DRM ON OPEN PLATFORMS – MAY BE POSSIBLE AFTER ALL

Hagai Bar-El & Yoav Weiss

Discretix Technologies Ltd., Israel

ABSTRACT

Handset development is progressing on two parallel lines. Today's phones are running various applications such as commerce, games, and data retrieval. This requires an open platform for application execution. On the other hand there is an increased need for DRM. DRM by nature requires that data is blocked by the device. Inherently, completely open platforms cannot provide means for such data blockage. There is a way to bridge these two contradicting requirements, which is by means of a secure and isolated subsystem that is integrated with the operating system.
This paper draws the lines for the implementation of such a system – a system that adopts the smallest possible subset of secure environment components to provide an environment that allows arbitrary applications to run side by side with a DRM (or other secure) application.

INTRODUCTION

Common platforms today, such as PCs, PDAs and high-end cellular phones, are open platforms. It is widely understood that to foster the advantages of the digital processing ("computing") capabilities, environments need to be open, allowing everyone to write applications and providing each of these applications with more-or-less complete access to the systems resources.

However, security aware applications, such as DRM, have their own requirements that do not go hand in hand with the general trend of keeping systems open. The ability of any application to get complete access to all resources implies that secure applications have no ways to conceal any part of their operation from hostile applications or from the user himself. Specifically, keys and other security related data objects cannot be kept in a way that prevents them from being revealed.

DRM is one of many applications which require a trusted platform in order to run securely. There are many others, but since DRM is considered to be one of the killer applications of mobile platforms it seems to deserve special focus.

Establishing a system in which arbitrary code (code that was neither approved nor shipped with the mobile device) cannot run will certainly solve the problem. If unknown code coming from external sources cannot be executed, it can certainly not cause damage to the DRM components and the problem is cleanly solved. However, this solution violates the openness of the environment and is thus not desirable. Users of mobile phones, like users of other platforms, would like to retain their right to use the device they purchased for whatever purpose they like.

The preferred solution for DRM may be one that provides a semi-closed environment. Hence, an architecture that allows arbitrary code to run, but which also keeps particular critical resources inaccessible to any module which is not the known and trusted one. This way, new code can still be introduced, and this code will have almost unlimited access to the environment resources, but this code will not be able to access the particular resources that are used for the secure execution of the DRM scheme, such as the DRM agent and the player.

In this paper I would like to show how DRM can run on a system which is not entirely moderated and which can run other, arbitrary, code at the same time. This model can be abstracted to suit other security related applications as well. The proposed design takes some of the concepts of trusted computing and draws the base minimum that is necessary to solve the problem while avoiding any constraints on the types of applications that the system can run.

The proposed architecture will be presented as a set of modifications that need to be made to a typical open platform. These modifications are not easy to implement. They involve hardware integration and they involve some integration with the operating systems kernel. However, they are simpler than establishing a tightly closed system and more importantly, they provide us with a system that is indistinguishable from the original open platform for any application that does not attempt to circumvent the DRM component or any other secure resource for that matter.

REQUIRED MODIFICATIONS

As explained above, this chapter lists the modifications and additions that need to be made to a typical open platform so it can securely execute a DRM application implementing a DRM scheme. Functions and system calls will be referred to by their function rather than by their name on a particular platform for the sake of keeping the description operating system agnostic.

Secure Boot

Perhaps the most essential change is for assuring that the operating system that loads at boot time is indeed the operating system on which we ground the secure system and that it was not modified while it was stored on secondary memory (such as flash memory). For this purpose we need to digitally sign every piece of code that loads from the initial boot until the operating system is fully functional in its user mode. If this integrity protection is not provided, an attacker can simply overwrite the security modules so to disable, or otherwise circumvent them.

Essentially, this means that digital signatures need to verify that the correct operating system is loaded and that this operating system was not modified to the level of the kernel as well as any auxiliary code that the kernel may load, such as device drivers, which naturally run in kernel mode. The effort of signing device drivers and "kernel-level code" is necessary so it shall not be possible for an attacker to introduce code of his choice, possibly in the shape of a device driver, which will utilize access to kernel functions of the operating system in an attempt to circumvent the protective measures.

The bootstrap process that employs digital signatures must start with a hardware component in order to assure that the anchor of trust (typically consisting of the verification code and the associated hash value, MAC or public key) cannot be replaced. In the usual case, ROM code is responsible for loading the initial operating system components. This code should be written to assure that the hash of the code that is being loaded conforms to a non-modifiable hash value. A more complex implementation, one which allows code to be legally modified without replacing ROM, can consist of the verification of a digital signature using a public key that is stored in ROM. The purpose of the ROM code is to assure that the operating system kernel is loaded properly and that it was not tampered with. If the scheme that is used is such that checks an asymmetric digital signature rather than compares a simple hash then several bits of modifiable memory in hardware should be added to retain an increasing-only counter that will retain the version number of the code. This is so it shall not be possible to revert to obsolete software versions that were once signed but that should no longer be used.

