

FIPS 140-3 Compliance Guide

Maverics Identity Orchestrator

Rubrik, Inc.

September 2026

Version v2026.10.1

Contents

1 About this guide **3**
- 1.1 Purpose . . . 3
- 1.2 Audience . . . 3
- 1.3 Conventions . . . 3
- 1.4 Notices . . . 4

Part I: Deploying a FIPS build **5**

2 The Orchestrator's compliance posture **6**
- 2.1 The validated module . . . 6
- 2.2 Shared responsibility model . . . 6

3 Covered operational environments **8**
- 3.1 Tested versus vendor-affirmed . . . 8
- 3.2 Coverage for the platforms Rubrik supports . . . 8

4 Obtaining and verifying a FIPS build **10**
- 4.1 Verification procedure . . . 10
- 4.2 Reading the heartbeat . . . 10
- 4.3 Setting `GODEBUG` explicitly . . . 11

Part II: Configuration required for compliance **12**

5 Transport layer security **13**
- 5.1 Cipher suites . . . 13
- 5.2 Key exchange groups . . . 13
- 5.3 Certificate verification . . . 14
- 5.4 Certificate and key requirements . . . 14
- 5.5 Certificate revocation . . . 15

6 OIDC (OpenID Connect), OAuth, and JWTs (JSON Web Tokens) **16**
- 6.1 Accepted signature algorithms . . . 16
- 6.2 Signing . . . 16
- 6.3 Key requirements . . . 17
  - 6.3.1 Keys the Orchestrator signs with . . . 17
  - 6.3.2 Keys the Orchestrator verifies clients against . . . 17
  - 6.3.3 Keys the Orchestrator verifies the identity provider against . . . 17
- 6.4 Certificate chains in a retrieved JWKS . . . 17

6.5 Tokens carrying an embedded key with a certificate chain (DPoP) 18

7 SAML (Security Assertion Markup Language) 19
7.1 Configuration validated at startup 19
7.2 Outbound signing 19
7.3 Inbound signatures 20
7.4 What your SAML peers must do 21

8 Directory services 22
8.1 Transport (LDAP) 22
8.2 Authentication mechanisms 22
8.3 What your directory must support 22

9 Secret providers 24
9.1 Client certificates 24

10 Authorization policy (Rego) 25
10.1 Writing policies for approved mode 25

11 Redis cache encryption 26
11.1 Key rotation 26

Part III: Service Extension compliance 27

12 Service extensions 28
12.1 Packages withheld in approved mode 28
12.2 Symbols withheld from packages that remain 29
12.2.1 The `crypto` package still fails on these identifiers 30
12.3 Approved alternatives 31
12.3.1 Directory lookup and authentication 31
12.3.2 Hashing 31
12.3.3 UUIDs 32
12.3.4 Symmetric encryption 32
12.3.4.1 AES-GCM 32
12.3.5 Signatures and tokens 33
12.4 Extension compliance gaps 33
12.4.1 Do not implement cryptography yourself 34
12.4.2 Do not build a `crypto.Hash` by arithmetic or conversion 34
12.4.3 Do not pass non-approved parameters to approved calls 34
12.5 What remains available 35

Appendices 36

Appendix A: Approved algorithm reference 37
Cryptography the Orchestrator performs 37
Cryptography the Orchestrator accepts 37

Appendix B: References 39

Chapter 1

About this guide

1.1 Purpose

This guide tells you how to deploy and operate the Maverics Identity Orchestrator so that your deployment is compliant with FIPS 140-3.

1.2 Audience

This guide is written for three readers:

1. **Administrators and operators** deploy, configure, and run the Orchestrator in an environment subject to FIPS 140-3 requirements. Work through Parts I and II, which cover how to obtain and verify a FIPS build and what you must configure.
2. **Compliance officers, assessors, and authorizing officials** need to establish what cryptographic claim the Orchestrator supports, on what basis, and with what boundaries. Start with Part I for the module and shared-responsibility boundary, then Part II for where that boundary falls in each subsystem; Appendix B collects the primary sources.
3. **Service extension authors**, whose code carries compliance obligations that configuration does not, should read Service extensions in Part III for what those obligations are.

