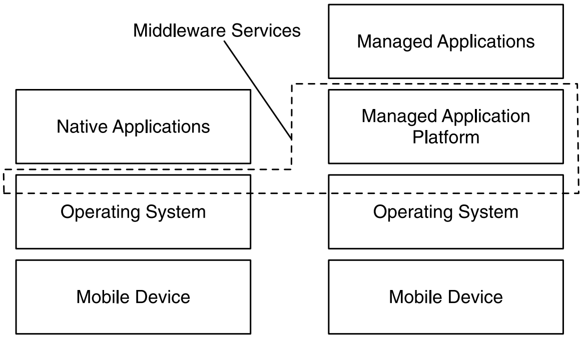

Fig. 12.1 Mobile application platforms: native platforms (left) and managed platforms (right)

mentary in the sense that the first technique focuses on applications running on native platforms, while the latter improves security properties for managed platforms. Besides motivating and discussing the general approach of these techniques, their usefulness for mobile devices in particular is highlighted as well.

The structure of the rest of this chapter is as follows. In Sect. 12.2, the security landscape for mobile devices is discussed with a strong focus on general security threats. Additionally, the characteristics of the security architectures of these platforms is briefly discussed, but an exhaustive comparison is out of the scope of this chapter. Sections 12.3 and 12.4 elaborate on two specific techniques, execution memory protection and security by contract respectively. Section 12.5 concludes and provides some ideas for future work.

12.2 The Security Landscape for Mobile Devices

12.2.1 Overview of Mobile Device Platforms

Since mobile devices are becoming small but powerful general-purpose computers, mobile platforms are getting more advanced and they are adopting many of the features found on typical desktop platforms. Mobile software platforms can be divided in two important classes: native platforms and managed platforms (see Fig. 12.1).

In native platforms, applications are compiled into machine code and are executed directly on the processor of the device. Native applications make efficient use of the limited resources that are available, and therefore this has been the traditional approach of developing mobile applications. The most important native platforms in the market today are Symbian (version 9), Windows Mobile (version 5 and 6), Garnet OS (previously known as Palm OS version 5), various flavors of linux and, recently, the version of Mac OS X found on Apple’s iPhone.

In managed platforms, applications are compiled into bytecode and executed on a virtual machine instead of on the real device. As a consequence, applications are portable to any device that has a native implementation of the virtual machine. Because of the variety of mobile devices, managed platforms have become very important in the mobile development space. The two best known managed platforms for mobile devices are the .NET Compact Framework [866] and the Mobile Information Device Profile (MIDP) [785] of Java Micro Edition (Java ME). Java ME consists of the Connected Device Configuration (CDC) [784] for more powerful devices and the Connected Limited Device Configuration (CLDC) [783] for limited devices. MIDP is the profile that specifies Java ME runtime within CLDC for the mobile devices that fall within the scope of this chapter.

As indicated in Fig. 12.1, both native and managed platforms include middleware services for developing mobile applications. The virtual machine of a managed platform is just one example of such a middleware service. Others are offered to developers either explicitly through APIs or transparently in the form of compiler extensions.

A managed platform always needs to run on some native underlying platform. This means that it is possible that native applications and managed applications coexist on the same device. Usually, managed platforms solely define the middleware layer between the applications and the native platform. Java ME and the .NET Compact Framework are examples of this approach. Other platforms define both the middleware layer and the native platform that is used. In these platforms, it is usually the case that only managed applications are supported. An example of such a combined approach is Google's Android Platform, which defines both a Java-based middleware layer and an underlying linux-based operating system.

12.2.2 Protection of Mobile Platforms

Just as any other platform that executes general-purpose applications, mobile platforms need security mechanisms to protect the resources on the devices. Two important techniques can be distinguished: memory management and software protection.

Native platforms need to ensure that applications cannot access memory that belongs to other applications or to the platform itself. This separation is guaranteed by *memory management*. Some of the older mobile device platforms such as Palm OS have no form of memory protection at all, which makes these platforms inherently insecure. However, most modern native platforms (Windows Mobile, Symbian) do provide basic memory protection. One of the important strengths of managed platforms (.NET Compact Framework, Java ME) is that they have a strong memory protection model because applications have no direct access to the memory and because these platforms guarantee type safety.

Other resources on the device such as personal data, wireless networking, or SMS messages are accessed via APIs and are protected by the *security architecture* of the platform. Over the years, security concerns on mobile devices have gotten more attention and the security architectures in the latest versions of current mainstream platforms such as Symbian, Java ME and the .NET Compact Framework are be-

Table 12.1 A summary of the security issues

Issue	Description
I1	Important security mechanisms found on full-scaled platforms are omitted
I2	The developer has very little options in customizing security mechanisms
I3	APIs can be protected but often in an all-or-nothing way
I4	It is hard to securely store sensitive data
I5	A mobile device is subject to different kinds of threats

coming more powerful. These security architectures mainly focus on reducing the privileges under which applications execute. Although some security architectures are more advanced than others, the underlying security model they implement is similar. The platform APIs are divided in two or more trust levels and each application is assigned a certain level of trust based on credentials that are shipped with the application (typically a signature from the developer). The security architecture enforces that applications cannot invoke APIs of a higher trust level than their own. Untrusted applications are prohibited or are executed in a sandbox to ensure that they can only use a minimal and tightly-controlled set of resources.

