Independent Submission
Internet-Draft
Intended status: Informational
Expires: 21 November 2026

Project Tick
20 May 2026

The MMCO Module Component Object and MMCS Composite Security Format
draft-projecttick-mmco-mmcs-00

Abstract

This document specifies two cooperating formats used by the MeshMC launcher to discover, validate, and load third-party native code modules at run time:

* MMCO (MMCO Module Component Object), a host-native dynamic library container with a fixed C ABI describing the module's metadata, dependencies, lifecycle entry points, and runtime context.

* MMCS (MMCO Module Composite Security), an OpenPGP-based detached signature trailer appended to an MMCO file, together with the license-driven verification policy that decides whether a given module is permitted to load.

The two formats are layered: MMCS is defined strictly on top of MMCO and never modifies the bytes that the host operating system parses as a shared object. A conforming MMCO implementation MAY ignore MMCS and still load OSI-approved open-source modules; a conforming MMCS implementation MUST refuse to load a module whose trailer is present but does not verify against a trusted key.

This document is intended as a stable reference for independent implementers of MMCO toolchains, plugin authors, and packagers.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 21 November 2026.

Table of Contents

1. Introduction

The MeshMC launcher loads third-party functionality at run time from self-contained native code modules called MMCO files. Each module is a host-native shared library (ELF [ELF] on Linux, Mach-O [MACHO] on macOS, PE [PE] on Windows) that exports a fixed set of C symbols describing its identity, its required ABI version, its declared dependencies on other modules, and three lifecycle entry points.

Native code loaded into the host process carries the same authority as the host itself. Distributing such code therefore requires a verifiable trust path between the module's bytes on disk and the identity of its author. MMCS provides that trust path by appending a detached OpenPGP signature [RFC4880] (or its successor [RFC9580]) to the end of an MMCO file in a fixed-layout trailer, and by applying a license-aware policy that distinguishes mandatory-signature modules (typically proprietary) from optional-signature modules (under OSI-approved open-source licenses [OSI]).

This document specifies both formats in a single normative reference. Section 2 defines MMCO. Section 3 defines MMCS as a layer on top of MMCO. Section 4 describes their interaction during discovery, verification, dependency resolution, and lifecycle. Section 5 registers the file extension, magic numbers, and hook identifier ranges. Section 6 discusses the threat model and known weaknesses.

1.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

The terms "MUST", "MUST NOT", "SHOULD", "SHOULD NOT", and "MAY" in this document are to be interpreted as described in [RFC2119] and [RFC8174] when, and only when, they appear in all capitals.

1.2. Terminology

Host: The process that loads MMCO modules. In the canonical implementation, this is the MeshMC launcher binary.

Module: A single MMCO file. Also called a "plugin" colloquially.

ABI: Application Binary Interface; the set of C symbols, struct layouts, calling conventions, and version constants that the host and a module agree on at load time.

Trailer: The MMCS-defined byte range appended to an MMCO file containing the detached signature and the framing footer.

Payload: The contiguous byte range from offset 0 up to (but not including) the trailer. The payload is what the host operating system parses as a shared object, and is also the exact byte range covered by the MMCS signature.

OSS module: A module whose license field carries an SPDX expression [SPDX] that resolves to at least one OSI-approved [OSI] identifier under the rules of Section 3.5.

Trusted key: An OpenPGP public key present in the host-configured keyring at signature verification time.

2. MMCO Module Component Object

An MMCO module is a host-native shared library that conforms to all of the following:

1. The host operating system's shared-object format (ELF, Mach-O, or PE), unmodified at the byte ranges the host loader parses.

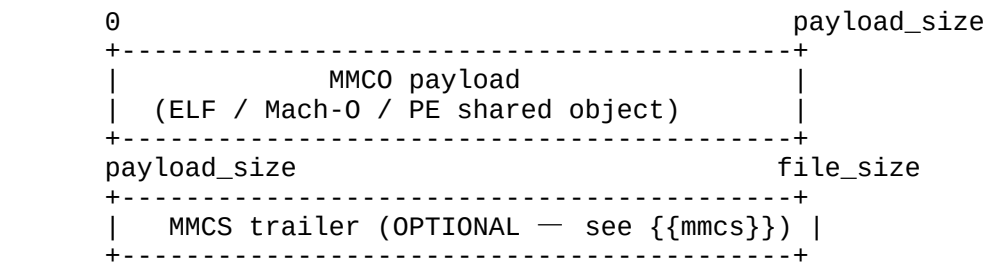
2. The MMCO C ABI defined in Section 2.2.

3. The exported-symbol contract defined in Section 2.2.2.

A file is RECOMMENDED to use the .mmco extension. Hosts MUST NOT require any particular extension and MUST identify modules by parsing the exported mmco_module_info symbol (see Section 2.2.2).

2.1. File Container

An MMCO file is the concatenation of two byte ranges:



When no MMCS trailer is present, payload_size == file_size. The MMCS trailer is fully described in Section 3.

A conforming MMCO loader MUST be able to load a module whose file ends exactly at payload_size (no trailer present). A conforming MMCO loader MUST tolerate, without error, additional bytes appended to the payload by an MMCS layer it understands, provided those bytes do not change any byte the operating system loader parses. In practice, the operating-system loader reads only sections referenced by the program header table, none of which extend past payload_size for shared objects produced by current ELF, Mach-O, or PE linkers; the MMCS trailer therefore appears as opaque tail bytes to the OS loader.

2.2. ABI

The MMCO ABI is identified by a 32-bit magic number and a 32-bit version number declared as preprocessor constants:

Constant	Value	Notes
MMCO_MAGIC	0x4D4D434F	ASCII "MMCO" (little-endian on disk).
MMCO_ABI_VERSION	2	Incremented for incompatible changes.
MMCO_EXTENSION	".mmco"	RECOMMENDED file-name suffix.

Table 1

Both constants are also embedded as fields of the MMCOModuleInfo structure exported by every module (see Section 2.2.3). A host MUST reject a module whose embedded magic field is not equal to MMCO_MAGIC. A host MUST reject a module whose embedded abi_version field is not equal to the host’s compiled-in MMCO_ABI_VERSION.

The ABI version is a single integer; semver-style ranges are not honoured. ABI changes that add new fields at the end of a struct, or new function pointers at the end of a context struct, are considered incompatible and MUST bump MMCO_ABI_VERSION. This document specifies MMCO_ABI_VERSION == 2.

2.2.1. Symbol Visibility

The host MUST resolve module symbols via the operating-system dynamic loader’s exported-symbol table. Modules MUST mark their exported symbols with default visibility:

* On ELF and Mach-O targets, modules MUST use __attribute__((visibility("default"))).

* On PE targets, modules MUST use __declspec(dllexport).

A reference implementation defines MMCO_EXPORT to expand to the appropriate attribute for the build target.

2.2.2. Required Exported Symbols

Every MMCO module MUST export the following three symbols with C linkage. Names are case-sensitive and reserved:

Symbol	Type / Signature
mmco_module_info	MMCOModuleInfo (object, not function)
mmco_init	int mmco_init(MMCOContext *ctx)
mmco_unload	void mmco_unload(void)

Table 2

The host MUST resolve mmco_module_info before invoking any other entry point. The host MUST validate the magic and abi_version fields of mmco_module_info before resolving mmco_init or mmco_unload. If either symbol is missing, the host MUST unload the module without invoking any entry point.

mmco_init is invoked exactly once per module, after all the module's hard dependencies (see Section 2.2.4) have been successfully initialized. It MUST return 0 on success. A non-zero return value MUST be interpreted by the host as initialization failure. On initialization failure, the host:

* MUST NOT subsequently dispatch any hook callback into that module.

* MUST mark the module as not-initialized for the remainder of the host's lifetime.

* MAY (but is not required to) close the underlying shared library immediately. The reference implementation closes it (subject to the RTLD_NODELETE caveat described in Section 2.4).