The guide assumes familiarity with the Orchestrator's configuration model and with your own identity infrastructure.

1.3 Conventions

Convention	Meaning
Standard build	The default Orchestrator build. No FIPS constraints.
FIPS build	An Orchestrator binary compiled against the validated module.
The module / the validated module	Shorthand for the **Go Cryptographic Module**, the FIPS 140-3 validated module (certificate #5247) the FIPS build performs its cryptography through; see The validated module.

Convention	Meaning
Approved mode	The state in which the Orchestrator enforces its own FIPS-specific checks against configuration and inputs. It requires only that FIPS 140-3 mode be enabled. Approved mode is the Orchestrator's own enforcement, not the module's state itself.
Maverics	Short for the Maverics Identity Orchestrator. The binary is named `maverics`.

1.4 Notices

Copyright © 2026 Rubrik, Inc. All rights reserved.

This document may be used free of charge. Selling without prior written consent is prohibited. In all cases, this copyright notice and disclaimer must remain intact.

THE CONTENTS OF THIS DOCUMENT ARE PROVIDED "AS IS," AND COPYRIGHT HOLDERS MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, OR TITLE; THAT THE CONTENTS OF THE DOCUMENT ARE SUITABLE FOR ANY PURPOSE; THAT THE IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS. COPYRIGHT HOLDERS WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY USE OF THE DOCUMENT OR THE PERFORMANCE OR IMPLEMENTATION OF THE CONTENTS THEREOF.

Rubrik is a registered trademark of Rubrik, Inc. All company names, product names, and other such names in this document are registered trademarks or trademarks of the relevant company.

Part I: Deploying a FIPS build

Chapter 2

The Orchestrator's compliance posture

2.1 The validated module

The Orchestrator's FIPS build performs its own cryptographic operations through the **Go Cryptographic Module**, a FIPS 140-3 validated module maintained by Geomys and distributed as part of the Go standard library.

Property	Value
Module name	Go Cryptographic Module
Module version	v1.0.0
Vendor	Geomys LLC
Cryptographic Module Validation Program (CMVP) certificate	csrc.nist.gov: certificate 5247
Security policy	csrc.nist.gov: 140sp5247.pdf

The module version is pinned when the Orchestrator binary is compiled at `GOFIPS140=v1.0.0`. Approved mode is enabled by default in a FIPS build.

> ⓘ Note
>
> The module's security policy is the authoritative statement of what the module does, which algorithms it implements, and under what conditions its validation applies. Where this guide and the security policy appear to disagree, the security policy governs; please report any discrepancies to Rubrik.

2.2 Shared responsibility model

The Orchestrator deployment (the binary, its configuration, and the environment it runs in) is FIPS 140-3 compliant when all four of the following hold.

#	Condition	Whose responsibility	Where addressed
a	Built against the validated cryptographic module	**Rubrik** (a property of the shipped binary)	Obtaining and verifying a FIPS build

#	Condition	Whose responsibility	Where addressed
b	Operates the module in its approved mode of operation (FIPS 140-3 mode enabled)	**Rubrik** (enabled by default in the FIPS build)	Obtaining and verifying a FIPS build
c	Runs on a covered operational environment	**You** (a property of where you deploy)	Covered operational environments
d	Has its non-compliant features disabled or unused	**You** (a property of how you configure)	Part II

Conditions (a) and (b) are satisfied by obtaining and running a genuine FIPS build as shipped. Rubrik has done that work and Part I tells you how to confirm it.

Conditions (c) and (d) cannot be satisfied by shipping a binary. They are properties of your operational environment and your configuration.