Sometimes, for instance in the case of MIDP and Symbian, the platform realizes the aforementioned model by an underlying permission-based access control mechanism. Instead of grouping API operations in trust domains, operations have one or more required permissions. Domains are then defined indirectly in terms of permissions and applications that belong to a certain domain get the permissions of the domain. This approach supports more fine-grained specification of granted behavior (by giving an application an additional permission). Symbian and MIDP also support the dynamic assignment of permissions based on a prompt with the user. These dynamic permissions are only valid for a single execution of an operation or until the application quits, based on a policy that is included with the platform.

12.2.3 Security Issues

It is understandable that mobile platforms have simpler security models due to the limited resources that are available to them. However, users and developers must realize that on these platforms there are a number of important security issues. In this section, a number of general security issues are discussed that are found in today's mainstream mobile device platforms. Table 12.1 gives an overview of these issues. In the rest of this section, each of these issues is discussed in more detail.

I1: Important Security Mechanisms are Omitted. Many mobile platforms are effectively stripped-down versions of full-scale platforms that run on normal desktop computers, for instance: mobile linux-based devices, the mobile vs. the regular version of Mac OS X, Java ME vs. Java SE and the .NET Compact Framework vs. the full .NET Framework. The full versions of these platforms have extensive security architectures and offer many security mechanisms and countermeasures.

When comparing the scaled-down versions of these platforms with their full-sized counterparts, it is notable that many of these security mechanisms are omitted or replaced by much less powerful variants because of resource constraints. However, since mobile devices have become general-purpose computing devices with a high degree of connectivity, they are susceptible to many of the same threats as found on full-scaled platforms. Two examples are given below.

Many security problems on full native platforms are caused by poorly programmed software that leads to buffer overflows. By feeding a vulnerable application a carefully crafted input, an attacker can overwrite certain memory locations in the system. This enables the attacker to execute arbitrary code, potentially resulting in system compromise. Since these kinds of attacks are among the oldest and best known, most modern operating systems such as Windows Vista, Mac OS X and linux contain at least some countermeasures to prevent buffer overflows from occurring, or making them harder to exploit. Mobile device platforms on the other hand, have no support for any of these countermeasures. The recent hacking of Apple's iPhone for unlocking purposes illustrates that buffer overflow-based attacks are not unrealistic on mobile devices.

One of the important security mechanisms that are found on full-scaled versions of managed platforms is access control based on stack inspection [288], in which the effective permissions of a subject that invokes an operation is determined by inspecting the execution context. By doing so, an untrusted application cannot gain more privileges by accessing a resource indirectly via a trusted application or module. Although mobile devices run untrusted applications and dynamically downloaded code, the MIDP and the .NET Compact Framework do no support stack inspection.

I2: Flexibility and Extensibility are Limited. The security architectures on mobile platforms are deliberately kept simple. One of the simplifications with respect to full-scaled platforms is that the flexibility and extensibility of the security architectures are limited. On full-scale platforms, the user or the administrator has the flexibility of changing the security policies that are enforced to reflect evolving or unanticipated security requirements. Mobile platforms typically use hard-coded policies or policies that are specified once by the manufacturer of the device. Moreover, full-scale platforms allow the extension of the security architecture to support the enforcement of different or more advanced security policies, whereas the security architectures on mobile platforms are not extensible at all. If developers are faced with the limits of an inflexible security architecture, they need to fall back to implementing the necessary security mechanisms in their applications, which is clearly not desirable.

In Symbian, for instance, applications used to be able to write and read all data on the device, including data from other applications or from the operating system. Since version 9, Symbian includes access control for data (this is called *data caging*) that limits the directories in which an application can read and write. The policy that states which directories are accessible is hardcoded and the only way that applications can be given access to more data is by giving them access to all data on the system.

A good example of limited extensibility can be found in Java ME's MIDP profile. The full-scaled version of Java can be extended with customized SecurityManagers or LoginManagers in order to change the way in which security decisions are made or in which users are authenticated. None of these features can be found in the security architecture of MIDP. Moreover, the store of trusted certificates on which the access control model is based, cannot be changed. Finally, it is impossible for developers to extend permissions in MIDP.

I3: Access Control Towards APIs is Coarse-Grained. Apart from some exceptions (Palm OS), most mobile platforms are able to limit access to individual API operations (see Sect. 12.2.2). Applications are placed in a trust domain based on a digital signature. Often, users can decide to trust an unsigned application anyway. Applications in a low trust domain are executed in a sandbox that restricts access to sensitive API operations (e.g., only applications in the mostly trusted domain can send SMS messages). An example of this approach is Windows Mobile, which defines two security domains: *trusted* applications have full system privileges and *normal* applications have no access to sensitive operations on the API. The operating system maintains a privileged and an unprivileged keystore. A security policy configures the mapping of privileged, unprivileged and unsigned applications to the trusted or normal domains. This mapping can be influenced by a user prompt.

For the user that wants to run a downloaded application, this model is too coarse-grained. Either the application is trusted and can do anything (which requires blind trust in the developer of the application), or the application is not trusted and it cannot invoke any sensitive operations. Moreover, the decision on whether to trust an application or not solely depends on the signature of the application.

Some platforms (MIDP and Symbian) have a more fine-grained security model. The assignment of applications to security domains is still based on code signing, but these platforms are able to identify, assign and revoke individual permissions. Because of this reason, these platforms can offer support for dynamically raising the permissions of an application at runtime based on prompting the user. For instance, in MIDP, the user can be prompted when an untrusted applications wants to make a network connection. The user can then grant the application this permission temporarily (once or until the application quits).