* SHOULD NOT call mmco_unload on a module whose mmco_init returned non-zero, because mmco_unload is contracted as the tear-down of a successfully-initialized module. A module that fails initialization is responsible for releasing the resources it has already acquired before returning non-zero.

mmco_unload is invoked exactly once per successfully-initialized module, in the reverse of the initialization order, during host shutdown. It MUST NOT call back into the host's MMCO API: by the time mmco_unload runs, the host has already begun tearing down the resources backing the function pointers in MMCOContext.

2.2.3. MMCOModuleInfo Structure

The mmco_module_info symbol is an object of type MMCOModuleInfo, laid out as follows in C. All pointer fields point to objects of static storage duration owned by the module; the module MUST keep them valid until mmco_unload returns.

```
struct MMCOModuleInfo {
    uint32_t              magic;             /* = MMCO_MAGIC       */
    uint32_t              abi_version;       /* = MMCO_ABI_VERSION */
    const char           *name;              /* UTF-8, non-NULL    */
    const char           *version;           /* UTF-8, non-NULL    */
    const char           *author;            /* UTF-8 or NULL      */
    const char           *description;       /* UTF-8 or NULL      */
    const char           *license;           /* SPDX, or NULL      */
    uint32_t              flags;             /* reserved, MUST = 0 */
    const char           *code_link;         /* URI or NULL        */
    const char           *icon_set_resource; /* logical, or NULL   */
    const MMCODependency *dependencies;      /* or NULL if count=0 */
    uint32_t              dependency_count;
    const char           *signing_key_id;    /* OpenPGP id or NULL */
};
```

Field requirements:

magic: MUST equal MMCO_MAGIC. Used to detect accidentally-loaded unrelated shared libraries.

abi_version: MUST equal the host's MMCO_ABI_VERSION.

name: A non-empty, UTF-8 human-readable identifier. The combination of name (case-insensitive) and the on-disk file name are both used by the host for module identification; modules SHOULD ensure these match so that the user-disable list (which is keyed on lowercased name) and the dependency table (which is keyed on lowercased name) refer to the same entity.

version: A UTF-8 string compared lexicographically segment-by-segment under the algorithm of Section 2.2.5. The empty string and NULL are equivalent and compare equal to themselves.

license: An SPDX license expression [SPDX] or NULL. NULL and the empty string are treated identically by MMCS and MUST cause the module to be classified as non-OSS (see Section 3.5).

flags: Reserved for future use. Modules MUST set this field to zero in the absence of an allocated flag bit. Hosts MUST tolerate unknown bits (i.e. MUST NOT refuse to load a module solely because flags is non-zero) so that a future flag bit can be introduced without invalidating older hosts. Future revisions of this document MAY allocate specific bits and define their semantics; once allocated, hosts implementing the new revision MUST honour them.

code_link: An informational URI (typically https://) pointing to source code, or NULL.

icon_set_resource: Logical name of a Qt-resource-bundled icon set inside the module, or NULL. The host resolves icons via the path expression ":/plugins/<icon_set_resource>/<name>" at runtime.

dependencies, dependency_count: See Section 2.2.4.

signing_key_id: An informational identifier (typically an OpenPGP key fingerprint or short id) of the key with which the module was signed, or NULL. This field is purely a hint; the authoritative signing key is whichever key the MMCS verifier extracts from the trailer (see Section 3).

2.2.4. Dependencies

A module MAY declare dependencies on other MMCO modules via the dependencies array. Each entry is laid out as follows:

```
struct MMCODependency {
    const char *name;         /* UTF-8, matched case-insensitively */
    const char *min_version;  /* SemVer-ish or NULL/""             */
    uint32_t    optional;     /* non-zero == optional dependency   */
};
```

Field requirements:

name: The lowercased value MUST match the lowercased name field of the intended dependency’s MMCOModuleInfo.

min_version: Either NULL, the empty string, or a SemVer-ish version string. When set, the resolved dependency's version MUST compare greater-than-or-equal to min_version under Section 2.2.5.

optional: When non-zero, absence or version-mismatch of the dependency MUST NOT cause the depending module to be marked unloadable; the host MUST still attempt to load the depending module without the dependency.

The host MUST construct a directed acyclic graph from the union of all modules' dependencies, MUST initialize modules in topological order (so that every module sees its hard dependencies already initialized in mmco_init), and MUST refuse to initialize any module that participates in a cycle. The specific algorithm is specified in Section 2.5.

2.2.5. Version Comparison

A version string is split on the characters ., -, +, and _, yielding an ordered list of segments. Empty segments are removed.

To compare two segment lists A and B, walk them in parallel; for each index i, compare the segments A[i] and B[i] (treating a missing segment as the empty string). Segment comparison proceeds as follows:

* If both segments parse as non-negative integers, compare them numerically.

* Otherwise compare them as Unicode strings, case-insensitively.

The first non-equal segment determines the result of the comparison. If all segments compare equal, the lists are equal.

The empty version string (and NULL) compares equal to itself and is treated as "less than" any non-empty version.

This algorithm is intentionally relaxed: it accepts common SemVer inputs, "1.2.3a", "1.2.3-rc.1", "1.2.3+build.7", and dotted Java-style versions, without requiring strict SemVer conformance.

2.2.6. MMCOContext

mmco_init receives a pointer to an MMCOContext instance owned by the host. The first three fields of MMCOContext are normative guard fields:

```
struct MMCOContext {
    uint32_t struct_size; /* host writes sizeof(MMCOContext)    */
    uint32_t abi_version; /* host writes MMCO_ABI_VERSION       */
    void    *module_handle;
    /* ... function pointers (see {{api-surface}}, informative) ... */
};
```

Field requirements:

struct_size: The host MUST set this to sizeof(MMCOContext) as the host saw the struct at compile time. Modules MAY use this to detect a host built against a newer revision that grew MMCOContext past the module's compile-time view, and refuse to dereference function pointers that lie beyond the module's last known field.

abi_version: The host MUST set this to MMCO_ABI_VERSION. Because the host has already verified mmco_module_info.abi_version matches before calling mmco_init, the two values are guaranteed equal at this point. The redundancy is intentional: it gives a module a single fail-safe assertion site at the very top of mmco_init.

module_handle: Opaque host-owned cookie that identifies the calling module to the host's API trampolines. Modules MUST treat this as opaque and MUST pass it verbatim as the first argument to every function pointer in MMCOContext. Hosts MUST guarantee that module_handle is unique per module and stable for the lifetime of the module's loaded state.

The remaining fields are function pointers whose layout, names, and semantics are part of the ABI defined by this revision but are not enumerated normatively here. See Section 2.2.7.

2.2.7. API Surface (Informative)

The full MMCOContext exposed to modules in MMCO_ABI_VERSION == 2 covers, at a minimum:

* Sectioned logging (info, warn, error, debug).
* Hook registration and dispatch (see Section 2.2.8).
* Per-module sandboxed settings and a read-only application settings accessor.
* Instance, mod, world, account, and Java enumeration and mutation.
* Filesystem and archive helpers scoped to a per-module data directory.

* HTTP GET / POST.

* UI builders for dialogs, pages, layouts, widgets, system-tray icons, and main-window event filters.

* Launch modifiers (environment variables and wrapper commands) scoped to the in-flight pre-launch hook.

* News-feed enumeration and management.

The exact signature and semantics of each function pointer in MMCOContext are part of the host implementation rather than of this format specification, and are therefore not enumerated normatively in this document. They are, however, part of the binary ABI identified by MMCO_ABI_VERSION: a host MUST NOT change the order, count, or signature of any function pointer in MMCOContext within a single ABI version. The hook payload structures, by contrast, ARE normative for this revision and are defined in Appendix B.

Future revisions of this document MAY split the MMCOContext specification into a companion reference.

2.2.8. Hooks

A module observes or modifies host behaviour by registering callback functions against numeric hook identifiers. The current allocated ranges, as of MMCO_ABI_VERSION == 2, are:

Range	Purpose
0x01000x01FF	Application lifecycle.
0x02000x02FF	Instance lifecycle.
0x03000x03FF	Settings.
0x04000x04FF	Content and mod management.
0x05000x05FF	Network.
0x06000x06FF	UI extension points.
0x07000x07FF	News.
0x08000x08FF	Authentication.