Your responsibility also extends to the cryptographic posture of every system the Orchestrator talks to, and any extension code you supply.

Chapter 3

Covered operational environments

FIPS 140-3 validation is granted against specific **operational environments**: combinations of operating system, hardware platform, and processor architecture on which the module was tested or for which the vendor has affirmed equivalence. Running a validated module outside its covered environments does not carry the validation.

3.1 Tested versus vendor-affirmed

Tested operational environments were exercised by the accredited laboratory during validation and are named on the certificate. They carry the strongest form of coverage. A tested entry pins three things together: the operating system, the hardware platform, and the processor.

Vendor-affirmed operational environments are environments the module vendor has asserted are equivalent to a tested one, without independent laboratory testing. This is an accepted mechanism under the CMVP's implementation guidance. Acceptance varies: not every accreditor accepts vendor affirmation.

> ⓘ Note
>
> The vendor of certificate #5247 is Geomys LLC, not Rubrik. Rubrik is a consumer of the module and is not the party that affirms an operational environment. Every affirmed environment is Geomys's assertion, published in their security policy.

The authoritative list of both operational environments is the module's security policy. Table 3 lists tested operational environments and Table 4 vendor-affirmed ones.

The security policy is included in the FIPS release bundle as `GeomysCMVPSecurityPolicy5247.pdf`. It is sourced from `csrc.nist.gov` as `140sp5247.pdf` (dated June 12, 2026).

3.2 Coverage for the platforms Rubrik supports

Rubrik supports Orchestrator deployments on RHEL 8 and later, Ubuntu 22.04 and later, and Debian 12 and later, all on `amd64`. Non-Linux platforms and non-amd64 architectures are out of scope, regardless of what the certificate covers.

Three things follow from Rubrik not being the module vendor:

1. **Contact Rubrik if you need an operational environment that is not listed.** Only Geomys, as the module vendor, can affirm an operational environment. Rubrik works with Geomys to have additional environments vendor-affirmed or tested.
2. **Do not infer coverage from the existence of a build.** That Rubrik ships a FIPS build for a platform and that certificate #5247 covers it are independent facts.
3. **Vendor-affirmed is not the same as tested.** If your compliance program requires a tested environment and your platform is not in Table 3, contact Rubrik before deploying.

The security policy states the consequence directly:

> CMVP makes no statement as to the correct operation of the module or the security strengths of the generated keys when so ported if the specific operational environment is not listed on the validation certificate.

If your target platform appears in neither table, treat it as uncovered and contact Rubrik. Do not assume a near neighbor in the table extends to it.

Chapter 4

Obtaining and verifying a FIPS build

4.1 Verification procedure

After downloading the FIPS release bundle from the Maverics Console, verify its integrity before installing, then confirm it is a genuine FIPS build and that approved mode is running before putting it into service.

1. Confirm the SHA-256 checksum of the downloaded zip matches the value shown in the Maverics Console:

   ```
   sha256sum maverics_fips.<version>.zip
   ```

2. Unzip the bundle and verify the artifacts inside against its `checksums.txt`:

   ```
   sha256sum -c checksums.txt
   ```

3. Install following the platform installation instructions.
4. Run `maverics -version` and confirm it reports `(FIPS 140-3)`.
5. Start the Orchestrator and confirm the startup log line `starting Maverics in FIPS 140-3 mode`.
6. Confirm the heartbeat reports `fipsEnabled=true`. This is the check to repeat continuously in production; see Reading the heartbeat.

4.2 Reading the heartbeat

The heartbeat log, emitted at INFO, carries the following three FIPS fields while FIPS 140-3 mode is enabled:

```
fipsEnabled=true fipsEnforced=false fipsModuleVersion=v1.0.0
```

`fipsEnabled` reports that FIPS 140-3 mode is enabled, the default for a FIPS build. It is the continuous evidence for condition (b) in production.

