Code Authorization on Brew MP through Digital Signing

QUALCOMM Incorporated
5775 Morehouse Drive
San Diego, CA. 92121-1714
U.S.A
.

HT##-VT###-### Rev. A
December 06, 2010

Contents

Mobile Application Security

Today, any platform that allows end-users to install applications must defend against malware - code designed to in some way attack the device, the network, or the end-user. Such code may contain a virus or a worm, or may attempt to 'own' the device, log key strokes, access confidential data, degrade mobile network performance, generate unintended billing events, and so on.

There are several complimentary approaches to defending against and mitigating the threat posed by both malware and poorly implemented code. Broadly speaking, there are two approaches. The first strategy is to control what code may execute on the device in an attempt to preclude malware; the second is to limit the services or data a given executable may access to constrain the potential damage it may inflict.

Controlling what code may run

There are several methods to control what runs on a device. A familiar model practiced on most PCs is virus scanning or blacklisting code that should not be allowed on the device. This approach attempts to catch and prevent code from running once it has been identified as malware. Virus scanning is by definition a reactive practice, reliant on continuous and ultimately unbounded updates to detection mechanisms.

The converse approach is to white-list the code that is permitted to execute on the device. This is most effectively accomplished through digitally signing code that you trust. If a platform is configured to permit only the execution of digitally signed code, the user can only install code that has been somehow vetted to ensure that it is not malware. This paper describes in detail how Brew MP supports and relies upon the white-listing of code through digital signing.

Constraining code that runs

To mitigate the havoc that malware, or even poorly acting code, may wreak, it is best to limit the capability of the code to only what is strictly necessary for the code to fulfill its purpose. There are several ways to control what code may do when it is running. These include memory protection across processes through virtualization, or sandboxing, which isolates potential malware from critical system components. (Memory protection is beyond the scope of this paper.) Constraining code is further achieved by restricting the services and data the code may access by controlling the privileges that the code is granted. For example, not every application requires access to user data such as the address book or platform services such as the network.

Control over privileges is much more effective in thwarting bad acting code when privileges are managed and granted by a party vested in the security of the device. Developers are not incented to constrain the privileges granted to their applications and attackers seek the greatest capability they can for their malware. This paper described how code authorization through digital signing on Brew MP provides a mechanism to control the privileges that any given code is allowed.

Summary

Brew MP is engineered with proactive defenses against viruses and malware, the cornerstone of which is explicitly authorizing, or *white-listing,* the code that may execute on the device. This inserts into the software release process an opportunity and a mechanism for device manufacturers or mobile network operators to understand, vet, and even verify the code, its origin, and its developer. Code authorization

through white-listing means knowing where the code came from before you run it. Effective white-listing avoids the introduction of anonymous code, which is more likely to be malware.

Code authorization

There are four methods for authorizing code to run on a device:

1. Digitally signing fully dynamic modules
2. Including the static hash of dynamic modules in the boot image
3. Including static (const) modules in the boot image
4. Hard linking modules into the boot image

The last three methods (described more fully in the appendix) are applicable only to code that is finalized and available at the time of manufacture. Each of these three methods binds the code to the boot image of the device. For each of these methods, the device manufacture is responsible to determine the suitability and trustworthiness of the code through their own methods and criteria.

The first method, digital signing, enables loading dynamic code onto a device that is already deployed commercially. Using digital signing, a device manufacturer may also empower third parties, called signing authorities, to determine whether code is trusted to run on a device. This paper focuses exclusively on the capabilities, requirements, and specifications for implementing code authorization on Brew MP.

Signing authorities

A signing authority is a legal entity that determines whether code can be trusted to run on a mobile device and assumes responsibility for the security of the code. Device manufacturers, mobile network operators, and other mobile services providers may be signing authorities, and each signing authority is responsible for setting their own criteria and methods for determining what code may be run on a device. A signing authority uses digital signatures to authorize code they deem trusted. A given device may be configured by the manufacturer to support multiple signing authorities.

Signing policies

Mobile network operators, device manufacturers, mobile service providers, and consumers depend upon signing authorities to practice or enforce a signing policy - the criteria under which they will authorize code to run on the device. A signing authority's policy can include standards such as:

- The type and degree of code testing they require
- The extent to which they authenticate application developers or publishers
- How they determine which services and data are allowed access to a given application

Some signing authorities authorize only code that their own organization produces, while others authorize code from a wide range of external sources. Those that sign only their own code (perhaps a device manufacturer) may rely on their internal development and testing processes for a signing policy. Signing authorities that authorize code from many external parties typically have a more conservative signing policy.

Certificate authorities

A certificate authority (CA) is a legal entity that determines which signing authorities are trusted by a given device. Certificate authorities are expert in the deployment of a Public Key Infrastructure (PKI) and the secure procedures to store and access private keys. Certificate authority enables signing authorities through the issuance of signing certificates, which give them the authority and the means to generate digital signatures for code according to the signing authority's signing policy.

Device manufacturers determine what certificate authorities (and therefore which signing authorities) are enabled on a device by installing and configuring a root certificate for one or more certificate authorities on a device at the time of manufacture. Multiple certificate authorities may be enabled on a device by including and configuring multiple root certificates.