When this part of the boot is complete we are guaranteed that an unmodified version of the kernel is running. We are also guaranteed that any other components that run in kernel mode were not modified and are ones we trust not to misuse their exclusive privileges.

The Protected Resource

At the heart of the DRM system is the resource we aim at protecting. In the case of DRM this resource is typically a driver that uses securely stored credentials in hardware to decipher rights objects and that uses content keys from within these right objects to decipher encrypted content and pass it to the player for playback or display. This resource is the "secure and isolated subsystem" referred to in the abstract.

Although not compulsory, the protected resource is likely to be linked to a hardware component. Hardware may be used either for the purpose of increased performance and CPU offload when deciphering content or for security reasons, as it is widely perceived that only hardware components are truly capable of hiding secret credentials from the possessor of the device - a feature that is required for DRM implementations. We shall thus treat the protected resource as a driver, possibly interfacing with a hardware component. This driver is responsible for providing cryptographic operations to legal consumer processes, hence (in the case of DRM), to the DRM agent and to the player applications. It is the moderation of the access to these services and to this kernel-level driver as well as to its consumers that is the target of this scheme.

We protect the access to this driver by maintaining a list within its memory space of process IDs of the processes that are allowed to use it. As far as our design is concerned, a binary level of access control suffices; either a process is allowed to access the resource or it is not. Details on how items are added or removed from this list will follow. Upon each call, the driver is responsible for looking up its runtime table of "allowed" process IDs for the ID of the current process. Since the protected resource runs in kernel mode it should have access to the ID of the current process (this information must appear in this or that form to allow the program counter to eventually return to the calling process). In the DRM case it is likely that the

processes that are allowed to access the protected resource are the DRM agent and player applications. Once the protected resource driver has figured whether or not the calling process is allowed to access the resource, it is either serving the calling process or terminating. Given the two consumers of the resource are the DRM agent and the player, both being trusted applications, we do not need to implement any sort of privileged access control facilities.

System Calls Monitoring

Our protected resource, being a kernel-level driver, and running only among trusted ones, shall be set to monitor a few of the operating systems system calls. As we aim at minimal interference with the given configuration, we attempt to monitor only the bare minimum of the system calls that is required to assure that neither the protected resource nor its legal consumers are misused.

Debug System calls. System calls used for debugging should generally be avoided on the platform when used in production if called against the trusted DRM components. Debug system calls are risky because they allow a process to read information that belongs to another process. This feature may be harmful if targeted at the protected resource or at its consumers. Following the DRM use-case, a debug call could extract data from the player which may consist of the deciphered content that can further be dumped into a file. In other cases where the player is deciphering the content by itself the debug call might reveal the content encryption key that is temporarily stored in the player's data segment. Therefore, debug calls that aim at debugging an "allowed" process must be trapped and terminated.

It should be emphasized that in order for the process ID examination at debug sys-call time to be effective against race condition attacks, the examination of the process ID for which debug is requested should be repeated when the code that handles the call returns. This will prevent a situation in which debugging is requested for a process ID that will be assigned to a trusted (or "approved") process (the player or the DRM agent) shortly after the call is made and acknowledged.

The necessity of debug calls prevention applies not only to interactive debugging but also to passive forms such as dumped cores. Such memory images that are stored as ordinary files on secondary storage might be a goldmine for an attacker that is likely to scan it for plaintext content and keys. Therefore, it must be assured that neither the protected resource nor its customers can be tricked into triggering a segmentation fault or any other circumstance that may result in the kernel dumping the core of the process into secondary memory.

Other than making sure that the code does not contain bugs that can lead to this undesired action, we must trap the exception signal which triggers a core-dump. This signal can, on some operating systems, be faked by other processes. Because this signal can be either fake or legitimate and therefore shall not be generally ignored, its handling should be set to simple process termination without the side-effect of dumping the core to secondary storage.

An additional or alternative precaution can be taken by assuring that the process context cannot write core files. In cases where the dump is written through the process context, limiting the ability of the context to write such a file will prevent it from dumping its core shall the circumstances cause that. An alternative approach may be to trap the appropriate system call that leads to the generation of the file and mask it so no file is written.

Execution Calls. The most important system calls that need to be monitored are the system calls used to load new object codes that start new processes. Upon each load of an object code which by its name or location seems to be a legal consumer of the protected resource (hence, a player or the DRM agent), a digital signature or hash has to be verified. In order to prevent race conditions in which the code object file is modified in the time interval between its signature verification and the actual execution process, it is advisable to check the validity of the signature on the code when it is already in memory, just before passing the execution pointer to it. Alternatively, the signature verification can be done on the code when in RAM by the protected resource driver itself, whenever the driver is called by that process.

If the signature verifies properly, hence the code is indeed the valid consumer's code and was not modified, then the process ID that is assigned to this process shall be added to the list of "allowed" process IDs, which is maintained by the driver of the protected resource. Otherwise, no change to the table is made. There is no need to prevent non-trusted code from running, it just needs to be marked properly so any service calls it makes to the protected resource are denied.