`fipsEnforced` reports strict enforcement specifically: it is `true` only under `GODEBUG=fips140=only`. A production deployment shows `fipsEnforced=false`.

> **Note**
>
> Keep `health.heartbeat.logLevel` at its default (`info`) for production deployments to ensure continuous evidence of FIPS state is captured.

4.3 Setting GODEBUG explicitly

> **Warning**
>
> `GODEBUG=fips140=off` disables FIPS 140-3 mode. No error, warning, or refusal to start follows, and no heartbeat FIPS fields are emitted. Alert on the absence of `fipsEnabled` in the heartbeat, not on a `false` value.

In production, do not set `GODEBUG=fips140=only`.

As per Go's own documentation, setting `fips140=only` states the option "introduces crashes and potentially unhandled errors by design" and "is not intended to be used in production."

Set the environment variable `GODEBUG=fips140=only` explicitly when testing, debugging, or running an assessment. This makes the module itself refuse a non-approved primitive that no Orchestrator check caught first.

Part II: Configuration required for compliance

Chapter 5

Transport layer security

Approved mode requires that connections carrying credentials or sensitive data use TLS. This is Rubrik policy rather than FIPS mandate. Only TLS 1.2 and 1.3 are allowed.

Configure the Orchestrator's inbound listener to use TLS. If the listener has no `tls:` profile, the Orchestrator logs an error naming the listener address but continues to run, serving plaintext.

5.1 Cipher suites

Not setting `enabledCiphers` is recommended: approved mode already restricts the Orchestrator to the six permitted Authenticated Encryption with Associated Data (AEAD) cipher suites. If you must restrict the list further, `enabledCiphers` selects among the four available to TLS 1.2:

- `TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256`
- `TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384`
- `TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256`
- `TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384`

TLS 1.3 negotiates its own two, which are not configurable:

- `TLS_AES_128_GCM_SHA256`
- `TLS_AES_256_GCM_SHA384`

Naming any other suite in `enabledCiphers` is refused at configuration load.

5.2 Key exchange groups

The offered groups are not configurable; the validated module applies its own policy. In approved mode they are, in order:

```
X25519MLKEM768, SecP256r1MLKEM768, SecP384r1MLKEM1024,
CurveP256, CurveP384, CurveP521
```

A TLS 1.3 handshake normally negotiates **`X25519MLKEM768`**, a post-quantum hybrid. The X25519 component is approved *as part of the hybrid construction*; plain X25519 on its own is refused by the module.

What this means in practice:

Peer offers	Result
X25519 only	Handshake fails. No approved way to complete it.
X25519 preferred, P-256 also offered	Succeeds, negotiating P-256.
TLS 1.2	Negotiates P-256. The hybrids are TLS 1.3 only.

5.3 Certificate verification

`insecureSkipVerify: true` in a `tls:` profile is refused at configuration load and stops Orchestrator startup.

No configuration path in a FIPS build disables certificate chain verification.

> **Warning**
>
> This guarantee covers configuration, not extension code. A service extension can construct its own TLS client with verification disabled, because extensions have unrestricted access to the standard library's TLS and HTTP packages. See Service extensions. Extension source review is part of your compliance boundary.

5.4 Certificate and key requirements

Key type	Requirement
RSA	2048 bits minimum
ECDSA	P-256, P-384, or P-521
Ed25519	Permitted

P-224 is also approved by the module's security policy, but excluded above because the TLS layer will not carry a P-224 certificate.

The Orchestrator enforces these requirements at configuration load: server certificates on a `tls:` profile, secret-provider client certificates, and signing keys are all checked. A failure stops Orchestrator startup. For a server certificate, the error names the `tls:` profile and both file paths.

> **Note**
>
> A handshake-time error naming:
>
> ```
> tls: no FIPS compatible certificate chains found
> ```
>
> is not about this certificate. Your own server certificate key is already validated above, before any handshake happens. If this error surfaces during a handshake instead, it names the peer's certificate.