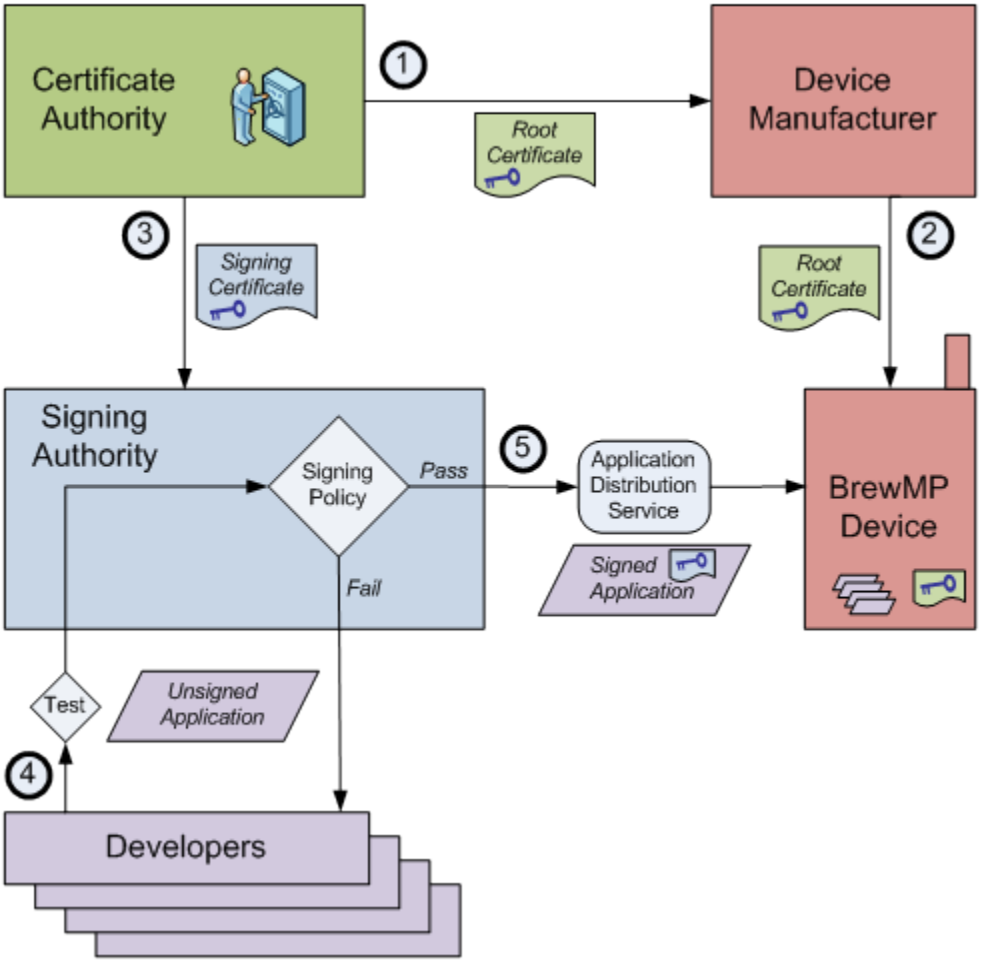

Figure 1. Digital Signing Ecosystem -

1. The certificate authority provides the device manufacturer with a root certificate.
2. The device manufacturer immutably configures the root certificate into the device image.
3. The certificate authority issues signing certificate(s) to the signing authority.
4. Developers submit unsigned code to the signing authorities.
5. The signing authority issues digital signatures for code that meets the criteria set forth in their signing policy.

Relationships

The relationship between a device manufacturer and a signing or certificate authority is most often contractual in nature. Device manufacturers delegate trust to certificate authorities, who in turn delegate trust to signing authorities. The criteria for what parties and what code are trusted should be fully negotiated and understood before a root certificate is embedded in a commercial device.

As mentioned previously, a device manufacturer or mobile network operator may act in the role of a signing authority. For some code authorization business models, the signing authority and the certificate

authority may be a single legal entity. As an example, a device manufacturer (or mobile network operator) may deploy and manage both a certificate management and a code signing operation. In such a scenario, the distinct activities of each function remain:

- **Certificate authority**
  - Securely manage private root keys
  - Provision device manufacturers with root certificates
  - Issue signing certificates to signing authorities
- **Signing authority**
  - Determine whether code is trustworthy for execution on a device in accordance with their signing policy
  - Authorize the code through digital signing

Digital Signing on Brew MP

By design, Brew MP requires that all dynamic code (applications, extensions, or modules for which there is no static hash in the boot image) be authorized thorough digital signing in order to execute. This requirement is the principal control and protection from malware attacks for mobile devices and networks. It gives device manufacturers (and mobile operators) control over what runs on their devices through explicit control over what signing authorities are trusted by the device.

Manufacturers ship Brew MP devices with embedded root certificates and those devices will only execute code that is explicitly trusted by one of these roots. There must be a process in place for determining what code is trusted to run on a device and a mechanism for digitally signing the code determined to be trusted.This service is run by a signing authority.

Signing dynamic modules

Most Brew MP applications comprise the following files:

- Application Module (.mod) file - the executable application
- Module Information file (.mif) - includes the privileges an application is granted
- Resource file (.bar) - (optional) contains application resources
- Signature file (.sig)

The .mod, and .mif are together signed to generate the .sig using an application signing key that is trusted by the root. The .sig file contains the digital signature that Brew MP validates before executing the application.

There are two types of signatures that Brew MP allows, Code Signatures and Developer Enablement Signatures.

Code signatures

Code signatures (sometimes called production signatures) are for code that is completed, tested, and authorized to run on many devices. Code signatures ensure code integrity by enabling the device to detect any modification of the code. If an application is modified (even 1 bit) after signing, the signature is no longer valid and Brew MP will not execute the code. Code signatures do not expire and may be run on any device that has the necessary root certificate configuration.