Table 3

Hook identifiers in the range 0x00000x00FF and 0xFF000xFFFF are reserved for future use. See Section 5.2 for the registry.

A registered hook callback has the signature:

```
typedef int (*MMCOHookCallback)(void     *module_handle,
                                uint32_t  hook_id,
                                void     *payload,
                                void     *user_data);
```

The callback return value carries a single piece of information, "requested cancellation":

* A return value of zero means "continue".

* A non-zero return value means "cancel".

The host MUST short-circuit hook dispatch on the first non-zero return: when a callback returns non-zero, subsequently-registered callbacks for the same hook identifier MUST NOT be invoked for the same dispatch.

Whether cancellation has any externally-visible effect is a property of the specific hook, not of the dispatcher. A hook identifier is called "cancellable" if the host action that follows the dispatch is contractually aborted when the dispatcher reports cancellation. A

hook identifier that is not cancellable is called "observational": callbacks for an observational hook MAY return non-zero, but the host MUST ignore the value for purposes of aborting the surrounding action (it still short-circuits remaining callbacks).

For MMCO_ABI_VERSION == 2, the cancellable hooks are:

Hook identifier	Cancels
MMCO_HOOK_AUTH_REQUEST	The in-flight authentication request.

Table 4

All other hooks defined by this document are observational, even those whose name contains "PRE". In particular, MMCO_HOOK_INSTANCE_PRE_LAUNCH is observational: a non-zero return short-circuits subsequent listeners but does not abort the launch. Plugins that wish to abort a launch under this revision MUST do so out-of-band (e.g. by raising a UI dialog that prompts the user) before the launch reaches the pre-launch hook.

The identifiers MMCO_HOOK_CONTENT_PRE_DOWNLOAD (0x0400) and MMCO_HOOK_NETWORK_PRE_REQUEST (0x0500) are allocated in this revision but the reference host does not yet dispatch them. Hosts MAY dispatch them in future revisions; when dispatched, they are intended to be cancellable. Plugins MAY register against these identifiers under the current revision, but their callbacks will not be invoked until the dispatch sites land in a future host revision.

Future revisions of this document MAY promote currently-observational hooks to cancellable status; doing so is considered backward-compatible because callers that returned zero continue to behave identically.

Payload pointers are owned by the host and valid only for the duration of the callback invocation. Modules MUST copy any data they wish to retain.

The complete list of hook identifiers and their payload structures is defined normatively in Appendix B.

2.3. Discovery

A host MUST scan a configurable, ordered list of directories for candidate modules. The host MUST identify candidates by attempting to open every regular file whose name ends in MMCO_EXTENSION and which the operating-system loader accepts as a shared object.

The host MUST de-duplicate discovered modules by the lowercased basename (with the trailing MMCO_EXTENSION stripped). When two candidates resolve to the same identifier, the host MUST retain the first one discovered in the configured directory order and MUST unload the duplicate.

Recommended default directories (informative; per-platform):

* ELF hosts: the directory mmcmodules alongside the host binary, then $HOME/.local/lib/mmcmodules, then $HOME/.local/share/<application>/mmcmodules, then /usr/local/lib/mmcmodules, then /usr/lib/mmcmodules.

* Windows hosts: the directory mmcmodules alongside the host binary, then a per-user mmcmodules directory inside the platform's writable app-data location (%LOCALAPPDATA%\<application>\mmcmodules in practice).

* macOS hosts: Contents/Resources/mmcmodules inside the application bundle. The bundle location Contents/PlugIns/mmcmodules is the conventional Apple choice for plug-in code, but is unsuitable for MMCO because Apple's codesign --strict validator walks Contents/PlugIns/ recursively and refuses to accept any Mach-O object that carries trailing bytes past __LINKEDIT (which an MMCS trailer does). Hosts on macOS therefore install MMCO modules under Contents/Resources/, which codesign treats as opaque data and seals via _CodeSignature/CodeResources without attempting to parse the contents as code. The Contents/MacOS directory is also unsuitable, because Apple treats every non-main-binary Mach-O found there as a subcomponent of the main executable and requires it to carry its own Apple signature. Hosts MAY additionally probe the legacy locations Contents/PlugIns/mmcmodules and Contents/MacOS/mmcmodules as read-only fallbacks for installations that predate this layout.

Search-path ordering is significant: candidates discovered in an earlier directory take precedence over duplicates discovered in a later directory (Section 2.3, second paragraph). Hosts that accept user-supplied extra directories SHOULD insert them ahead of the built-in defaults so that user-installed overrides shadow system-installed copies.

The host MUST NOT consult LD_LIBRARY_PATH, DYLD_LIBRARY_PATH, PATH, or any similar environment variable for module discovery, because doing so would allow an unprivileged process to inject a module by side-effect on a privileged host invocation.

2.4. Loading

For each candidate file, the host MUST:

1. Open the shared object via the operating-system dynamic loader (dlopen, LoadLibraryW, or equivalent), with flags equivalent to RTLD_NOW | RTLD_LOCAL. On ELF hosts, RTLD_NODELETE SHOULD be set; see Section 2.4.1 for the rationale.

2. Resolve mmco_module_info. If the symbol is absent, or its magic differs from MMCO_MAGIC, or its abi_version differs from MMCO_ABI_VERSION, the host MUST close the library and discard the candidate.

3. Resolve mmco_init and mmco_unload. If either is absent, the host MUST close the library and discard the candidate.

4. Perform the MMCS trust pre-flight specified in Section 3.4. If the pre-flight marks the module as unloadable, the host MUST retain the loaded library handle (so its metadata can be inspected by administrative tooling) but MUST NOT subsequently invoke mmco_init on it.

After all candidates are processed and the dependency resolver (Section 2.5) has produced an order, the host MUST invoke mmco_init exactly once on each enabled module, in resolver order. If mmco_init returns non-zero, the host MUST mark the module as not-initialized and skip it for the remainder of the host’s lifetime, per the requirements in Section 2.2.2.

2.4.1. Static-Storage Rationale

The RTLD_NODELETE recommendation in step 1 prevents the C runtime from running the module’s global destructors at process exit (or at a subsequent dlclose call). This is required when the module statically links any object whose destructor mutates process-wide state shared with the host: such destructors would run twice (once from the host binary’s tear-down and once from the module’s) and corrupt the shared state on the second pass.

Modules that link only against the public MMCO SDK (Section 2.2.7) and against shared libraries that the host itself uses (e.g. Qt, libc, OpenSSL) are not subject to this hazard, because they do not introduce duplicate definitions. The reference implementation nevertheless applies RTLD_NODELETE universally, on the grounds that:

* The cost is bounded (a small amount of resident memory per loaded module retained until process exit).

* It removes a load-time ordering hazard for plugin authors who may, in good faith, statically link a helper library that later turns out to be incompatible with destructor re-runs.

A host MAY omit RTLD_NODELETE if it can establish that no such duplicate-static-state hazard exists in its plugin ecosystem.

2.5. Dependency Resolution

The host MUST resolve the load order using a topological sort over the dependency DAG. Implementations are RECOMMENDED to use Kahn’s algorithm [KAHN-TOPO]. The algorithm operates as follows:

1. Build a name-to-index map over the discovered modules, keyed on the lowercased name field.

2. For each module M that is not already disabled, for each declared dependency D of M:

   a. If D is not present in the map, and D is non-optional, mark M disabled with reason DependencyMissing. b. Else if the resolved D is already disabled, and D is non-optional, mark M disabled with reason DependencyMissing. c. Else if the resolved D’s version compares less than D’s min_version, and D is non-optional, mark M disabled with reason DependencyMissing. d. Else, add an edge from D to M in the DAG.

3. Initialize a queue with every enabled module of in-degree zero.

4. While the queue is non-empty: dequeue a module, append it to the load order, and decrement the in-degree of each enabled successor; enqueue successors that reach in-degree zero.

5. Any enabled module not visited by the walk participates in a cycle. The host MUST mark such modules disabled with reason DependencyCycle.