> **Note**
>
> A misspelled `clientAuth` value is not refused; it falls through to `NoClientCert` and the listener stops asking for a client certificate. Verify mutual TLS is in effect by testing a connection without one.

5.5 Certificate revocation

A `tls:` profile with no `crl:` or `ocsp:` block performs no revocation checking, and a deployment that has not enabled it is unaffected by everything below.

When revocation checking is enabled, approved mode imposes requirements on the public key infrastructure behind the certificates that profile checks, not on the Orchestrator's configuration. None of these is checked at startup; each fails the first handshake that encounters it. All four must hold:

1. Certificate revocation lists (CRLs) signed with SHA-256 or stronger.
2. Online Certificate Status Protocol (OCSP) responses signed with SHA-256 or stronger.
3. Delegated OCSP responder certificates issued with a SHA-256-or-stronger CA signature. This is the certificate authority's choice, not the responder's. The CA signature on that certificate can be SHA-1 even when the responses themselves are not.
4. An OCSP responder that accepts SHA-256 CertIDs. The Orchestrator requests with SHA-256 in approved mode.

Chapter 6

OIDC (OpenID Connect), OAuth, and JWTs (JSON Web Tokens)

6.1 Accepted signature algorithms

In approved mode the Orchestrator verifies inbound JWTs signed with these algorithms and no others:

Family	Algorithms
RSA PKCS#1 v1.5	`RS256`, `RS384`, `RS512`
RSA-PSS	`PS256`, `PS384`, `PS512`
ECDSA	`ES256`, `ES384`, `ES512`
EdDSA	`EdDSA` (Ed25519)

A token signed with any other algorithm is refused before any key is consulted. The affected request fails without disrupting other requests.

> **Warning**
>
> The breaking case is HMAC (`HS256` / `HS384` / `HS512`). An identity provider signing ID tokens with a symmetric key (commonly derived from the client secret) stops working entirely. Every login through that connector fails.
>
> It produces no startup error. The deployment starts cleanly and fails at first login. Before switching, check `id_token_signing_alg_values_supported` in your provider's discovery document.

A JWT carrying multiple signatures is refused if any of them uses an unapproved algorithm.

6.2 Signing

Every JSON Web Signature (JWS) the Orchestrator produces (ID tokens, access tokens, client assertions) is signed with **RS256**, in both standard and FIPS builds. Published discovery metadata advertises exactly `["RS256"]`.

6.3 Key requirements

6.3.1 Keys the Orchestrator signs with

Key	Requirement	Checked at
`tokenSigning[].privateKey`	RSA, at least 2048 bits	configuration load
`oauthClientAssertion.privateKey`	RSA, at least 2048 bits	configuration load

An under-strength key stops Orchestrator startup.

6.3.2 Keys the Orchestrator verifies clients against

A client authenticating with `private_key_jwt` must register a key of at least 2048 bits.

Key	Requirement	Checked at
A client's `jwtPublicKeys` entry	RSA of at least 2048 bits	configuration load
A client's hosted JSON Web Key Set (JWKS) in its client metadata document (CIMD)	RSA of at least 2048 bits	each fetch

An under-strength key configured directly stops Orchestrator startup.

An under-strength key published in a CIMD is skipped at fetch time. A client that published one usable key alongside a weak one keeps working on the usable one. A client whose keys are *all* under-strength cannot authenticate, and the failure names the cause rather than reporting an empty key set.

6.3.3 Keys the Orchestrator verifies the identity provider against

An identity provider's published keys must be RSA of at least 2048 bits. They are not checked at startup. An under-strength key fails verification at request time, and every verification failure looks identical, so a provider with a weak key looks exactly like one sending a forged token. If verification is failing and you have ruled out the algorithm, check the provider's key size.