Developer enablement signatures

Developer Enablement Signatures enable the application developer to execute code for which they do not have a code signature during the testing phase of development. Instead of being specific to a single module, developer enablement signatures are tied to a specific device via an available unique identifier such as the IMEI or ESN. Developer enablement signatures are valid only for a finite amount of time, typically between 1 and 6 months. With a developer enablement signature issued under a root certificate that is enabled on a device, you can run any dynamic module. In other words, you need only one developer enablement signature to execute any application on the device. Unlike code signatures, developer enablement signatures do not verify the integrity of the code. Instead, they enable a developer to rapidly test an application (code) during the development process without the need to generate a new signature after each code change.

Developer enablement signatures for multiple hardware IDs may also be generated, and hardware IDs can be listed individually, as a range, or as a list of ranges. This makes it possible to use one test signature across many devices, and is especially suited to a team development environment.

Signing authorities and signatures

A signing authority may issue code signatures, developer enablement signatures, or both. The signing policy for code signatures is necessarily different than for that of test signatures.

- For commercial environments, code signatures are typically issued only after the application has been tested and the signing authority is comfortable that it does not pose a risk to the handset, the network, or the end user. Because a code signature enables the application to run on most any device configured with the root certificate under which the signing authority's signing certificate was issued, code signatures are more difficult to obtain than developer enablement signatures.
- Developer enablement signatures are typically issued to developers without a specific review of their code. Because the developer enablement signature allows the code to run only on specific hardware and only for a limited duration, developer enablement signatures are easier to obtain than code signatures.

Because developer enablement signatures (by design) make so much possible on a given device, it is important to control and limit their issuance. Signing authorities that provide test enabling services should strictly limit their issuance to only known and trusted developers.

Privileges and digital signing

In Brew MP, applications require privileges to access services and data through APIs. Digital signing is the mechanism that protects privileges by guaranteeing that they cannot be undetectably changed after the application and its resource file (.mif, which contains the privileges), have been signed. In other words, privileges are granted when the signature is issued. Before granting the requested privileges, the signing authority typically considers what privileges the code requires given its intended function, applying the principal of least privilege.

The principle of least privilege

Least privilege is a fundamental security principle specifying that entities (in this case the code intended to run on a device) are only given enough privilege or ability to accomplish their intended and advertised task or purpose and no more. Application of this principal helps limit the potential for damage to the consumer, the device, or the network if the application is malicious or exploited by an attacker. Brew MP allows the application of this principal by providing a fine-grained mechanism for the granting of privileges.

For example, a developer might submit a simple game application (e.g. bowling) and request privileges for address book or network access. The signing authority should (according to the principle of least privilege) reject such a submission, requiring that the developer either justify and demonstrate the need for such privileges or resubmit their applications without the unnecessary privileges.

Constraining a certificate authority's privileges

In Brew MP, a device manufacturer can decide to constrain the privileges that a root certificate may confer by configuring the root certificate to only grant certain privileges. These constraints are managed in the root configuration.

Constraining a signing authority's privileges

A certificate authority can constrain the privileges a signing authority may confer. These constraints are included in the signing certificate issued that is issued by the certificate authority to the signing authority.

What privileges does the code receive?

For an application to be granted a privilege by Brew MP, the privilege must be:

- Included in the MIF that has been digitally signed
- Included in the signing certificate that is issued to the signing authority by the certificate authority
- Authorized in the corresponding root certificate configuration by the device manufacturer

For both a root certificate configuration and a signing certificate, if no list or range of privileges is specified, the root or signing certificate is assumed to be capable of granting any privilege.

Privileges for extensions

Brew MP supports dynamic code that is not an application or a service and only executes when it is called by an application. The application's .mif specifies what code is an extension. When the extension is called and executes, it is granted only the privileges of the code from which it was called.

Configuring privileges

Subsequent chapters provide a more detailed discussion on how to configure privileges in Brew MP devices and signing certificates.

Trusted code groups

As discussed earlier, digital signatures in Brew MP allow for a single certificate authority to support and enable multiple signing authorities through the issuance of signing certificates. Similar to constraining what privileges a signing authority may allow, the certificate authority may constrain the devices for which a signing authority may authorize code through the use of trusted code groups (TCGs).

By design, a TCG is a unique identifier. The most familiar use of a TCG is to indicate a specific mobile network operator or mobile service provider.

Example - TCG as carrier identifier

Devices shipped by a mobile network operator may include a certificate authority's root certificate configured for a single TCG. (In this case, the TCG is sometimes referred to as a Carrier ID.) In the simplest case, that same mobile network operator may have contracted with one signing authority, whose signing certificate may be similarly constrained (by the certificate authority upon issuance) to a single TCG. Thus the signing authority has only been delegated the ability to authorize code that executes on the devices of that mobile network operator.

Example - TCG as service provider identifier

TCGs can be used to identify and independently authorize a mobile service provider (e.g. an application aggregator and distributor). Such providers may choose to function as their own signing authority under their own TCG. If a mobile network operator decides to enable the service provider, they do so by having their devices configured with the necessary root certificate with the service providers TCG.

What TCGs are authorized on a Brew MP device?

For an application to run on a Brew MP device, one of the TCGs in the signing certificate must be listed in the corresponding root certificate configuration on the device.

The manufacturer can configure a device to be globally constrained to a set of TCGs. These devices will only run code where the globally configured TCG in the signing certificate is listed in the corresponding root certificate configuration

Configuring TCG's

Subsequent chapters present a more detailed discussion on how to configure TCGs in Brew MP devices and signing certificates.

Example Deployments

Business requirements and relationships largely dictate how a dynamic application ecosystem is deployed, including which parties fulfill the role of certificate authority and signing authority. Below we detail several of the possible permutations.