Library (DLL) Mapping. System calls issued by processes for mapping of new dynamic libraries should be monitored as well, for the case in which the loaded library is linked to a process which appears in the

"approved" list. In this case, the object code of the linked library should be verified against a digital signature or hash of its own. In the case of a failed verification, the process shall promptly be removed from the "approved" list or killed (see later comment on this matter). Clearly, the purpose of checking loaded libraries is solely for the removal of allowed processes from the list or from memory as soon as they link to additional code which is not trusted.

The same race condition issue that was dealt with when loading a new process needs to be dealt with again here. Libraries that are loaded in runtime need to be verified only after they already reside in RAM, right before they are actually linked to, rather than to be verified while still on secondary storage.

The removal of a trusted process (the process of the DRM agent or player) from the "approved" list without removing it entirely from memory can pose a security hazard because this process may contain sensitive information in its data segment, such as content keys. This data may be misused after loading a malicious library. Indeed, the malicious library will cause the process to lose its security clearance, but this will not affect data that was already obtained by the process. In order to avoid this risk, a process may be killed, rather than just be removed from the "approved" list, if it attempts to link to an untrusted library.

An alternative solution may be to kill a process which links to an untrusted library only if this process was served by the protected resource since it was loaded. In other cases the process shall just be removed from the "approved" list so it is never served in the future. The protected resource driver will retain a flag for each "approved" process so it is able to determine whether or not a process was served. This flexible approach will allow a player, for example, to load insecure codecs that are linked separately as long as it uses them only for non-DRM content.

Monitoring system calls of library mapping is not necessary if the consumers of the protected resource are compiled in a way that they do not consist of function calls that require loading of separate modules. This, however, seems as a tough requirement for DRM players which are likely to consist of various codecs that may be installed conditionally and linked at runtime. If integrity of these codecs is not assured, an adversary may change them, e.g. so they dump a plaintext copy of the media file to flash memory for further unlimited use and distribution.

Process Termination. As important as it is to add process IDs to the "approved" list, if they are verified as trustworthy, it is important to remove such an item from the list when the process terminates, gracefully or brutally. It is strongly required to remove entries of killed or exited processes because the "approved" list identifies processes by their process ID, which is reusable by the operating system. Once a process has terminated, its ID can be assigned to another process which is not necessarily trusted.
Once again due to race conditions, the removal from the "approved" list must occur as close as possible to the stage in which the process is killed, to make sure the right process ID is removed.

Other Possible System Calls. The system calls presented so far comprise of the minimum that is required to protect the protected resource of the DRM environment and its consumers. However, throughout this design runs an assumption of reliable code. We trust the protected resource as well as its consumers to be written in an error-free fashion. Implementation errors in any of these components may cause an adversary to be able to exploit these components into performing illegal operations such as serving out keys and/or raw content by the use of buffer overflows or by causing the kernel to dump the memory space to flash storage due to a critical error. For similar reasons we are forced to rely on the operating systems kernel code to be clean of bugs that may lead to an attacker being able to circumvent the protection scheme in part or in whole.

Whereas the requirement for bug-free code holds unconditionally for the operating system kernel module and for the protected resource driver, it can be softened for the player and for the DRM agent at the expense of trapping additional system calls that, when issued by the player or agent, indicate the occurrence of an invalid operation that is outside the frame of its normal behaviour. For example, a system call can be trapped to alert the protected resource in case the player attempts to write a file to secondary memory, which is not its own configuration file. Also, a trapped system call can be used to detect attempts to run programs (such as an OS shell) from within a trusted process. Such alerts, if carried out properly, can mitigate many possible attacks that are made possible by poor coding.

SUMMARY

Trusted and open environments are often distinguished in a binary fashion, bluntly separating between environments that are trusted versus environments that are open. The strict distinguishing between open and closed platforms leads to inconvenience when it comes to DRM applications on mobile phones. The trend of open systems in which the user is allowed to run code of his choice does not seem to coexist with the strict

robustness requirements of DRM schemes, which all require a moderated system.

Much work was and is done in an attempt to solve the trusted computing issue. Architectures known today for secure computing environments do allow the existence of trusted and not-trusted applications, but they usually require them to run separately, assuring the platform has to deal with just one type of applications at a time. This approach does have clear advantages in terms of security and may be required when running top-grade security applications. However, it forms what I believe to be difficult-to-accept limitations in terms of flexibility for the user, as well as of convenience. There is further uncertainty regarding the applicability of the trusted platforms designed for PCs to the mobile environments. Furthermore, some trusted computing platforms that are proposed today consist of extra features that raise user privacy concerns that may significantly delay their acceptance and deployment.

The solution outlined in this paper attempts to tackle the challenge without going all the way into a trusted computing platform of the types known today. Instead, it outlines the set of modifications that provide the minimal hardening that is required for DRM schemes to run securely in mobile commercial environments.