6.4 Certificate chains in a retrieved JWKS

No change is required at your identity provider. In approved mode the Orchestrator strips the certificate chain (`x5c`) and thumbprint fields from a fetched key set before parsing it, because SHA-1 hashing any chain it encounters is not permitted in approved mode. Nothing is lost: keys are matched by `kid` and signatures verified from the key's public parameters, so the certificate is never used.

6.5 Tokens carrying an embedded key with a certificate chain (DPoP)

A JSON Object Signing and Encryption (JOSE) header that embeds a `jwk` containing an `x5c` chain is refused:

```
failed to parse token: embedded jwk carries an x5c certificate chain
```

Unlike a fetched key set, header content is signed material and cannot be rewritten without invalidating the signature, so refusal is the only safe option.

The practical case: Demonstrating Proof of Possession (DPoP). The refusal applies to every JWT the Orchestrator parses. DPoP is the only protocol where an embedded `jwk` header is normal, required content (see RFC 9449), so it is the only context where a client would carry one without already deviating from spec. A DPoP client that also includes a certificate chain in that header is refused. The client sees:

```
{
  "error": "invalid_dpop_proof",
  "error_description": "DPoP proof is not a valid JWT"
}
```

DPoP proofs accept the same algorithms as any other signed token.

Chapter 7

SAML (Security Assertion Markup Language)

7.1 Configuration validated at startup

Setting	Permitted in approved mode
`keyEncryptMethod`	`RSA_OAEP` only (`RSA-1_5` is refused)
`dataEncryptMethod`	`AES128CBC`, `AES192CBC`, `AES256CBC`
`digestMethod`	`SHA256`, `SHA512`
encryption `certificate`	RSA, at least 2048 bits
signing `privateKey`	RSA, at least 2048 bits
`requestVerification certificate`	RSA, at least 2048 bits
`samlSPKeyPath` (SAML connector)	RSA, at least 2048 bits

`RSA-1_5` (PKCS#1 v1.5) is not an approved key transport scheme under NIST SP 800-56B Rev 2, and the validated module refuses it. The permitted list is an allow-list, so an unrecognized value is also refused.

A setting outside these values stops Orchestrator startup, naming the application or connector at fault.

7.2 Outbound signing

Everything the Orchestrator signs is already approved:

Element	Value
Signature method	`http://www.w3.org/2001/04/xmldsig-more#rsa-sha256`
Digest method	`http://www.w3.org/2001/04/xmlenc#sha256`
Canonicalization	`http://www.w3.org/2001/10/xml-exc-c14n#`

This applies whether the Orchestrator acts as a service provider (signing `AuthnRequest` and logout messages) or as an identity provider (signing responses and assertions). No configuration field changes any of it.

Your SAML peer must be able to accept RSA-SHA256 signatures with SHA-256 digests and exclusive canonicalization. A peer that requires `rsa-sha1` will not work.

> ⓘ Note
>
> The Orchestrator's identity provider metadata document is not signed. It does advertise the algorithms above in its `<md:Extensions>` element, so a service provider that honors algorithm-support metadata will select approved algorithms on its own. If your service provider requires signed metadata, reach out to Rubrik to discuss options.

7.3 Inbound signatures

A message signed with `rsa-sha1` is refused cleanly, in both roles. The algorithm named on an inbound message is read off the wire and checked against the approved set before the message reaches the verifier, so an unapproved algorithm is rejected before verification runs. Each combination below is refused in approved mode:

Orchestrator role	SAML message	SHA-1 in
Identity provider	`AuthnRequest`, `LogoutRequest` (POST)	signature method
Identity provider	`AuthnRequest`, `LogoutRequest` (Redirect)	`SigAlg` parameter
Identity provider	any	digest method
Service provider	`Response` / `Assertion` (POST)	signature or digest
Service provider	`LogoutResponse` (POST)	signature or digest

> ⓘ Note
>
> The redirect-binding logout response is a separate case. When the Orchestrator validates a `LogoutResponse` over HTTP-Redirect, it ignores the `SigAlg` parameter entirely and always verifies with SHA-256. An identity provider signing that message with SHA-1 fails with an invalid-signature error rather than one naming the algorithm, so the symptom looks different from the rows above.