The host MUST honour pre-existing disable flags (e.g. set by the MMCS pre-flight or by user configuration) and MUST propagate DependencyMissing to anything that depended on a disabled module.

2.6. Per-Module Sandbox

The host MUST expose to every module a private, isolated view of two host-managed resources: a settings namespace and a filesystem directory. The intent is to give modules a stable place to keep small amounts of state without colliding with other modules or with the host’s own configuration.

2.6.1. Module Identity

For the purposes of sandboxing, each module is identified by a case-sensitive UTF-8 token derived from the on-disk file name: take the basename of the module’s file path, then strip the .mmco suffix if present. This identifier (the "module id") is distinct from the human-readable name field of MMCOModuleInfo; the latter MAY contain spaces and is intended for display, whereas the module id is filesystem-safe and intended for namespacing.

2.6.2. Settings Sandbox

The host MUST namespace every per-module setting under a host-internal key of the form plugin.<module_id>.<key>. A module that requests the setting key foo MUST receive the value of the host setting plugin.<module_id>.foo, and only that value; it MUST NOT be able, through the per-module settings API, to read or write any host setting outside its own namespace.

The host MAY additionally expose a separate read-only accessor for global application settings, distinct from the per-module setter/getter. The reference implementation provides such an accessor under the name app_setting_get.

2.6.3. Filesystem Sandbox

The host MUST allocate, on first request, a unique per-module directory at a stable path of the form <base>/plugin-data/<module_id>/. The choice of <base> is platform-specific:

* ELF hosts: $HOME/.local/share/<application> is RECOMMENDED.

* Windows hosts: the platform’s writable application-data location (%APPDATA%\<application>\ in practice).

* macOS hosts: ~/Library/Application Support/<application> is conventional.

The per-module data directory MUST exist when mmco_init is invoked. The host MUST provide module-facing filesystem helpers that resolve relative paths against the per-module directory.

A host MAY additionally provide unrestricted filesystem helpers that take absolute paths; such helpers grant the same authority as direct system calls from within the module and are intended for plugins that legitimately need to operate outside their sandbox (e.g. mod managers writing into instance directories). Those helpers are outside the security boundary of Section 2.6 and are governed by the broader trust model in Section 6.

2.6.4. User-Disable List

The host MUST persist an end-user-controlled set of disabled module names across launches. The reference implementation stores this set under the host setting key plugins.disabled as a comma-separated list of lowercased module names. The host:

* MUST treat the comparison against discovered modules as case-insensitive.

* MUST accept either the lowercased name field of MMCOModuleInfo or the module id (Section 2.6.1) as a match against the disable list.

* MUST mark a matched module with disable reason UserDisabled. The module is still loaded (so its metadata is visible to administrative tooling) but its mmco_init is not invoked.

* SHOULD warn the user that toggling the disable list takes effect on the next host restart, because modules are not unloaded mid-session.

2.7. Shutdown

During host shutdown, the host MUST call mmco_unload on every initialized module in the reverse of the load order produced by Section 2.5. After all mmco_unload calls return, the host MAY release the underlying shared-library handles. Modules SHOULD NOT rely on the host calling dlclose, because RTLD_NODELETE may be in effect.

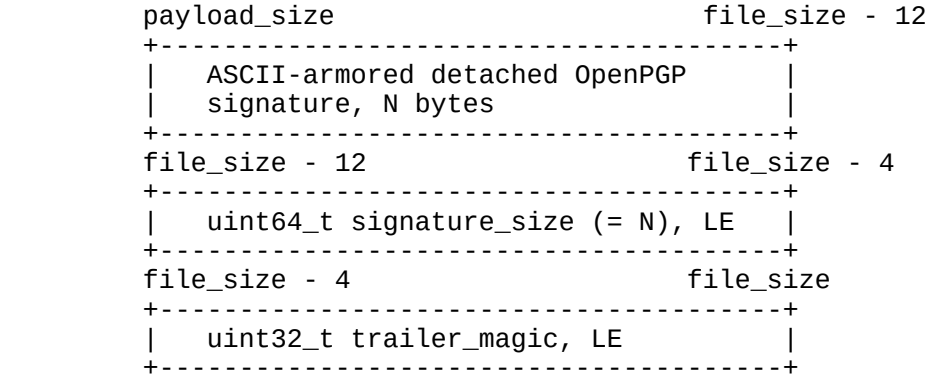
3. MMCS Composite Security

MMCS specifies how a detached OpenPGP signature is attached to an MMCO file, how the host extracts and verifies it, and the license-driven policy that decides whether a module is allowed to load.

MMCS is layered cleanly on top of MMCO: it never modifies the MMCO payload bytes that the operating system loader parses. An MMCO host that does not implement MMCS MAY load any module whose license is OSI-approved; conversely, an MMCS-aware host MUST refuse to load a module whose trailer is present but does not verify.

3.1. Trailer Layout

When an MMCS trailer is present, it is appended verbatim to the end of the MMCO payload and is structured as follows:



Constants:

Constant	Value	Notes
MMCO_TRAILER_MAGIC	0x53434D4D	ASCII "MMCS", little-endian.

Table 5

Both the signature_size field and the trailer_magic field MUST be stored little-endian on disk regardless of the host architecture.

The signature itself MUST be an OpenPGP detached signature, as defined by Section 5.2.3 (signature packets) and Section 11.4 (detached signatures) of [RFC4880] (or the equivalent sections of [RFC9580]). The signature MAY be in ASCII-armored form (Section 6 of

[RFC4880]) or in binary form. Producers SHOULD emit ASCII armor so that the trailer can be extracted and inspected with standard text tools; verifiers MUST accept both forms.

The signed data is exactly the contiguous byte range from offset 0 to payload_size; that is, the MMCO payload as written by the linker, with no transformation, no canonicalisation, and no inclusion of the trailer itself.

The combined size of the trailing footer is exactly 12 bytes (sizeof(uint64_t) + sizeof(uint32_t)).

3.2. Trailer Extraction

To extract an MMCS trailer from a file F, a host MUST:

1. Compute file_size. If file_size < 12, no trailer is present; the extraction terminates with status "absent".

2. Read the final 12 bytes of F and parse them as (uint64_t signature_size, uint32_t magic), little-endian. If magic != MMCO_TRAILER_MAGIC, no trailer is present; terminate with status "absent".

3. If signature_size == 0 or signature_size > file_size - 12, the trailer is corrupt; terminate with status "malformed".

4. Compute payload_size = file_size - 12 - signature_size. Read payload_size bytes from offset 0 (the payload) and the following signature_size bytes (the signature). If either short-reads, terminate with status "malformed".

5. Otherwise, terminate with status "present", returning the payload bytes and the signature bytes.

A trailer marked "malformed" MUST be treated by the policy (Section 3.4) as equivalent to BadSignature.

3.3. Signature Verification

A host MUST verify the extracted signature against the extracted payload bytes using an OpenPGP [RFC4880] implementation configured with a keyring of trusted public keys.

The keyring location is host-configurable. The reference implementation exposes a host setting (named plugin.signing.keyring_path in the reference INI schema) that specifies an OpenPGP home directory; when the setting is unset or the

empty string, the implementation falls back to whatever default the underlying OpenPGP backend uses (typically the GNUPGHOME environment variable, falling back to ~/.gnupg).

A host MAY further restrict trust beyond mere keyring membership — for example, by requiring a particular trust level or by pinning a specific fingerprint per module. Any such restriction MUST be expressed as the Untrusted state in Section 3.3, not as BadSignature, so that the license-driven policy in Section 3.4 treats the case uniformly.

The signed data passed to the OpenPGP verifier is exactly the payload bytes recovered by Section 3.2: the contiguous byte range from offset 0 to payload_size. The host MUST NOT apply any line-ending or text canonicalisation. The OpenPGP text-mode flag MUST NOT be set.

The result of verification MUST be classified into exactly one of the following terminal states, as observed by the policy layer (Section 3.4):

Absent: No trailer is present in the file.

Valid: The signature parses correctly, verifies against the payload, and was made by a key that is present in the trusted keyring with at least the "valid trust" level.