I4: Privacy Is Poorly Protected. By now, many mobile platforms have included support for access control to APIs. However, the protection of sensitive data on the device is often very poor.

Palm OS and Windows Mobile have no way of letting a user limit application access to his/her data. MIDP is unable to protect data on the device except for data that is accessed via the PIM API: it is possible to deny read and write access to contacts, meetings and todo's. Symbian supports a simple form of access control towards data (this is called data caging). More specifically, applications have their own `/private` directory that is readable and writable, system files `/sys` cannot be read or written and there is a publicly readable directory `/resources` that can only be written to by the system. However, all other directories on the file system are fully accessible by all applications.

I5: Mobile Devices Have Different Threats. In many respects, mobile devices are playing catch-up with desktop computers: they have gotten powerful processors, more memory and they can be always-online. This evolution has undoubtedly made mobile devices vulnerable to the same kinds of threats than desktop systems. However, it is important to realize that due to their specific nature, there are important differences in the kinds of threats to these devices.

First of all, many known threats to normal computers (such as self-propagating worms) target server processes that offer a certain network service. On mobile devices, it is much less likely that there are any servers that can be targeted. The only kind of server processes that are commonly found in this setting are the short-distance wireless communication (Bluetooth and Infrared) processes of the operating system. Therefore, the focus of the threats on mobile devices shifts completely from servers to applications.

Secondly, the high degree of connectivity (WiFi, wireless broadband, cellular networks, Bluetooth, Infrared) in combination with the mobility of the device increase the attack surface in comparison with desktop computers and opens a whole new family of threats. For instance, there have been exploits for Symbian that use SMS or MMS messages as a propagation vector [269, 270], or that allowed an attacker to take over complete control of the device via Bluetooth [268], allowing him to initiate calls and send SMS messages.

A third category of threats on mobile devices are denial-of-service threats. Because of resource limitations, mobile device platforms often very poor process management models. Therefore, simply feeding a badly written application with corrupt data that causes it to crash or hang, can often bring the whole system to a halt, forcing the user to reset its device. The most extreme example is Palm OS, which has no support for multitasking and cannot stop unresponsive applications at all.

12.3 Protection for Native Platforms: Memory Protection

As discussed in Sect. 12.2.3, issue I1, mobile devices will often run stripped down versions of desktop operating systems, making them vulnerable to the same type of threats. Applications that run natively on a mobile device, are often written in C or C++ and as a result are vulnerable to buffer overflows and similar vulnerabilities which could allow an attacker to execute arbitrary code on the device.

A buffer overflow occurs when a program writes past the bounds of an object in memory and starts to overwrite adjacent objects. By overwriting memory addresses (e.g., stored code pointers), an attacker may be able to control the execution flow of a program. This could allow the attacker to execute arbitrary code with the privileges of the application that is being executed. A typical type of buffer overflow is the stack-based buffer overflow, where the program will overflow past the bounds of a stack-allocated array of characters. The stack is used to facilitate the execution of functions and recursion: it contains the local variables of each function, together with the return address (the address of the instruction at which execution must resume once the function has terminated) and the value of several registers that must preserve their values after the function has finished executing. If a buffer overflow occurs in a local variable, the attacker can overwrite the return address. When the function terminates, it will transfer control to the instruction at the return address. If attackers make the return address point to their injected code (provided as data input to the program), they can force the program to execute arbitrary code.

In August 2006, a number of vulnerabilities [646] were discovered in LibTIFF.[1] LibTIFF is used in a number of desktop operating systems, like Linux and Mac OS X. However, it is also used on the Apple iPhone, where this vulnerability was widely exploited by users of the iPhone [588] to perform a "Jailbreak".[2] This vulnerability can be triggered in both MobileMail (the iPhone mail client) and MobileSafari (the iPhone web browser) and as a result is remotely exploitable by letting the user browse to a site containing a specific TIFF file or by emailing a TIFF file to the user.

This vulnerability was also present on another mobile device: the Sony PlayStation Portable, where it was also exploited to allow behavior not condoned by the manufacturer. In this case, the vulnerability was used to gain more permissions to allow users to run homebrew[3] games.

These two examples show that software originally designed for desktop environments is being ported widely to these new devices, resulting in the same types of vulnerabilities being present in these devices. As more and more of these devices enter into the market, more of these types of vulnerabilities will be discovered and exploited.

12.3.1 Existing Countermeasures

Many countermeasures exist on desktop operating systems that can prevent these kind of attacks. As such, they can help in providing protection against issues I1 and I5 as discussed in Sect. 12.2.3. An extensive survey can be found in [891]. Very few countermeasures have been ported to mobile devices however. In this section the few countermeasures that do exist are described and their shortcomings are discussed. In the next section, other existing countermeasures and issues that may exist when porting them to mobile devices are discussed.

Non-executable Memory. These countermeasures will make data memory non-executable. Most operating systems divide process memory into at least a code (also called the text) and data segment and will mark the code segment as read-only, preventing a program from modifying code that has been loaded from disk into this

[1] LibTIFF is a library for reading and writing TIFF files, a popular image format.

[2] By default, it is not possible for users to install additional native applications on the iPhone. The term Jailbreaking refers to the escaping of these limitations, allowing users to gain full control of the device.