As you read through each, keep in mind that Brew MP devices may be configured with multiple root certificates, and that a single certificate authority might enable multiple signing authorities. For these reasons, the examples below are not mutually exclusive. They could all be enabled on a single Brew MP device if the business required it.

Mobile network operator as certificate authority and mobile service provider as signing authority

The mobile network operator often assumes fundamental responsibility for the application security of devices operating on their network, so a mobile network operator is a natural owner of the root certificate. Acting as the certificate authority allows them to set policy and directly manage the delegation of proscribed signing authorities through the issuance of signing certificates.

The mobile network operator may then determine what sources of dynamic applications to make available to their subscribers. They may elect to partner with a mobile service provider that aggregates, vends, and downloads applications to mobile devices. In this case, the mobile service provider might act as the signing authority, running the signing operation according to an agreed signing policy. For such an arrangement, there is typically some understanding between the mobile network operator and the mobile service provider regarding the level of testing and vetting necessary to prevent the introduction of malware or misbehaving code. This arrangement is often manifested in a contract, but most certainly in the signing policy of the mobile service provider.

In practice, a mobile network operator provides its root certificate to a device manufacturer along with their requirements for how it should be configured on the device. The mobile network operator also issues one or more signing certificate(s) to the mobile service provider. Application developers and publishers submit their code to the mobile service provider signing authority, which subjects it to whatever testing, developer authentication, or other processes their signing policy requires. The mobile service provider then generates signatures for and makes available to subscribers that code which is determined to be trustworthy.

The signing policy for such code or application aggregation service is likely to be quite rigorous and might include strict vetting and real-time authentication of developers, an extensive testing and review process. Furthermore, due to a high volume of signing activity, the process and implementation for secure signing operations may be significant.

The mobile network operator as certificate authority may enable multiple service providers as signing authorities under this model.

Mobile network operator as certificate authority and device manufacturer as signing authority

Similar to the previous example, the mobile network operator as certificate authority might issue a signing certificate to the device manufacturer, giving them the signing authority role. This might enable the device manufacturer to provide post-production updates of their dynamic applications without needing to become a certificate authority themselves.

Mobile service provider as certificate authority and signing authority

For devices that are not manufactured for a specific mobile network operator, sometimes called *retail devices*, a party other than the manufacturer or mobile network operator may act as a certificate authority and signing authority.

Such a mobile service provider would provide their root along with configuration requirements to the device manufacturer, issue their own signing certificates, establish signing operations, and establish a signing policy, likely with the input and consent of the device manufacturer. Device manufacturers only enable such service providers if it makes their devices more valuable to consumers, acceptably secure to mobile network operators, and cost effective relative to their own device warranty and support obligations.

Device manufacturer as certificate and signing authority

Perhaps the most straight forward deployment of dynamic code authorization through digital signing is that of a solution entirely owned and operated by the device manufacturer. This is also a common deployment scenario because device manufacturers create many dynamic applications in the course of handset development, and a significant portion of the Brew MP platform is also dynamic. While device manufacturers have other options with less operational overhead to authorize this code (see the appendix), digital signing is a flexible solution some may prefer. Most notably, it allows device manufacturers to update dynamic applications without needing to re-image the entire device.

In this example, the device manufacturer acts as the certificate authority, generating their own root keys and certificate and issuing signing certificates under their root. Root certificate policies are both determined and configured into the boot image by the manufacturer.

The manufacturer also acts as the signing authority, establishing signing operations and determining the signing policy for what code may be digitally signed under the root. The signing policy is likely tied to the manufacturer's already existing quality and release practices. If the device manufacturer is signing only their own code and signing activity is relatively infrequent (perhaps weekly), signing operations themselves may be small, and might be largely automated under the right circumstances.

Licensing vs. Code Authorization

As a rule, licensing and code authorization through digital signing are distinct systems because business drivers, stakeholders, and implementers of a licensing solution are most often different from those of code authorization. Licensing manages the question of whether the user is permitted to run the code (e.g. have they paid for the right), while code authorization indicates whether the code is trusted to run on the device. This separation is a best practice we recommend but there is overlap between a licensing and code authorization though digital signing implementation detailed below.

Signing the License System Designation

Licensing and signing overlap in that digital signatures may be used to authenticate the *license system designation* for a given piece of code. While the implementation and enforcement of the license is outside of the signature verification mechanism, the signature prevents the unauthorized association of the code with an alternative (perhaps more permissive or less secure) licensing solution.

Signing code with no license

Not all code requires a license, so Brew MP allows you to designate code as not requiring a license using mif attributes. To execute the code, a mif need only be appropriately signed under a root certificate configured on the device.

Public Key Cryptography

There are two types of keys and certificates necessary to enable the digital signing of dynamic modules on Brew MP devices.

- Root Keys and Certificates
- ·Signing Keys and Certificates

This section begins with an overview of public key cryptography, and later describes both types and how they are used.

Digital signing in Brew MP is based on public key (or asymmetric) cryptography. Portions of this chapter require some knowledge of public key cryptography and public key infrastructure (PKI). We provide only a very high level overview below; for a deeper understanding, please consult other sources.

Key pairs

In a public key cryptography-based digital signing scheme, keys are generated in exclusive pairs:

- A private key used to generate signatures
- A public key used to verify signatures

An attribute of public key cryptography is that, given adequate key lengths, one cannot deduce the private key from knowledge of its associated public key, the signature, and the digital object that was signed. This attribute is the basis of non-repudiation.