Untrusted: The signature parses correctly and verifies against the payload, but the signing key is not present in the trusted keyring (or is present below the configured trust threshold).

BadSignature: The signature does not verify against the payload, or the signature parses but is structurally invalid.

Malformed: The trailer footer matches MMCO_TRAILER_MAGIC but the trailer cannot be parsed (e.g. signature_size exceeds file bounds).

Error: The verification backend is unavailable (e.g. the OpenPGP daemon cannot be reached). This state MUST NOT be cached (see Section 3.6).

The host MUST treat BadSignature and Malformed identically for policy purposes.

3.4. License-Driven Policy

The MMCS verification policy is parameterised by:

* The result of trailer extraction and signature verification (Section 3.2, Section 3.3).

* Whether the module's license field resolves to an OSI-approved open-source license under Section 3.5.

The policy matrix is:

	Signature Valid	Absent	Untrusted	BadSignature/ Malformed	Error
OSS license	load	load	load	refuse, SignatureInvalid	load
Non-OSS / none	load	refuse, SignatureRequired	refuse, SignatureRequired	refuse, SignatureInvalid	refuse, SignatureRequired

Table 6

Justification:

* A Valid signature is always sufficient: identity has been cryptographically established and the host need not consult the license.

* For OSS modules, source is publicly available and reviewable; unsigned, untrusted, or backend-unavailable cases reduce to the conventional risk model for OSI-approved software distribution and are permitted.

* For non-OSS modules, identity attestation is required; any state that fails to establish identity (including a backend outage) MUST refuse the load.

* BadSignature and Malformed are hard failures regardless of license, because they indicate that an attacker may have tampered with the trailer of a known-good non-OSS module. Allowing them for OSS modules would create a downgrade attack in which a proprietary module is repackaged under an OSS SPDX identifier with an intentionally corrupt signature.

A module that is refused under this policy MUST be retained in the host's loaded-but-disabled set, so that administrative tooling can present a human-readable explanation. The host MUST NOT invoke mmco_init on a refused module.

3.5. SPDX Evaluation

The license check operates on the SPDX expression stored in the module's license field. NULL and the empty string MUST be treated as non-OSS.

The host MUST tokenise the expression as follows:

1. Convert the entire string to lowercase.

2. Replace parenthesis characters with spaces.

3. Split on whitespace runs. The tokens or and and are operator tokens; they partition the remaining tokens into atoms.

4. The token with is a non-operator token: it is concatenated into the current atom together with the atom on either side, so that gpl-3.0-or-later with classpath-exception-2.0 becomes a single atom rather than two.

An atom is considered OSS if it matches any identifier in the OSS allow-list, or if the head of a WITH-prefixed atom matches an identifier in the allow-list.

A module is OSS if at least one of its atoms is OSS.

The OSS allow-list MUST contain every identifier currently flagged "OSI Approved" in the SPDX License List [SPDX-LICENSE-LIST]. The allow-list MAY additionally contain a small set of widely-accepted non-OSI libre identifiers (e.g. cc0-1.0, unlicense, wtfpl, vim, bsd-3-clause-clear, bsd-4-clause). Implementations MUST NOT remove identifiers from the allow-list, because doing so retroactively breaks every module that ships under them.

The reference implementation's snapshot of the allow-list, taken from SPDX License List version 3.24, is reproduced in Section 5.3 of this document for unambiguous reproduction.

3.6. Verification Cache

A host MAY maintain a persistent cache of verification results, keyed on the tuple (absolute_path, file_size, file_mtime_milliseconds). Hosts that maintain such a cache MUST:

* Invalidate the entire cache whenever the host's trusted keyring is changed, because the same payload bytes may legitimately reach a different verdict under a different keyring.

* Refrain from caching the Error state. Transient backend outages MUST NOT poison subsequent runs.

* Treat any read or write failure on the cache as a missing entry rather than as a fatal error: cache faults MUST NOT prevent verification from running on the slow path.

The cache is RECOMMENDED to live under the host's per-user application-data directory (e.g. QStandardPaths::AppLocalDataLocation on Qt-based hosts, $XDG_DATA_HOME/<application>/ on bare XDG hosts, or %LOCALAPPDATA%\<application>\ on Windows). The reference implementation stores it as a file named plugin-signature-cache.json in that directory.

The cache schema is informational and is documented here for interoperability of administrative tooling. The reference implementation uses JSON with the following shape:

```
{
  "schema": 1,
  "entries": [
    {
      "path":        "<absolute path>",
      "size":        <integer bytes>,
      "mtime_ms":    <integer milliseconds since epoch>,
      "state":       "Valid|Untrusted|BadSignature|Malformed|Absent",
      "detail":      "<free-form text>",
      "fingerprint": "<OpenPGP key fingerprint, uppercase hex>"
    }
  ]
}
```

The host MUST allow a cache bypass on demand (for administrative "re-verify" operations). The reference implementation exposes this through the host's plugins-management UI.

3.7. Producing a Signed MMCO File

To produce a signed MMCO file from an unsigned payload P:

1. If P already ends in an MMCS trailer (as detected by reading the final 12 bytes and matching MMCO_TRAILER_MAGIC), strip it first, recovering the original payload bytes.

2. Compute a detached, ASCII-armored OpenPGP signature S over the stripped payload bytes, using the desired signing key.

3. Write the concatenation P || S || uint64_LE(len(S)) || uint32_LE(MMCO_TRAILER_MAGIC) as the signed file.

This procedure is idempotent: applying it twice to the same file with the same key yields a file whose signature covers the same payload bytes (subject to OpenPGP signature non-determinism in the timestamp field).

Tools that re-sign installed modules after the operating-system installer has rewritten in-payload bytes (RPATH rewrites, debug symbol stripping, code-signing surgery) MUST run after all such rewrites complete. A signature computed before such rewrites will always be invalidated.

4. Interactions

The host MUST execute the following sequence at startup, in order, for each search directory:

1. Discover candidates (Section 2.3).

2. Open the shared library and read mmco_module_info (Section 2.4).

3. Validate magic and abi_version.

4. Resolve mmco_init, mmco_unload.

5. Extract the MMCS trailer (Section 3.2). Consult the cache (Section 3.6) first; on miss, run signature verification and memoise the terminal state (except Error).

6. Apply the license-driven policy (Section 3.4) and, if it refuses the module, mark it disabled.

7. After all candidates have completed steps 16 across all directories, run dependency resolution (Section 2.5) over the union, propagating disable states.

8. Call mmco_init exactly once per enabled module in resolver order.

During host shutdown, the inverse: call mmco_unload in reverse resolver order, then release shared-library handles (subject to the RTLD_NODELETE caveat in Section 2.4).

A host MUST NOT call mmco_init on any module marked disabled by any step.

5. IANA Considerations

This section requests two registrations and defines a third internal registry.

5.1. File-Extension and Magic-Number Registration

This document defines the file extension .mmco and the four-byte magic number 0x4D, 0x4D, 0x43, 0x4F (ASCII "MMCO") at offset 0 of an MMCO file. Note that this magic number is in the host operating system’s shared-object header, not at offset 0 of the MMCO file itself; offset 0 of the MMCO file is whatever the host OS’s shared-object format requires. The MMCO identity is instead expressed via the embedded mmco_module_info::magic field. No collision with existing offset-0 magic numbers is therefore expected.

This document also defines the four-byte magic number 0x4D, 0x4D, 0x43, 0x53 (ASCII "MMCS") as the trailing magic of an MMCS trailer. Because this magic appears at the end of an MMCO file, it does not collide with offset-0 registrations.