[3] Homebrew games are games which are typically produced by consumers and are generally not authorized (or digitally signed) by the manufacturer of the product, resulting in the need to circumvent security restrictions on the device before they can be played.

segment (unless the program explicitly requests write permissions for the memory region). As such attackers have to inject their code into the data segment of the application. As most applications do not require executable data segments as all their code will be in the code segment, some countermeasures mark this memory as non-executable, which will make it harder for an attacker to inject code into a running application. A major disadvantage of this approach is that an attacker could use a code injection attack to execute existing code. One type of such an attack is called a return-into-libc attack [869]. Instead of injecting code on the stack and then pointing the return address to this code, the desired parameters are placed on the stack and the return address is pointed to existing code (a simple example is to call the libc wrapper for the *system()* system call and to pass it an executable that will execute the attacker's code as an argument). This is also the attack used in the original iPhone exploit: the stack is marked as non-executable so a return-into-libc attack was performed which would execute a number of library functions to gain the desired results. A later exploit uses the stack-based buffer overflow to perform a return-into-libc attack, which copies the injected code onto the heap and then transfers control flow there.

Stack Cookies. The observation that attackers usually tried to overwrite the return address when exploiting a buffer overflow led to a string of countermeasures that were designed to protect the return address. One of the earliest examples of this type of protection is the canary-based countermeasure [195]. These countermeasures protect the return address by placing a value before it on the stack that must remain unchanged during program execution. Upon entering a function the canary is placed on the stack below the return address. When the function is done with executing, the canary stored on the stack will be compared to the original canary. If the stack-stored canary has changed an overflow has occurred and the program can be terminated. A canary can be a random number, or a string that is hard to replicate when exploiting a buffer overflow (e.g., a NULL byte). StackGuard [196, 195] was the first countermeasure to use canaries to offer protection against stack-based buffer overflows, however attackers soon discovered a way of bypassing it using indirect pointer overwriting. Attackers would overwrite a local pointer in a function and make it point to a target location, when the local pointer is dereferenced for writing, the target location is overwritten without modifying the canary. Propolice [259] is an extension of StackGuard, it fixes these type of attacks by reordering the stack frame so that buffers can no longer overwrite pointers in a function. These two countermeasures have been extremely popular: Propolice has been integrated into the GNU C Compiler and a similar countermeasure has made it's way into Visual Studio's compiler [115, 332]. The countermeasure which was integrated into Visual Studio is also used on mobile devices running Windows CE 6.

12.3.2 Applicability of Desktop Countermeasures

Many of the countermeasures that are currently in use on desktop systems can be ported to embedded devices. However, due to limited memory and processing power, efficiency becomes a more important concern on these devices. Many countermeasures are also designed with a specific architecture in mind, making it hard to apply them in a mobile setting. Several other limitations in the architectures on mobile devices may also be important issues when trying to port countermeasures to these architectures. For example, many architectures on mobile devices have no support for paging, making it hard to implement a countermeasure like address space layout randomization (ASLR)on these devices. In this section different countermeasures that exist for desktop operating systems are discussed and the advantages, disadvantages and possible problems if they were to be implemented on mobile devices are examined. While the general techniques of these countermeasures are discussed, porting a specific implementation of a technique discussed here may require significant additional effort however.

Safe Languages. Safe languages exist in all types and forms on both desktop operating systems, many of these languages will prevent memory safety problems by removing control from the programmer (e.g., Java removes pointers from the programmer's control), by inserting extra checks or a combination of both. Many such languages exist, however since this section focusses on memory errors, only languages that stay as close as possible to C and C++ are examined. These safe languages are referred to as safe dialects of C. Some dialects [615] will only need minimal programmer intervention to compile programs, while others [424, 572] require substantial modification. Others [474] severely restrict the C language to a subset to make it safer or will prevent behavior that the C standard marks as undefined [637].

Some of these languages have a relatively high overhead, however most have acceptable overhead. The main disadvantage of using these languages on mobile devices is the same as for desktop devices: programmers must learn and port their existing code to this new language.

Boundschecking. Bounds checkers provide extensive protection against exploitation of buffer overflows: they check array indexation and pointer arithmetic to ensure that they do not attempt to write to or read from a location outside of the space allocated for them. Two important techniques are used to perform traditional bounds checking: adding bounds information to all pointers and adding bounds information for objects. In the first technique, pointers contain extra information about the object they are referring to. In the second technique the objects themselves will contain the extra information. Boundscheckers that use either technique will generally suffer from a high computational and memory overhead, making them less suited for mobile devices.

Obfuscation of Memory Addresses. Memory-obfuscation countermeasures use an approach that is closely related to stack cookies: their approach is also based on random numbers. These random numbers are used to 'encrypt' specific data in memory and to decrypt it before using it in an execution. These approaches are currently used for obfuscating pointers (XOR with a secret random value) while in memory [198]. When the pointer is later used in an instruction it is first decrypted in a register (the decrypted value is never stored in memory). If an attacker attempts to overwrite the pointer with a new value, it will have the wrong value when decrypted. This will most likely cause the program to crash. A problem with this approach is that XOR encryption is bytewise encryption. If an attacker only needs to overwrite 1 or 2 bytes instead of the entire pointer, then the chances of guessing the pointer correctly vastly improve (from 1 in 4 billion to 1 in 65000) [15]. If the attacker is able to control a relatively large amount of memory (e.g., with a buffer overflow), then the chances of a successful attack increase even more. While it is possible to use better encryption, it would likely be prohibitively expensive since every pointer needs to be encrypted and decrypted this way. While the prototype implementation in [198] is fairly efficient because of the increased use of registers by the modified gcc compiler, this type of protection could turn out to be expensive on other architectures.