> **Warning**
>
> This error names the wrong setting. If you use encrypted assertions, you may see:
>
> ```
> failed to retrieve SAML assertion: decrypting the encrypted assertion's key
> failed, and the certificate-fingerprint diagnostic is unavailable under
> FIPS 140-3 approved mode: crypto/sha1: use of SHA-1 is not allowed in
> FIPS 140-only mode
> ```
>
> The cause is a SHA-1 digest on the peer's key transport: the `DigestMethod` inside its `EncryptedKey`, which ADFS and Okta emit routinely. SHA-1 is not permitted there in approved mode.
>
> The message's advice points at the peer's signature algorithm ("the signing party must use RSA or ECDSA with SHA-256 or stronger"), which is a different setting and may already be correct. Change the peer's encryption digest to SHA-256 or stronger.

A SHA-1 signature or digest also logs a deprecation notice at INFO before it fails, naming which of the two is at fault:

```
deprecated sha1 hash used for signature method
deprecated sha1 hash used for digest method
```

7.4 What your SAML peers must do

The Orchestrator does not check peer configuration at startup. A peer that does not meet these requirements causes the first request it handles to fail; the Orchestrator continues running. Confirm peer settings before switching to an approved-mode deployment.

Required for a working approved-mode deployment:

1. Sign with RSA-SHA256 or stronger: `rsa-sha256`, `rsa-sha384`, `rsa-sha512`, or the ECDSA SHA-2 equivalents. Not `rsa-sha1`.
2. Use SHA-256 or stronger reference digests. The signature method and the digest method are separate settings in most products. A peer configured for "SHA-256 signatures" may still be emitting SHA-1 digests, and that fails too.
3. Apply these signature and digest requirements to every message and both bindings: `Response`, `Assertion`, `AuthnRequest`, `LogoutRequest`, `LogoutResponse`, over HTTP-POST and HTTP-Redirect. On the redirect binding, the `SigAlg` query parameter must carry an approved URI.
4. If encrypting assertions, use RSA-OAEP key transport, AES-CBC content encryption, and a SHA-2 digest. RSA-1_5 key transport is not permitted.
5. Use signing keys of RSA at least 2048 bits, or an approved ECDSA curve (P-256, P-384, or P-521).
6. No change to canonicalization is required. Inclusive or exclusive canonicalization is accepted; it involves no hashing.

Chapter 8

Directory services

This chapter covers LDAP (Lightweight Directory Access Protocol): the Orchestrator serving as an LDAP Provider, LDAP connectors, and LDAP attribute providers.

8.1 Transport (LDAP)

All LDAP connections must use `ldaps`. This applies to the inbound connections to the Orchestrator listener for the LDAP Provider and the outbound connections the Orchestrator makes to your directory.

`ldap` with StartTLS is not supported because a server cannot compel a client to upgrade before binding. TLS is enforced at the connection level, not left to the client to initiate.

These requirements are not checked at startup. For inbound connections, a client that does not meet them is refused at connect time. For outbound connections to your directory, the Orchestrator starts cleanly and fails at the first connection attempt.

8.2 Authentication mechanisms

Simple bind over TLS is the only supported mechanism. SASL, GSS-SPNEGO, and NTLM are not supported.

A client attempting a SASL bind receives `LDAPResultAuthMethodNotSupported`. A simple bind attempted over an unencrypted connection is refused with `confidentialityRequired`. Both are refused at request time and do not block startup.

8.3 What your directory must support

Requirement	Detail
LDAPS	TLS 1.2 or higher, approved cipher suite, supported key exchange group
Certificate	RSA of at least 2048 bits, or P-256/384/521
Simple bind	The only mechanism the Orchestrator will use

An Active Directory deployment that has only ever been reached over `ldap://` with NTLM or Kerberos bind requires both changes: an LDAPS endpoint, and simple bind credentials.