Certificates

A certificate contains a public key along with several attributes, which together are signed to guarantee the certificate's integrity and to indicate that it is trusted by the signer. The purpose of this certificate is to bind an identity (subject) to a public key.

Public key hierarchy

A public key hierarchy comprises two or more key pairs and a mapping of which keys are trusted by other keys. At the top of a public key hierarchy is a root key pair. Below the root are one or more subordinate key pairs that are trusted by the root. Subordinate keys may similarly have their own subordinate key pairs that they trust.

Root certificates

A public root key is intended to be shared to enable the verification of signatures. It is shared within a root certificate, which is the public root key signed by the private root key. Thus, the public root key may be used to validate itself. In Brew MP, root certificates are included and configured immutably into the device image.

Subordinate Certificates

The root is the final authority on what keys are trusted. One confers a root's trust by signing the public key of a subordinate key pair along with other data which details how the key is intended to be used. This creates a certificate for that subordinate key that anyone with the public root key may then validate.

Subordinate certificates can in turn issue certificates (sign the public key) for keys they deem trustworthy.

Chain of trust

Thus, a public key hierarchy represents one or more chains of trust in which holders of the root certificate can validate the signature or certificate issued by any of their subordinates.

Keys and Certificates in Brew MP

The following sections describe the Brew MP-specific requirements and specifications for keys and certificates for the digital signing of dynamic modules.

Brew MP signing certificates

Signing authorities generate one or more signing key pairs, and use these keys to issue digital signatures. After generating a signing key pair, they request that a certificate authority issue a signing certificate under their root. Certificate authorities issue the resultant signing certificate by signing the signing authority's public signing key with the private root key.

This section provides an overview of the attributes of Brew MP Signing Certificates. Specific ASN.1 definitions and encoding requirements are detailed in a later chapter.

Signing certificate capabilities

As discussed previously, there are two types of signatures that a signing authority may issue, code signatures and developer enablement signatures. The difference is manifested in their Brew MP capabilities listed below.

- **No Signed Files** - this capability allows for the creation of a digital signature that does not require a list of files that must be signed. Without this capability a list of signed files must be specified.
- **No Date** - this capability allows for the creation of a digital signature that has no date restrictions. Without this capability, a date range must be specified.
- **No HW SN** - this capability allows for the creation of a digital signature that has no binding to a particular set of hardware serial numbers. Without this capability, HW SNs must be specified.

As expressed in the following table, code signing certificates require a list of signed files but no date restrictions and no hardware serial numbers while developer enablement signing certificates require date restrictions and hardware serial numbers but no list of signed files.

Certificate	List of Files	Date Restriction	HW SN
Code Signing	Yes	No	No
Developer Enablement	No	Yes	Yes

What capabilities are authorized?

As with privileges and TCGs, capabilities must be explicitly allowed in the signing certificate and any intermediate certificates that chain to a root configured on the device. For a capability to be permitted on a Brew MP device, every certificate in the chain up to the root certificate must explicitly allow that capability.

In Brew MP 1.0 the root by itself has no restrictions on what capabilities it can sign for. The ability for a device manufacturer to constrain a given root's capabilities through configuration may be added in future releases of Brew MP.

Required Brew MP signing certificate standard extensions

All Brew MP Signing Certificates require the following standard extensions:

- **basicConstraints** (OID id-ce 19) where the only necessary field is the CA Boolean field, which *should be* set to FALSE
- **extKeyUsage** (OID id-ce 37)
  - Where the only necessary field is **apiOneCodeSigning** (OID 1.3.6.1.4.1.1449.9.4.1.20) granting the ability to sign for Brew MP and BREW Client 4.x devices.

Required Brew MP signing certificate proprietary extensions

All Brew MP Signing Certificates require the following Qualcomm-defined extensions:

- **trustedCodeGroups** (OID 1.3.6.1.4.1.1449.94.1.10)
  - With the value constrained to the TCGs for which the Signing Authority may authorize code.
- **capabilities** (OID 1.3.6.1.4.1.1449.9.4.1.12) with the values:
  - For a code signing certificate: ***NoDate*** and ***NoHwSN***
  - For a developer enablement certificate: ***NoSignedFiles***

Optional Brew MP signing certificate proprietary extensions

Brew MP Signing Certificates may include the following Qualcomm defined extensions:

- **privileges** (OID 1.3.6.1.4.1.1449.9.4.1.11)
  - With the value constrained to the privileges that the Signing Authority may authorize.
  - Not including this extension implies that the signing certificate may authorize any privilege.

Below is an example of both a code signing certificate and a developer enablement signing certificate. The primary difference between the two can be seen in their capabilities extension.

Example Brew MP code signing certificate

Below is an example of a Code Signing Certificate suitable for digitally signing code on a Brew MP device. It follows the structure of an X.509 v3 digital certificate.

Field			**Value**
Version			X
Serial Number			XXXXXXXXXX
Signature Algorithm			sha256WithRSAEncryption
Issuer			o=CA_NAME, Inc. cn=CA_NAME Signing Root
Validity			notBefore = XX/XX/XXXX notAfter = XX/XX/XXXX
Subject			o=SA_NAME, Inc. cn=SA_NAME Code Signing
Public Key Info			Algorithm = RSA Modulus = 1024 bits (or 2048 bits) Exponent = 3 (F0)
Extensions	**OID**	**Criticality**	
basicConstrai	{id-ce 19}	TRUE	cA = FALSE pathLenConstraint = n/a

extKeyUsage	{id-ce 37}	TRUE	apiOneCodeSigning {1.3.6.1.4.1.1449.9.4.1.20}
trustedCodeG	{1.3.6.1.4.1.1449.9.4.1.10}	TRUE	minTCG-maxTCG
capabilities	{1.3.6.1.4.1.1449.9.4.1.12}	TRUE	No Date, No HW SNs

Example Brew MP developer enablement signing certificate

Below is an example of a signing certificate suitable for creating developer enablement signatures for a Brew MP device. It follows the structure of an X.509 v3 digital certificate.