Address Space Layout Randomization. ASLR is another approach that makes executing injected code harder. Most exploits expect the memory segments to always start at a specific known address. They will attempt to overwrite the return address of a function, or some other interesting address with an address that points into their own code. However for attackers to be able to point to their own code, they must know where in memory their code resides. If the base address is generated randomly when the program is executed, it is harder for the exploit to direct the execution-flow to its injected code because it does not know the address at which the injected code is loaded. Shacham et al. [745] examine limitations to the amount of randomness that such an approach can use.[4] Their paper also describes a guessing attack that can be used against programs that use forking as the forked applications are usually not rerandomized, which could allow an attacker to keep guessing by causing forks and then trying until the address is found. It may not be possible to implement this type of countermeasure on architectures which do not have support for paging. Architectures that have limited address space (e.g. 16-bit architectures), may also not be able to benefit from this approach.

Instruction Set Randomization. ISR [63, 446] is another technique that can be used to prevent the injection of attacker-specified code. Instruction set randomization prevents an attacker from injecting any foreign code into the application by encrypting instructions on a per process basis while they are in memory and decrypting them when they are needed for execution. Attackers are unable to guess the decryption key of the current process, so their instructions, after they've been decrypted, cause the wrong instructions to be executed. This prevents attackers from having the process execute their payload and has a large chance crashing the process due to an invalid instruction being executed. However if attackers are able to print out

[4] This limitation is due to address space limitations in 32-bit architectures: often countermeasure will limit randomness to a maximum amount of bits, which will be less than 32 bit, making guessing attacks a possibility.

specific locations in memory, they can bypass the countermeasure since the encryption key can often be derived from encrypted data (since most countermeasures will use XOR). Other attacks are described in [859]. The current implementations are proof of concept implementations and suffer from high overheads. However, a CPU could be designed which supports this kind of countermeasure. Given the security problems with this approach, a CPU supporting a stronger encryption than XOR, as described in [382], may be more desirable, which is clearly difficult to accomplish on mobile devices.

Separation and Replication of Information. Countermeasures that rely on separation or replication of information will try to replicate valuable control-flow information or will separate this information from regular data [893, 894, 62]. This makes it harder for an attacker to overwrite this information using an overflow. Some countermeasures will simply copy the return address from the stack to a separate stack and will compare it to or replace the return addresses on the regular stack before returning from a function. These countermeasures are easily bypassed using indirect pointer overwriting where an attacker overwrites a different memory location instead of the return address by using a pointer on the stack. More advanced techniques try to separate all control-flow data (like return addresses and pointers) from regular data, making it harder for an attacker to use an overflow to overwrite this type of data. These countermeasures could easily be applied to mobile architectures given their low performance overhead. Care must be taken however in environments where memory is scarce as these countermeasures will tend to have some memory overhead when replicating or separating the information.

Paging-Based Countermeasures. Paging-based countermeasures make use of the Virtual Memory Manager, which is present in most modern architectures. Memory is grouped in contiguous regions of fixed sizes (e.g., 4 Kb on Intel IA32) called pages. Virtual memory is an abstraction above the physical memory pages that are present in a computer system. It allows a system to address memory pages as if they are contiguous, even if they are stored on physical memory pages that are not. An example of this is the fact that every process in Linux starts at the same address in the virtual address space, even though physically this is not the case. Pages in most architectures can have specific permissions assigned to them: Read, Write and Execute. Many of the countermeasures in this section will make use of paging permissions or the fact that multiple virtual pages can be mapped onto the same physical page on Intel IA32. These countermeasures could be applied to mobile devices that have support for paging. However, they will require architectures with support for paging or for applying permissions to those pages. These may not always be available in an architecture running on a mobile device.

Execution Monitors. Execution monitors monitor specific security relevant events (like system calls) and perform specific actions based on what is monitored. Some monitors try to limit the damage a successful attack on a vulnerability could do to the underlying system by limiting the actions a program can perform. These are called policy enforcers. In general, enforcement is done through a reference monitor where an application's access to a specific resource[5] is regulated. They can be very coarse or very granular. An example of a coarse grained policy enforcer is one that ensures that a program executes it's system calls in a given order (these can be learned from previous runs of the program). If the program tries to execute a different system call, it will be denied. Attackers can however perform a mimicry attack, where they mimic the behavior of the original program, while only providing different arguments for the system call. The more granular a policy enforcer is, the harder a mimicry attack becomes: a very fine grained enforcer could force the attacker to only execute the existing code [1]. The more fine grained policy enforcers could suffer from higher overhead, which may be a problem in a mobile setting. While the more coarse grained enforcers will usually not suffer from as high overheads, they can easily be bypassed by a determined attacker.

Fault isolators are a second type of execution monitor: they ensure that certain parts of software do not cause a complete system (a program, a collection of programs, the operating system, ...) to fail. The most common way of providing fault isolation is by using address space separation. However, this causes expensive context switches to occur that incur a significant overhead during execution. Because the modules are in different address spaces, communication between the two modules also incurs a higher overhead. For that reason, software fault isolation [846] exists, which allows for better performance. Software fault isolation can be an efficient way of preventing errors from causing an entire system to fail, however it may be incomplete when faced with a determined attacker. Ensuring proper isolation even from an attacker may add additional overhead.