5.2. MMCO Hook Identifier Registry

This document defines an internal registry of MMCO hook identifiers. Hook identifiers are 32-bit unsigned integers in the range 0x00000xFFFF. Allocations as of MMCO_ABI_VERSION == 2:

Range	Status	Reference
0x00000x00FF	Reserved	This document
0x01000x01FF	Allocated	Application lifecycle
0x02000x02FF	Allocated	Instance lifecycle
0x03000x03FF	Allocated	Settings
0x04000x04FF	Allocated	Content and mod management
0x05000x05FF	Allocated	Network
0x06000x06FF	Allocated	UI extension points
0x07000x07FF	Allocated	News
0x08000x08FF	Allocated	Authentication
0x09000xFEFF	Unassigned	Future use
0xFF000xFFFF	Reserved	Private experiments

Table 7

New range allocations MUST be made in 256-identifier blocks aligned on a multiple of 0x0100. Allocation requires a published description of the payload struct(s), the cancellation semantics (for any identifier whose name contains "PRE"), and the lifecycle constraints under which the host fires the hook.

5.3. SPDX OSI Allow-List Snapshot

This is a snapshot of the OSI-approved SPDX identifiers recognised by the reference MMCS implementation, taken from SPDX License List 3.24. Identifiers are lowercase.

0bsd, aal, afl-1.1, afl-1.2, afl-2.0, afl-2.1, afl-3.0, agpl-3.0, agpl-3.0-only, agpl-3.0-or-later, apache-1.1, apache-2.0, apl-1.0, apsl-1.0, apsl-1.1, apsl-1.2, apsl-2.0, artistic-1.0, artistic-1.0-cl8, artistic-1.0-perl, artistic-2.0, bsd-1-clause, bsd-2-clause, bsd-2-clause-patent, bsd-3-clause, bsd-3-clause-lbnl, bsd-3-clause-modification, bsl-1.0, cal-1.0, cal-1.0-combined-work-exception, catosl-1.1, cddl-1.0, cddl-1.1, cecill-2.1, cern-ohl-p-2.0, cern-ohl-s-2.0, cern-ohl-w-2.0, cnri-python, cpal-1.0, cpl-1.0, cua-opl-1.0,

ecl-1.0, ecl-2.0, efl-1.0, efl-2.0, entessa, epl-1.0, epl-2.0, eudatagrid, eupl-1.1, eupl-1.2, fair, frameworx-1.0, gpl-2.0, gpl-2.0-only, gpl-2.0-or-later, gpl-3.0, gpl-3.0-only, gpl-3.0-or-later, hpnd, intel, ipa, ipl-1.0, isc, jam, lgpl-2.0, lgpl-2.0-only, lgpl-2.0-or-later, lgpl-2.1, lgpl-2.1-only, lgpl-2.1-or-later, lgpl-3.0, lgpl-3.0-only, lgpl-3.0-or-later, liliq-p-1.1, liliq-r-1.1, liliq-rplus-1.1, lppl-1.3c, miros, mit, mit-0, mit-modern-variant, motosoto, mpl-1.0, mpl-1.1, mpl-2.0, mpl-2.0-no-copyleft-exception, ms-pl, ms-rl, mulanpsl-2.0, multics, nasa-1.3, naumen, ncsa, nokia, nposl-3.0, ntp, ofl-1.1, ofl-1.1-no-rfn, ofl-1.1-rfn, oldap-2.8, oset-pl-2.1, osl-1.0, osl-2.0, osl-2.1, osl-3.0, php-3.0, php-3.01, postgresql, python-2.0, python-2.0.1, qpl-1.0, rpl-1.1, rpl-1.5, rpsl-1.0, rscpl, simpl-2.0, sissl, sleepycat, spl-1.0, ucl-1.0, upl-1.0, vsl-1.0, w3c, w3c-19980720, w3c-20150513, watcom-1.0, xnet, zlib, zpl-2.0, zpl-2.1.

Additionally, the reference implementation accepts the following non-OSI but widely-accepted libre identifiers as OSS:

bsd-3-clause-clear, bsd-4-clause, cc0-1.0, cc-by-4.0, cc-by-sa-4.0, unlicense, vim, wtfpl.

This snapshot is informative. The normative requirement is in Section 3.5: the allow-list MUST track the OSI-Approved column of the SPDX License List, MUST be additive, and MAY include the above-listed libre identifiers.

6. Security Considerations

6.1. Trust Model

The MMCO host loads native code into its own address space. A loaded module has full process authority; in particular, it can read every file the host can read, open every socket the host can open, and modify every host-owned data structure reachable through the language runtime. MMCS therefore cannot, and does not, provide post-load isolation. Its threat model is solely about authorising the load decision.

MMCS authorises a load by tying the module bytes to an OpenPGP identity. The host’s trust root is its configured keyring; the strength of the chain is exactly the strength of the user’s process for adding keys to that keyring. The host SHOULD document its keyring location and SHOULD provide tooling that displays the fingerprint of every signing key encountered.

6.2. Tamper Resistance

Because the MMCS trailer is appended to the operating system's native shared-object format, it survives:

* Read by the operating-system loader, which ignores trailing bytes past the program-header-referenced sections.

* Hash verification by the host, which reads the file directly.

It does not survive operations that rewrite payload bytes:

* ELF strip(1), patchelf, chrpath, install_name_tool, and other link-time rewriters that mutate sections covered by the program header.

* Debug-symbol stripping during installation, when CMAKE_INSTALL_DO_STRIP=ON or its equivalent is in effect.

* Repacking by debugger-friendly post-processing tools.

A producer MUST re-sign an MMCO file after any such rewrite, and SHOULD arrange its installer to defer signing until all rewrites have completed. The reference implementation accomplishes this by attaching the trailer twice during a build: once as a POST_BUILD step on the build artefact (so an in-tree run sees a signed module), and once again from an install(CODE …) block that runs after CMake has finished its RPATH-rewriting and stripping on the installed file. Each re-signing step strips any pre-existing MMCS trailer before attaching the new one (see Section 3.7); the operation is idempotent.

6.2.1. Interaction with Platform Code Signing

The MMCS trailer is incompatible with platform code-signing schemes that themselves claim the tail of a shared object:

* On macOS, codesign(1) writes its signature into the Mach-O __LINKEDIT segment by extending the segment past its original end. After codesign runs, any MMCS trailer previously attached is no longer at end-of-file, breaking extraction. Furthermore, codesign --strict (the default in notarised builds) refuses to validate a Mach-O whose file tail contains bytes not accounted for by the Mach-O load commands.

  The conflict has two practical consequences:

1. An MMCO module on macOS MUST NOT be signed with both MMCS and Apple codesign in the same file. A host MAY choose either signing scheme; this document specifies MMCS, and a host that chooses Apple codesign instead is out of scope.

2. An MMCO host on macOS MUST install module files into a bundle directory that codesign does not walk recursively in code mode. The bundle's Contents/Resources/ directory satisfies this requirement; Contents/PlugIns/ and Contents/MacOS/ do not (see Section 2.3).

* On Windows, Authenticode signatures live in the PE optional header's IMAGE_DIRECTORY_ENTRY_SECURITY directory, which is located by an offset stored elsewhere in the header. An MMCS trailer appended after the Authenticode signature does not by itself invalidate the Authenticode signature, but it does enlarge the file past the signature's claimed extent and may cause some Authenticode verifiers (notably WDAC) to reject the file. Hosts that target Windows SHOULD pick exactly one scheme per module.

A host MUST NOT attempt to recover from a BadSignature or Malformed state by truncating the trailer and re-loading; doing so would convert any verified-then-tampered module into an apparently unsigned one and bypass Section 3.4 for any module whose declared license was downgraded to an OSS SPDX identifier in the same tampering step.

6.3. Downgrade Attacks

An attacker controlling distribution of a module can, in principle:

1. Replace a non-OSS module's license field with an OSS SPDX identifier, attach no trailer, and rely on Section 3.4 to admit the module. This is mitigated by the host displaying the declared license in administrative UI and by the user reviewing that license before installing the module. It is NOT prevented by MMCS, which does not encode license commitments cryptographically.

2. Wrap an unsigned non-OSS module in a forged trailer to convert Absent into BadSignature. This is detected; BadSignature is a hard refusal under Section 3.4 regardless of license.

3. Replace a Valid trailer with a trailer signed by a different key in the user's trusted keyring. This is prevented by the host displaying the fingerprint of the signing key and by the user adding only keys they have verified out-of-band to their keyring. MMCS does not authenticate which key SHOULD have signed a given module; the signing_key_id field in MMCOModuleInfo is informational.