Field			**Value**
Version			X
Serial Number			XXXXXXXXXX
Signature Algorithm			sha1WithRSAEncryption
Issuer			o=CA_NAME, Inc. cn=CA_NAME Signing Root
Validity			notBefore = XX/XX/XXXX notAfter = XX/XX/XXXX
Subject			o=SA_NAME, Inc. cn=SA_NAME Developer Enablement Signing
Public Key Info			Algorithm = RSA Modulus = 1024 bits Exponent = 3 (F0)
Extensions	**OID**	**Criticality**	
basicConstraints	{id-ce 19}	TRUE	cA = FALSE pathLenConstraint = n/a
extKeyUsage	{id-ce 37}	TRUE	apiOneCodeSigning {1.3.6.1.4.1.1449.9.4.1.20}
trustedCodeGroups	{1.3.6.1.4.1.1449.9.4.1.10}	TRUE	No TCGs
capabilities	{1.3.6.1.4.1.1449.9.4.1.12}	TRUE	No File List

Encoding of Brew MP-Specific Extensions

Brew MP Signing Certificates include Qualcomm-defined X.509 extensions, as described in earlier chapters. This section describes the coding and ASN.1 definitions for each.

Privileges

The privileges (OID 1.3.6.1.4.1.1449.9.4.1.11) extension is encoded using 32-bit unsigned integers, in little-endian format, in a structure defined in pseudo C as follows:

```
typedef unsigned int uint32 //32 bit unsigned integer

typedef uint32 Privilege;

typedef structure PrivilegeRange {
 Privilege lo;
 Privilege hi;
} PrivilegeRange;

typedef structure EncodedPrivileges {
 uint32 nbrOfRanges;
 PrivilegeRange idRanges[]; //optional array with 'nbrOfRanges' entries
 Privilege singleIds[];  //optional array
} EncodedPrivileges;
```

Trusted code groups

The trustedCodeGroups (OID 1.3.6.1.4.1.1449.94.1.10) extension is encoded using 32-bit unsigned integers, in little-endian format, in a structure defined in pseudo C as follows:

```
typedef unsigned int uint32 //32 bit unsigned integer

typedef uint32 Tcg;

typedef structure TcgRange {
 Tcg lo;
 Tcg hi;
} TcgRange;

typedef structure EncodedTcgs {
 uint32 nbrOfRanges;
 TcgRange idRanges[]; //optional array with 'nbrOfRanges' entries
 Tcg singleIds[];  //optional array
} EncodedTcgs;
```

Capabilities

The capabilities (OID 1.3.6.1.4.1.1449.9.4.1.12) extension is encoded as a DER bit string with the following ASN.1 definition:

```
apiOneCapabilities ::= BIT STRING
{
 noSignedFiles (0),
 noDate (1),
 noHwSn (2)
}
```

Securing Private Signing Keys

A signing authority has one or more key pairs that they use for the actual signing of applications. The private keys of those pairs must be kept confidential, or the whole signing operation can be compromised.

The practice and protection needed are similar to those for any public key operation, such as an SSL server or other code signing servers. This documentation describes only the particular characteristics for Brew MP signing and makes some suggestions about operational security. It is not a definitive step-by-step guide that, if followed, guarantees operational security. Those operating a signing authority should either have the necessary expertise to keep it secure, or obtain that expertise for hire.

Protecting the private key

There are a number of ways to manage the security of a private key. These include operational security on the server as well as enforcement mechanisms on the device. We begin by describing the Brew MP device characteristics that may not be modified by a signing authority.

No device enforcement of certificate expiration

Brew MP devices do not check the expiration dates of signing certificates or any certificate in the chain. For reasons of usability, the assumption is that once an application is signed, it is authorized by the signing authority to run forever. (This is the case unless the application is subsequently disabled or revoked by a kill switch mechanism, as discussed below.) The consumer has purchased the application and, baring a limit imposed by the licensing system, their ability to execute it does not expire. End-users need never reconnect to the server in order to get the same application with an updated signature. (Where counter signing is employed, there may be some expiration checking as described later.)

As discussed earlier, in monetized application distribution services, there may be an expiration of the license to run the application, but that is handled by the license system, not the signing system.

Kill Switch

Some Brew MP-based application distribution solutions provide a kill switch (or application recall mechanism) that can be used to stop an application found to be malicious from running. This too is distinct from code signing.

Implications of a compromised private key

Because Brew MP devices do not enforce expiration dates, once a private key is captured (known) by an attacker, they may use it to issue signing certificates or generate signatures for any code they like. Furthermore, the devices deployed with the corresponding root certificates are forever vulnerable unless there is some other mechanism to prevent or remediate such an event. The mechanism Brew MP supports to mitigate this risk is countersigning.

Countersigning

Brew MP supports countersigning to respond to the compromise of a certificate authority's private root key or a signing authority's private signing key. Manufacturers may configure Brew MP devices to require that every digital signature also be countersigned by a second signing authority. This configuration requires the inclusion and association of another root certificate, that of the countersigning service, in the

compiled Brew MP image on the device. Unless both signatures are present and valid, the applications signed under roots that require countersignatures will not be executed.

With countersigning in place, in order to sign unauthorized code, an attacker would need either knowledge of the signing (or certificate) authority's private key and undetected access to the counter signing service or knowledge of both the signing (or certificate) authority's private key and the countersigning authority's private key.