Hardened Libraries. Hardened libraries replace library functions with versions that contain extra checks. An example of these are libraries that offer safer string operations: more checks will be performed to ensure that the copy is within bounds, that the destination string is properly NULL terminated (something *strncpy* does not do if the string is too large). Other libraries check whether format strings contain '%n' in writable memory [711] (and will fail if they do) or will check to ensure that the amount of format specifiers are the same as the amount of arguments passed to the function [197]. Since these libraries operate at a higher level than most other countermeasures described in this section, applying them when programming applications for a mobile device should not pose any significant problems. As with their desktop variants, the programmer has to use these libraries correctly and consistently, possibly modifying existing code to make use of these libraries.

Runtime Taint Trackers. Taint tracking is an important type of countermeasure for web-based vulnerabilities. It is ideally suited for detecting cross-site scripting, SQL injection, command injection and other similar vulnerabilities in web applications [672, 875]. These taint trackers instrument the program to mark input as tainted.[6] If such input is used in a place where untainted data is expected (like an SQL query), an error is reported. Taint tracking can also be used to detect memory errors. In this case, the taint tracker will generate an error when an trusted memory

[5] The term resource is used in the broadest sense: a system call, a file, a hardware device, ...

[6] Tainted data is data which is untrusted, usually derived from input.

location (like a return address) has been modified by tainted data. One important limitation with these taint trackers is that they suffer from false positives. Such a false positive can occur when a tainted data is used in a place where untainted data is expected but is not actually vulnerable to attack. For example, if a format string is derived from tainted data and used as format specifier to *printf*, however a check has occurred to ensure that this tainted data is in fact benign. A taint tracker may report this as a vulnerabilty, while it is in fact safe code. These countermeasures will generally also suffer from a significant overhead, both in terms of performance and memory usage, which may not be acceptable in a mobile device.

12.3.3 Model-Based Countermeasure Design

An important issue with porting these desktop applications to embedded devices is the fact that many desktop countermeasures were designed with a specific architecture or operating system in mind. This can make porting more difficult and prone to being bypassed. When a countermeasure is ported, the countermeasure developer must ensure that the countermeasure can not easily be bypassed on the new platform. An example of such porting going wrong occurred when Microsoft ported the StackGuard countermeasure [195] from GCC to Visual Studio [116]. The specifics of the Windows operating system were not taken into account, which resulted in attackers being able to bypass the countermeasure by ignoring the return address and continuing to write on the stack until they overwrote the function pointers used for exception handling. Subsequently, an exception would be generated by the attackers and their injected code would be executed [522]. A possible way to prevent these types of problems when porting countermeasures to a new platform is to use machine model aided countermeasures [892].

Model-based countermeasure design builds a model of the execution environment of the program based on the memory locations and abstractions that influence the execution flow. This abstract high-level model contains memory locations and abstractions that can be used by an attacker to directly or indirectly influence the control flow of a particular application, supplemented with the locations that could lead to indirect overwriting of these memory locations. Finally, these are supplemented with contextual information: what these specific memory locations are used for at different places of the execution flow and what operations are performed on them. This machine model[7] allows a designer of countermeasures to view a platform in a more abstract way and as a result more effort can go into designing countermeasures rather than understanding obscure, possibly insignificant, platform details. It also allows a designer to take into account what the effects of a particular countermeasure are on a platform before having to implement it.

[7] This is model is a high level representation of a platform and should not be confused with an executable model of a specific program (e.g. a bytecode representation of a Java program).

12.4 Protection for Managed Platforms: Security by Contract

As explained in Sect. 12.2, the typical way security is enforced on a modern mobile device is by checking that an application is certified by some trusted third party. If the application is signed by a trusted third party, it is allowed to run. If the signature cannot be verified, the user is asked whether the application should be run or not. This signature-based system doesn't work well in the case of mobile code. The first problem is that the decision of allowing an application to run or not, is too difficult for a user to make. He would like to run an application as long as the application doesn't misbehave or doesn't violate some kind of policy. But he is in no position to know what the downloaded application exactly does, so he cannot make an educated decision to allow or disallow the execution of a program. A second problem is that certifying an application by a trusted third party is rather expensive. Many mobile application developers are small companies that do not have the resources to certify their applications. A third, and perhaps most damning, problem is that these digital signatures do not have a precise meaning in the context of security. They confer some degree of trust about the origin of the software, but they say nothing about how trustworthy the application is. Cases are already known where malware was signed by a commercial trusted third party [696]. This malware would have no problems passing through the mobile security architecture, without a user noticing anything.

Recent research [739, 224] has addressed these issues by working out a security-by-contract (SxC) paradigm for the development, deployment and execution of mobile applications. The key idea behind SxC is to have a system that can automatically prove that a mobile application will not try to access resources to which it *doesn't* have access, or that it doesn't try to abuse resources to which it *does* have (limited) access. SxC supports a number of different mechanisms to prove this claim. The application is allowed to run if one of these mechanisms can certify that the application will not violate the policy. As such, the SxC paradigm is particularly suited to address issues I2 and I3 as discussed in Sect. 12.2.3.

One of the characteristics of a managed platform is that it offers an environment where so-called ‘safe languages’ can be used to guarantee type soundness and memory safety. These two features make it much easier to prove certain security-related properties. Because proving properties about unknown code is of central importance in this new SxC paradigm, it works best in managed environments.