6.4. Cache Poisoning

The verification cache (Section 3.6) is keyed on (path, size, mtime_ms). An attacker who can write to a module file and reset its mtime can in principle reuse a previously cached Valid verdict against new payload bytes. Mitigations:

* The host SHOULD store the cache under a path the unprivileged attacker cannot write to (e.g. the user's configuration directory).

* A host MAY harden the cache key by including a strong digest of the payload, at the cost of one full file read per launch.

* The host MUST provide a cache-bypass administrative command.

6.5. Backend Failure

When the OpenPGP backend is unavailable, the host receives Error. Section 3.4 requires that non-OSS modules be refused in this state. This is intentional: a process that cannot establish identity for a module that requires identity MUST NOT execute that module's code. OSS modules are admitted in this state because the alternative is to make the entire plugin system non-functional whenever the OpenPGP daemon is missing (notably on stripped-down container images), which would push users toward disabling MMCS entirely.

6.6. Static Constructors and RTLD_NODELETE

The reference host opens modules with RTLD_NODELETE on ELF platforms, for the reasons detailed in Section 2.4.1. This is a load-time decision that prevents the C runtime from running a module's global destructors at process exit (or at a subsequent dlclose). Without this flag, a module that statically links any object whose destructor mutates host-shared state would corrupt that state during shutdown.

This is a correctness measure rather than a security measure; it is documented in the security section because failing to observe it manifests as exit-time heap corruption that an adversary may be able to weaponise into a use-after-free. Hosts on platforms without

RTLD_NODELETE (notably Windows, where FreeLibrary always runs destructors) SHOULD achieve the same outcome architecturally — typically by ensuring that plugin authors do not statically link libraries whose globals the host also instantiates, and by exposing the host's own copies of those globals through the API surface (Section 2.2.7) instead.

7. Acknowledgements

MMCO and MMCS were designed for, and are implemented in, the MeshMC launcher of the Project Tick BSD/Linux distribution. The authors acknowledge the prior art of the Linux kernel's signed module facility (modsign), the macOS Mach-O code-signing trailer, and the systemd Portable Service signature attached to disk images, all of which informed the trailer layout.

8. References

8.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC4880] Callas, J., Donnerhacke, L., Finney, H., Shaw, D., and R. Thayer, "OpenPGP Message Format", RFC 4880, DOI 10.17487/RFC4880, November 2007, <https://www.rfc-editor.org/rfc/rfc4880>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

[RFC9580] Wouters, P., Ed., Huigens, D., Winter, J., and Y. Niibe, "OpenPGP", RFC 9580, DOI 10.17487/RFC9580, July 2024, <https://www.rfc-editor.org/rfc/rfc9580>.

8.2. Informative References

[ELF] Tool Interface Standards (TIS) Committee, "Executable and Linkable Format (ELF) Specification, Version 1.2", May 1995.

[KAHN-TOPO] Kahn, A. B., "Topological sorting of large networks", Communications of the ACM 5(11), pp. 558-562, 1962.

[MACHO] Apple Inc., "Mach-O Programming Topics", n.d., <https://www.cs.miami.edu/home/burt/learning/Csc521.091/docs/MachOTopics.pdf>.

[OSI] Open Source Initiative, "OSI Approved Licenses", n.d., <https://opensource.org/licenses/>.

[PE] Microsoft Corporation, "PE Format", n.d., <https://learn.microsoft.com/windows/win32/debug/pe-format>.

[POSIX] IEEE / The Open Group, "IEEE Std 1003.1-2017, Standard for Information Technology -- Portable Operating System Interface (POSIX) Base Specifications, Issue 7", 2017.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <https://www.rfc-editor.org/rfc/rfc5234>.

[SPDX] Linux Foundation, "SPDX Specification, Version 2.3", n.d., <https://spdx.github.io/spdx-spec/>.

[SPDX-LICENSE-LIST] Linux Foundation, "SPDX License List", n.d., <https://spdx.org/licenses/>.

Appendix A. Implementation Notes (Informative)

A.1. Reference Constants

For unambiguous interoperability, the reference implementation defines the following preprocessor constants. They are reproduced here verbatim:

```
#define MMCO_MAGIC          0x4D4D434F  /* "MMCO"           */
#define MMCO_ABI_VERSION    2
#define MMCO_EXTENSION      ".mmco"
#define MMCO_FLAG_NONE      0x00000000

#define MMCO_TRAILER_MAGIC  0x53434D4D  /* "MMCS"           */
```

The reference implementation additionally exposes a four-nibble version number for tooling convenience:

```
#define MMCO_VERSION        "8.0.0"
#define MMCO_VERNUM         0x08000000L
#define MMCO_VER_MAJOR      8
#define MMCO_VER_MINOR      0
#define MMCO_VER_REVISION   0
```

These are not normative; they describe the implementation's release identity, not the format itself.

A.2. Reference Pseudocode for Trailer Verification

The following pseudocode summarises Section 3.2 through Section 3.3:

```
function verify_file(path, keyring) -> (state, detail, fpr):
    if cache.has((path, size_of(path), mtime_ms_of(path))):
        return cache.get(...)
    f = open(path)
    if size_of(f) < 12:
        return ("Absent", "", "")
    seek(f, -12, end)
    sig_size, magic = read_u64_le_u32_le(f)
    if magic != 0x53434D4D:
        return ("Absent", "", "")
    if sig_size == 0 or sig_size > size_of(f) - 12:
        return ("Malformed", "malformed trailer", "")
    payload_size = size_of(f) - 12 - sig_size
    seek(f, 0)
    payload = read(f, payload_size)
    signature = read(f, sig_size)
    if len(payload) != payload_size or len(signature) != sig_size:
        return ("Malformed", "short read", "")
    (state, detail, fpr) = openpgp_verify_detached(
        signature, payload, keyring)
    if state != "Error":
        cache.put((path, size, mtime_ms), (state, detail, fpr))
    return (state, detail, fpr)
```

A.3. Cross-References to the Reference Implementation

For maintainers of independent implementations, the canonical source files in the reference implementation are:

* launcher/plugin/MMCOFormat.h — ABI constants, MMCOModuleInfo, MMCODependency, trailer magic.

* launcher/plugin/PluginAPI.h — MMCOContext definition.

* launcher/plugin/PluginHooks.h — hook identifier enum and payload structures.

* launcher/plugin/PluginLoader.cpp — discovery, loading, trust pre-flight.

* launcher/plugin/PluginDependencyResolver.cpp — Kahn's algorithm, version comparison.

* launcher/plugin/PluginSignature.cpp — trailer extraction, OpenPGP verification, persistent cache, SPDX allow-list.

* launcher/plugin/PluginManager.cpp — MMCOContext build-out, hook dispatch, per-module sandboxing, lifecycle.

* scripts/mmco_sign.py — idempotent signing of an MMCO file.

* cmake/MMCOSign.cmake — build-time signing and install-time re-signing.

These files are made available under the GPL-3.0-or-later license, with an additional permission described in the MeshMC MMCO Module Exception 1.0.

A.4. Reference Build-Time Switches

The reference implementation exposes the following build-time configuration knobs, all of which are informative and have no bearing on the on-the-wire format:

MeshMC_PLUGIN_SIGNATURES (CMake option): Enables linkage against the GpgME C++ bindings and selects the full MMCS verifier. When OFF, PluginSignature.cpp compiles into a stub that reports every module's signature state as Error — under Section 3.4 this admits OSS modules and rejects non-OSS modules. The option defaults OFF on toolchains where upstream GpgME does not build (MSVC) and on macOS universal builds (where Homebrew ships single-arch dylibs that cannot link into a fat binary), and ON elsewhere.

MeshMC_PLUGIN_SIGNING_KEY / MeshMC_PLUGIN_SIGNING_HOMEDIR (CMake

cache variables): Default GPG fingerprint and GnuPG home directory used by MMCOSign.cmake when signing built modules. When empty, the POST_BUILD signing step is skipped.