Countersignature verification on the device

Roots that require countersigning along with the countersigning root certificate are configured and associated with one another in the device image by the manufacturer. Countersignature verification is performed when signature verification occurs, before an application is installed and before it is loaded into RAM for execution.

Countersigning operations

Countersignatures are typically issued at the same time the first signature is generated by accessing a countersigning service through an IP connection.

- First, the local signing authority's private signing key is used to generate a signature.
- The resultant signature is then sent to the countersigning service, which validates the signature, the signing key, and the expiration date of the signing certificate.
- If each is valid, the countersigning service generates and returns a countersignature appended to and encompassing the original signature.

The countersignature format is an IETF-standard time stamp, so any time stamp server compliant with the IETF standard can be used to generate these countersignatures, so long as it verifies liveness and correctness and monitors for revocation.

The principle of countersigning is to double a potential attacker's work. In order to enable otherwise unauthorized code to run on a device, they must successfully attack both operations, and capture private keys from each. To maximize the security benefit, and reduce the opportunity for a successful insider attack, the countersigner should be in a wholly separate organization or company and geographically removed from the signing authority.

Countersigning log verification

Signing authorities are advised to check their own signing log against the countersigning log. Generated countersignatures that do not correspond to signatures that the signing authority generated, are an indication that the signing key or the counte signing service has been compromised. In such a case, the counter signingservice should stop signing against the key in question.

Certificate revocation

Certificate revocation is a common response when a key is compromised. For reasons of usability, Brew MP does not currently support a Certificate Revocation Lists (CRLs) or Online Certificate Status Protocol (OCSP) to monitor for compromised keys. CRLs and OCSP require periodic connections back to the server to refresh the list of compromised keys. CRLs and OCSP deployments are subject to a class of attacks that countersigning avoids - malicious applications that prevent access to updated CRLs. Additionally, though mobile phones are usually connected, they may be out of coverage, so requiring a real-time verification of key validity is problematic.

For revocation-dependent signature verification, the inability to update a CRL leaves two undesirable options:

- Run code with no regard for the relative freshness of the CRL, which is akin to having no revocation check at all.
- Prevent any digitally signed code from executing, thus degrading and potentially bricking the phone in the event of the above attack or prolonged spells with no data connectivity.

Today, Brew MP supports countersigning to manage the compromised key scenario. A future version of Brew MP may support OCSP or CRLs.

Mitigating risk with TCGs and privileges

Brew MP supports delegating privileges and trusted code groups. Signing certificates can be created so they are only able to sign for the privilege they require and the code groups they are intended to authorize. This constrains an attack that results from a compromised private key to only those devices that support the TCG(s) authorized by the signing certificate and limits attacker access to only those privileges granted to the signing certificate.

Operational Security

There are a number of industry best practices to employ in the protection of the private signing keys. The following are only high level suggestions for how to secure a private signing key. The list below is not exhaustive, nor is it intended as a substitute for having the necessary security expertise to securely deploy a public key infrastructure.

Private root key vs. private signing key security

The nature of private root keys is such that they should be accessed infrequently, only to issue subordinate signing certificates. This is by design and because a compromised private root key forever marginalizes physical security for all the devices that were shipped with its corresponding root certificate. The physical and procedural security for private root keys should be the best possible. The cost and inconvenience of stringent security measures are outweighed by the results of a compromise and manageable by virtue of the infrequency of interactions requiring the private root key.

Conversely, day to day signing operations necessitate that private signing keys be more frequently accessed to generate the necessary signatures. For this reason, signing certificates are given finite life spans (typically less than 2 years) to mitigate any consequences resulting from their compromise. That said, as much operational security as possible should be implemented around signing operations and the way in which private keys are stored, accessed, and used in the issuance of digital signatures.

Store private keys in a hardware token

Tamper-resistant hardware tokens are helpful in the secure storage of private keys in that they never allow the key to be removed. They support only the following operations

- Input of key data
- Encryption or signature generation within the token

The keys in such tokens may never be copied by an attacker, even if the attacker has access to the host in which the token is installed. The only way to steal the key is to physically take it within the token. Note however that if an attacker has overrun the host in which the token is installed, they can sign apps until they are discovered and locked out.

Keep private keys offline

Private keys can be stored on a stand-alone system which is not connected to a network. For example, signing operations could take place on an isolated system in a secured facility. Code deemed trustworthy enough to be signed might be physically brought into the secure signing room on removable storage such as a USB Flash drive. (Note that for operations that leverage countersigning, a network connection may be unavoidable.)

Operational security

Good operational security is a must in all cases, and the hosts used for signing should be well managed. For example, web browsers and wireless modems should be disabled. Passwords that protect against access to the private key for signing operations should be long, strong pass phrases. Only a known and limited number of people should have access to the signing operations. Auditable signing procedures, records of signing activity, and access to the signing facility are absolutely required.

Digital signing environments

When considering digital signing for devices, there are two distinct environments with different requirements for the signing services that serve them and different requirements for the handling of root keys and signing keys.

- **Commercial Device Environments** - for devices to be delivered to retailers, consumers, or operators for use by end-users running commercial software.
- **Device Development Environments** - wherein a manufacturer is developing, building, and testing pre-commercial releases of the device software.