12.4.1 Policies and Contracts

Loosely speaking, a system policy is a set of rules to which an application must comply. These rules usually limit the access of an application to a specific part of the system API. For instance, there could be a set of rules to prohibit applications from accessing the network or to limit the access to the file system. These accesses are the *security-related events*. One could think of a policy as an upper-bound description of what applications are allowed to do.

Contracts are very similar, but instead of defining the upper-bound of what an application *can* do, it describes the upper-bound of what the application *will* do. It's the 'worst case' scenario of security-related behavior of an application. The contract is typically designed by the application developer and is shipped together with the application as metadata.

An example of a policy could be *"An application can use the GPS device, cannot place a phone call, and can send a maximum of 1000 bytes over the network"*. If an application arrives that comes with a contract that says *"This application will use the GPS device, and will send at most 200 bytes over the network"*, then the application should be allowed to run because it complies with the policy.

Policies are in essence the system view of the device's security requirements. This is why they are often called *system policies*. Likewise, contracts are the application view of the security requirements. They are also called *application contracts*.

12.4.2 Policy Enforcement

The main focus of an SxC system is to make sure that a mobile application will never violate the system policy. A number of different *enforcement technologies* can be used to ensure this, each with different benefits and drawbacks. Some of the most common techniques are discussed here.

Matching The basic idea behind policy and contract matching, is to automatically check whether an application contract is subset of the system policy. If the policy is the upper-bound of what the application *can* do, and the contract is the upper-bound of what the application *will* do, then having an application contract that is a subset of the policy means that the application will never do anything that violates the policy.

Different matching algorithms exist, ranging from simple algorithms like identical matching or hash-based matching, to more advanced implementations like simulation matching or language inclusion matching. Identical matchers compare the application contract and system policy on a byte-per-byte basis. If they are exactly the same, the contract and the policy are said to match; otherwise, the algorithm cannot guarantee that they match. Hash-based matching is a bit smarter than identical matching, but not by much. A hash-based matcher will break the contract and policy in smaller pieces, and will check that the different pieces of the contract are all present in the policy. This is normally done by generating a hash of every piece, and comparing the hashes of the contract with the hashes from the policy. A disadvantage of these two types of algorithms is that they only work for the most trivial cases of policy-contract matching. For instance, they cannot successfully match the simple policy *"An application cannot send more than 1000 bytes"* with a contract *"This application will not send more than 200 bytes"*. Fortunately, the more advanced forms of matching, like simulation matching or language inclusion, are able to successfully handle these cases.

Static analysis and proof-carrying code A second option is to automatically inspect the binary code of the application and to construct a proof that explains *why* the application will comply with the contract. This process is called 'static analysis'. However, static analysis is still a developing discipline and current implementations often need some kind of input from the application developer (e.g., code annotations). Furthermore, it is also a complex and slow process. Statically analyzing a non-trivial application requires a lot of computational power, which is not practical on a mobile device.

In proof-carrying code [614], the result of a static analysis (performed by an expert on a powerful computer) is stored and distributed along with the application. When the application arrives on the mobile device with this extra metadata, the problem of constructing a proof is reduced to verifying a proof. Since proof verification can be done more efficiently than proof generation, it is realistic to implement a proof-checker on a mobile device.

Inlining Another enforcement technology is policy inlining [255]. During the inlining process, the system goes through the application code and looks for calls to security-related events (SRE). When such a call is found, the system inserts calls to a monitoring component before and after the SRE. This monitoring component is a programmatic representation of the policy. It keeps track of the policy state and intervenes when an application is about to break the policy. As such, the application is made to comply with a contract that is equivalent to the system policy.

The biggest advantage of this technique is that it can be used on applications that are deployed without a contract or any other additional metadata. It can be used as a fall-back mechanism for when the other approaches fail and it can also ensure backwards compatibility. A disadvantage is that the application is modified during the inlining process, which might lead to subtle bugs. Also, the monitoring of the SREs comes with a performance hit. The performance hit is strongly dependent on the complexity of the policy being enforced. However, the decrease in performance is often relatively small.

Digital signatures Finally, it should be mentioned that the classical mobile security architecture, based on digital signatures, can also be modeled within an SxC system. A trusted third party could be used to certify that an application complies to a given policy. This trusted party would have a different public/private key pair for every different policy it certifies. An application developer would send his application to this trusted party for compliance certification with one of the trusted party's policies. When the application is found to comply with the policy, it gets signed with the key corresponding to that policy. The signature can then be stored in the application metadata.

Notice the subtle difference between the meaning of the digital signature in the SxC system and in the classical mobile security architecture. Both systems make use of the exact same mechanism, but on the SxC system, the signature tells more about the application than simply its origin. It certifies that the application will not violate a specific policy. It certifies that this application will not harm the