MeshMC_PLUGIN_SIGN_ALL (CMake option): When ON, every in-tree MMCO plugin target is signed automatically using the above key.

MESHMC_SKIP_INSTALL_RESIGN (environment variable, build-time): When set to 1, suppresses the install-time re-signing step described in Section 6.2. Useful for distribution packagers who lack the upstream signing key. The installed module will then carry a build-tree signature that has been invalidated by the installer's RPATH / strip operations, and will be rejected at load time as BadSignature per Section 3.4.

A.5. Reference Hook Payload Definitions

The complete normative listing of hook identifiers and their payload struct layouts is given as Appendix B in this document (Appendix B). The reference implementation's PluginHooks.h header is the authoritative C declaration for those structs and MUST be kept in sync with Appendix B for every MMCO_ABI_VERSION revision.

Appendix B. Hook Payload Definitions

This appendix defines, normatively, every hook identifier allocated as of MMCO_ABI_VERSION == 2 and the C-language layout of its payload struct. The host MUST pass a payload pointer that points to a struct of the indicated type for the corresponding hook identifier; the payload pointer MAY be NULL only for hooks whose payload type is listed as "none".

B.1. Application Lifecycle (0x01000x01FF)

Identifier	Value	Payload	Cancellable	Dispatched?
MMCO_HOOK_APP_INITIALIZED	0x0100	none	no	yes
MMCO_HOOK_APP_SHUTDOWN	0x0101	none	no	yes

Table 8

B.2. Instance Lifecycle (0x02000x02FF)

All four payloads are of type MMCOInstanceInfo:

```
struct MMCOInstanceInfo {
    const char *instance_id;
    const char *instance_name;
    const char *instance_path;
    const char *minecraft_version; /* MAY be NULL */
};
```

Identifier	Value	Payload	Cancellable	Dispatched?
MMCO_HOOK_INSTANCE_PRE_LAUNCH	0x0200	MMCOInstanceInfo*	no	yes
MMCO_HOOK_INSTANCE_POST_LAUNCH	0x0201	MMCOInstanceInfo*	no	yes
MMCO_HOOK_INSTANCE_CREATED	0x0202	MMCOInstanceInfo*	no	not yet
MMCO_HOOK_INSTANCE_REMOVED	0x0203	MMCOInstanceInfo*	no	not yet

Table 9

Note: MMCO_HOOK_INSTANCE_PRE_LAUNCH is observational despite its name; see Section 2.2.8. The host applies launch modifications (environment variables and wrapper commands) submitted via the side-channel API after the hook chain finishes, regardless of the dispatch return value.

B.3. Settings (0x03000x03FF)

```
struct MMCOSettingChange {
    const char *key;
    const char *old_value;
    const char *new_value;
};
```

Identifier	Value	Payload	Cancellable	Dispatched?
MMCO_HOOK_SETTINGS_CHANGED	0x0300	MMCOSettingChange*	no	not yet

Table 10

B.4. Content / Mod Management (0x04000x04FF)

```
struct MMCOContentEvent {
    const char *instance_id;
    const char *file_name;
    const char *url;
    const char *target_path;
};
```

Identifier	Value	Payload	Cancellable	Dispatched?
MMCO_HOOK_CONTENT_PRE_DOWNLOAD	0x0400	MMCOContentEvent*	yes (when dispatched)	not yet
MMCO_HOOK_CONTENT_POST_DOWNLOAD	0x0401	MMCOContentEvent*	no	not yet

Table 11

B.5. Network (0x05000x05FF)

```
struct MMCONetworkEvent {
    const char *url;
    const char *method;     /* "GET", "POST", ...        */
    int         status_code; /* 0 for pre-request events */
};
```

Identifier	Value	Payload	Cancellable	Dispatched?
MMCO_HOOK_NETWORK_PRE_REQUEST	0x0500	MMCONetworkEvent*	yes (when dispatched)	not yet
MMCO_HOOK_NETWORK_POST_REQUEST	0x0501	MMCONetworkEvent*	no	not yet

Table 12

B.6. UI Extension Points (0x06000x06FF)

```
   struct MMCOUiMainReadyPayload {
       void *main_window;       /* opaque host UI handle    */
       void *news_toolbar;
       void *more_news_action;
       void *news_label_button;
   };

   struct MMCOMenuEvent {
       const char *context;     /* e.g. "main", "instance"  */
       void       *menu_handle; /* opaque host UI handle    */
   };

   struct MMCOInstancePagesEvent {
       const char *instance_id;
       const char *instance_name;
       const char *instance_path;
       void       *page_list_handle; /* opaque host UI handle */
       void       *instance_handle;  /* opaque host data handle */
   };

   struct MMCOGlobalSettingsPagesEvent {
       void *page_list_handle;       /* opaque host UI handle */
   };
```

Identifier	Value	Payload	Cancellable	Dispatched?
MMCO_HOOK_UI_MAIN_READY	0x0600	MMCOUiMainReadyPayload*	no	yes
MMCO_HOOK_UI_CONTEXT_MENU	0x0601	MMCOMenuEvent*	no	yes
MMCO_HOOK_UI_INSTANCE_PAGES	0x0602	MMCOInstancePagesEvent*	no	yes
MMCO_HOOK_UI_GLOBAL_SETTINGS_PAGES	0x0603	MMCOGlobalSettingsPagesEvent*	no	not yet

Table 13

The opaque handle fields are pointers to host-internal UI objects. Modules MUST NOT dereference them directly; they MUST be passed only to host API functions that document them as acceptable inputs.

B.7. News (0x07000x07FF)

Identifier	Value	Payload	Cancellable	Dispatched?
MMCO_HOOK_NEWS_UPDATED	0x0700	none	no	yes

Table 14

B.8. Authentication (0x08000x08FF)

```
struct MMCOAuthRequestEvent {
    /* Read-only request snapshot */
    const char *url;
    const char *method;       /* "GET" | "POST" | ...     */
    const char *body;         /* MAY be NULL for GETs     */
    int         body_size;

    /* Mutable response slots */
    const char *redirect_url; /* set non-NULL to rewrite  */

    /* Header-injection helper */
    void *request_handle;
    int (*add_header)(void *request_handle,
                      const char *key,
                      const char *value);
};

struct MMCOSessionFillEvent {
    /* Read-only context */
    const char *account_id;
    int         account_is_msa;
    int         wants_online;

    /* Read-only current session view */
    const char *current_player_name;
    const char *current_uuid;
    const char *current_user_type;

    /* Mutable overwrites (NULL = leave alone) */
    const char *overwrite_access_token;
    const char *overwrite_session;
    const char *overwrite_player_name;
    const char *overwrite_uuid;
    const char *overwrite_user_type;
    const char *overwrite_client_token;

    /* Mutable user-properties append (NULL = none) */
    const char *extra_user_properties;
};
```

Identifier	Value	Payload	Cancellable	Dispatched?
MMCO_HOOK_AUTH_REQUEST	0x0800	MMCOAuthRequestEvent*	yes	yes
MMCO_HOOK_SESSION_FILL	0x0801	MMCOSessionFillEvent*	no	yes

Table 15

For MMCO_HOOK_AUTH_REQUEST, cancellation aborts the in-flight authentication request and causes the calling authentication step to fail with a protocol error.

For MMCO_HOOK_SESSION_FILL, the host copies every non-NULL overwrite_* field into the in-flight session before launching; NULL values leave the host-populated default in place. The return value of the callback is reserved and SHOULD be zero. String pointers placed in overwrite_* and extra_user_properties are read by the host during the callback only; the host copies their contents.

B.9. Reserved Ranges

Identifiers in the ranges 0x00000x00FF, 0x09000xFEFF, and 0xFF000xFFFF are not allocated by this revision. Hosts MUST ignore registrations for unallocated identifiers (i.e. accept the registration call and silently never invoke the callback) so that newer modules remain loadable on older hosts.

Author’s Address

Mehmet Samet Duman
Project Tick
Email: yongdohyun@projecttick.org
URI: https://projecttick.org/