Commercial device environments

For signing in commercial device environments, it is critical that the confidentiality of the private root key and private signing keys be maintained, and that signing operations be strictly controlled to prevent the signing of code not intentionally approved by the signing authority. ***Commercial environments require that the confidentiality and control of private keys be maintained from the moment they are generated.***

Device development environments

In a device development environment, where the current device image is not intended to be shipped in commercial handsets, device manufacturers are able to operate a less secure signing solution with little or no control of private keys and signature generation. In these device development environments, there is little or no concern about the threat of malware or what code is permitted to run on these devices. For this reason, there is no need to implement strong controls around use of the Device Root Private Key or the Signing Certificates in such an environment.

Reasonable care should be taken to prevent these pre-commercial builds including pre-commercial root certificates from being used to re-flash devices already in the field.

Common signing deployment across environments

Because the operational security and private key management requirements are higher for a commercial device environment than for a device development environment, it is practical to use a commercial signing solution for a device development environment. ***However, it is not practical to extend a device development signing solution into a commercial deployment.***

Configuring a Root Certificate in a Device

To include their own root certificate in a device image, a manufacturer embeds the root in *OEMCodeSigConfig.c*.

To include another signing authority's root certificates, the device manufacturer must create another code signature configuration (e.g.*MNOCodeSigConfig.c* or *SACodeSigConfig.c*) and create an entry for the new configuration in *OEMCodeSigConfigFactory.c.*

Detailed instructions for how to modify or add a code signature configuration can be found in*ICodeSigConfig.h* and *ICodeSigConfigFactory.h*.

To quickly enable a device manufacturer to run dynamic code in a development environment, the Brew MP Code Signing Kit includes a Non-Commercial Sample Root configuration that may be easily dropped into a device image.

When including a root certificate in a Brew MP device build, there are several configurations that may be made.

Configuring root certificate privileges and TCGs

To configure which privileges and TCGs a root certificate may authorize, you must explicitly list the privileges, individually or as ranges. More details can be found in ICodeSigConfig.h.

Appendix - Alternatives to Digital Signing

There are three alternatives to digital signing available to device manufacturers for authorizing code to run on a Brew MP device. These methods rely upon explicit inclusion in the boot image, and therefore may not be used to load applications not known at the time of manufacture.

Including the static hash of dynamically loaded modules

For modules that are finalized when the device image is being built, the static hash of a dynamic module can be included in the boot image. The dynamic module remains in the EFS, and when it is invoked, a hash of the module is compared to that which was included in the boot image. If the hashes match, Brew MP executes the module. These applications are secure from modification, but do not need to be statically linked and validated at compile time. This approach gives device manufacturers a means of shipping dynamic modules in a device without establishing their own root certificate and signing operations.

Modules loaded this way may also be upgraded (or overridden) by versions that have a valid digital signature associated with them. This ability is available only through a signing authority whose root certificate is included in the device image.

Including static (const) modules in the boot image

Including static (const) modules in the boot image presumes that the module is finalized prior to device manufacture. In this case, images of the .mif and .mod are hard-linked into the boot image. They are, therefore, always loaded into RAM. Additionally, when the module is invoked, it is copied into RAM a second time where it then executes as a typical module.

Unlike dynamic modules, static (or const) modules remain in memory even when they are not running. Static modules can be upgraded by versions with digital signatures; however, the original module always persists in RAM.

Hard linked modules (static extensions) in the boot image

You may also hard-link a module into the boot image. In this case, however, the code is effectively no longer a module. It may have BREW-style interfaces, but there is no mif. This is sometimes referred to as a static extension. The code is always in RAM, just as the whole boot image is always in RAM. Unlike the other options, static extensions may never be overridden by new dynamic modules.

Comparing the methods of code authorization

Of the four code authorization options, two are for dynamic modules stored in the EFS:

- Digital signing a fully dynamic module
- Including the static hash of a dynamic module in the boot image

Two options imply static code written entirely to the boot image:

- Including static (const) modules in the boot image
- Hard-linking modules into the boot image

Options for dynamic modules

Dynamic signing is the most flexible method of code authorization, giving the device manufacturer, operator, or other signing authority a mechanism for updating devices after they have been shipped.

It is the only method that allows for changes to an application without the need to recompile the boot image. Digital signing does, however, require that a signing authority undertake the overhead and effort necessary for securely building and maintaining a public key hierarchy and signing operation.

	RAM Usage	Module can be Upgraded?	May be Factory Loaded	May Be Loaded after Manufacture	Requires Certificate/ Signing Authority
Signing Dynamic Modules	Module is only in RAM when it is executing.	Yes, replacing the previous version in the EFS	Yes	Yes	Yes
Including the Static Hash of a Dynamic Module	Module is only in RAM when it is executing.	Yes, with a signed dynamic module replacing the previous version in the EFS.	Yes	No	No
Including Static (Const) Modules	Module is always in RAM. A second instance is created in RAM when the module is invoked.	Yes, with a signed dynamic module. The previous version remains loaded in RAM.	Yes	No	No
Hard Linked Modules	Module is always in RAM	No*	Yes	No	No

* May be upgraded only through class ID override

Including the static hash of a dynamic module in the boot image is a good option for a device manufacturer who prefers not to be a signing authority. If there is another signing party (such as the operator), to act as signing authority on the device in order to load additional applications, the device manufacturer may be able to distribute upgrades of their own statically hashed applications through that signing authority.

Options for static code

Including static (or const) modules in the boot image is the worst option for consumption of resources. Not only does an inactive module consume RAM, but upon invocation, a second copy of the module is loaded into RAM.

The hard-linking of code into the boot image is not as RAM-intensive as the static (const) module option, but it has other drawbacks. It does not enable upgrade, regardless of the availability of a signing authority's root certificate on the device. Further, because it may not have a .mif, it may only be an extension, unable to register for events or notifications by itself.