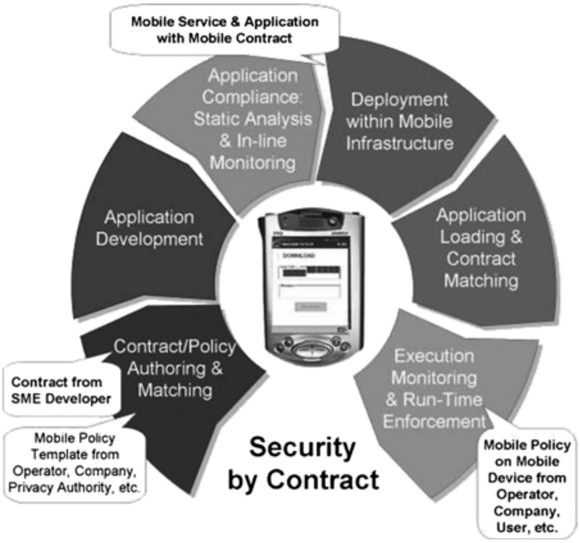
Mobile Service & Application with Mobile Contract
Application Compliance: Static Analysis & In-line Monitoring
Deployment within Mobile Infrastructure
Application Development
Application Loading & Contract Matching
Contract/Policy Authoring & Matching
Execution Monitoring & Run-Time Enforcement
Contract from SME Developer
Mobile Policy Template from Operator, Company, Privacy Authority, etc.
Security by Contract
Mobile Policy on Mobile Device from Operator, Company, User, etc.

Fig. 12.2 The application development life cycle

mobile device, whereas a signature in the classical security architecture gives no precisely specified guarantees.
The upside of using this enforcement technology is that it is relatively simple to implement and use. However, third party certification can be costly, and requires trust in the certifying party.

Working prototypes of the SxC architecture are available for both the .NET Compact Framework and the Java Micro Edition [739, 224]. The .NET implementation implements the SxC system by modifying the Windows Mobile application loader. When an application is started, it is sent to a custom application loader instead of the regular loader. This custom loader tries to enforce the policy by using one of the above enforcement techniques. The Java implementation works in a very similar fashion, with only some low-level technical differences.

12.4.3 The SxC Process

To take full advantage of this new paradigm, applications have to be developed with SxC in mind. This means that some changes occur in the typical *Develop-Deploy-Run* application life cycle. Figure 12.2 shows an updated version of the application development life cycle.

The first step to develop an SxC compliant application, is to create a contract to which the application will adhere. Remember that the contract represents the security-related behavior of an application and specifies the upper-bound of calls made to SREs. Designing a contract requires intimate knowledge of the inner workings of the application, so it's typically done by a (lead-)developer or technical analyst. Once the initial version of the contract has been specified, the application development can begin. During the development, the contract can be revised and changed when needed.

After the application development, the contract must somehow be linked to the application code in a tamper-proof way. One straightforward method to do this, is by having a trusted third party inspect the application source code and the contract. If they can guarantee that the application will not violate the contract, they sign a combined hash of the application and the contract. Another way to link the contract and the code, is by generating a formal, verifiable proof that the application complies with the contract, and adding it to the application metadata container. When this step is completed, the application is ready to be deployed. The application is distributed together with its contract and optionally other metadata such as a digital signature from a third party or a proof.

When the program is deployed on a mobile device, the SxC framework checks whether the application contract is compatible with the device policy. All supported policy enforcement technologies are tried in order, until one of the techniques can certify that the contract indeed matches with the policy. If none of the formal enforcement technologies can certify that the application will comply with the policy, the application can be inlined.

When an application exits the SxC-cycle, the user can be assured that either some formal or trust-based proof is available that guarantees the application's compliance with the system policy, or that the application has been rewritten to ensure that it will never violate the policy.

Having an SxC platform on a mobile device alleviates most of the problems discussed in Sect. 12.2.3 for managed applications. The standard security mechanisms that are missing on mobile devices are replaced with an extensible new mechanism, offering fine-grained control over which API functions are allowed to be called. Experiments have shown that this new security infrastructure performs well enough to be more than workable on today's devices.

12.5 Summary and Outlook

Over the last and coming decades, the use of mobile devices and their supporting platforms have evolved considerably and, hence, the security middleware thereon has to adapt accordingly. This chapter has discussed the state-of-the-art in security middleware for mobile devices. In a first part of the chapter, the basic protection mechanisms present on these devices have been overviewed and their specific character has been discussed by highlighting a number of current limitations and problems. A second part has elaborated on two protection techniques that can resolve a number of these problems. Security by contract improves device and data protection by empowering end-users to strictly control the behavior of mobile applications either via a number of enforcement techniques, including static analysis or via runtime enforcement. Memory protection, on the other hand, guarantees that the run-time memory of an executing application cannot be accessed, or tampered with, by malign applications or users.

The challenges that lie ahead of us are manifold. Firstly, given the fact that these devices constitute a core part of our daily lives, and since they impact both business and personal life, data protection on these devices (both via API's as well as via direct access) is becoming a critical success factor. At the same time, the tighter coupling of mobile devices with other personal and company systems will have to be realized, both from a functionality perspective as well as from a security perspective. An example of the latter is the more straightforward deployment of company-wide security policies. Secondly, the ongoing proliferation of platforms has lead developers towards web applications as a new model of applications for mobile devices. From a security perspective, this means that new efforts will have to be spent in securing the web browsers on these devices, which again are typically more limited that full-blown variants. Also, the grafting of general purpose security mechanisms upon different platforms could help in supporting this situation. Finally, the close connection between mobile devices and their users warrants more research on improving the usability of the security middleware on these devices, and in particular the trade-off with functionality and performance.

Acknowledgements

This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, and by the Research Fund K.U. Leuven.

Springer

http://www.springer.com/978-3-540-89706-4

Middleware for Network Eccentric and Mobile
Applications
Garbinato, B.; Miranda, H.; Rodrigues, L. (Eds.)
2009, XXI, 454 p., Hardcover
ISBN: 978-3-540-89706-4