Chapter 9

Secret providers

9.1 Client certificates

A secret provider's client certificate key must be RSA of at least 2048 bits, or ECDSA on P-256, P-384, or P-521. Ed25519 is also permitted. The Orchestrator checks this at configuration load wherever a secret provider is reached using mutual TLS, and a key that fails stops Orchestrator startup.

Every secret provider connection uses the Orchestrator's outbound TLS defaults: TLS 1.2 minimum, the six approved cipher suites, and full certificate chain verification.

Chapter 10

Authorization policy (Rego)

This chapter applies to authorization policy written in Rego within the AI Identity Gateway's per-tool authorization and token-minting policies.

Policies written in Rego are compiled when the Orchestrator loads its configuration. In approved mode four cryptographic built-ins are withheld:

```
crypto.md5
crypto.sha1
crypto.hmac.md5
crypto.hmac.sha1
```

Everything else remains available, including:

- `crypto.sha256`, `crypto.hmac.sha256`, `crypto.hmac.sha512`
- `crypto.hmac.equal`, and the `crypto.x509.*` family
- `io.jwt.*`, `rand.*`, and `http.send`

A policy calling a withheld built-in means the Orchestrator will error at startup until the policy is corrected.

10.1 Writing policies for approved mode

Keeping a policy FIPS compliant is your responsibility.

A policy's own keys, secrets, certificates, and algorithm choices are not inspected. A non-approved choice runs and returns a result with no error and no log line. Review policy source against the same algorithm and key requirements this guide sets everywhere else.

Chapter 11

Redis cache encryption

Cached data is encrypted at rest with an approved algorithm: **AES-256-GCM** with a random nonce generated by the validated module. In approved mode a Redis cache must also have a `tls:` profile; a cache configured without one stops startup.

11.1 Key rotation

SP 800-38D caps random-nonce GCM at 2^32 messages per key before collision risk stops being negligible.

> **Warning**
>
> Crossing that limit is silent: the result is no longer FIPS approved, but nothing errors or logs to indicate it.

On high-volume deployments, rotate before you would plausibly reach it:

1. Set a new `encryption.keys.current`.
2. Move the previous value into `encryption.keys.old`, kept only for decrypting data already in the cache.

Part III: Service Extension compliance

Chapter 12

Service extensions

Service extensions are Go code you supply, loaded into the Orchestrator's own process through an embedded Go runtime with access to nearly the entire Go standard library.

> ⓘ Note
>
> Any cryptography an extension performs runs alongside the cryptography the deployment's compliance claim rests on, but the Orchestrator's own compliance does not extend to it.

The Orchestrator withholds the non-approved cryptographic packages it can, but cannot see inside a function body. Extension compliance gaps covers what import restrictions cannot catch.

Review extension source before deploying to a FIPS Orchestrator. The Orchestrator enforces which packages an extension may import, not which algorithms it implements or how it opens network connections. Two consequences follow:

- An extension can contain hand-written cryptography that runs undetected.
- An extension can construct its own TLS client with certificate verification disabled, bypassing both the configuration-level prohibition and the Orchestrator's outbound connection defaults.

12.1 Packages withheld in approved mode

Importing any of these fails at configuration load, with an error naming the package and the alternative.

Package	Why	Use instead
`crypto/md5`	MD5 is not an approved hash	`crypto/sha256` or `crypto/sha512`
`crypto/sha1`	SHA-1 is not an approved hash	`crypto/sha256` or `crypto/sha512`
`crypto/des`	DES and 3DES are not approved ciphers	`crypto/aes` with an approved cipher mode

Package	Why	Use instead
`crypto/rc4`	RC4 is not an approved cipher	`crypto/aes` with an approved cipher mode
`crypto/dsa`	DSA is not an approved signature algorithm	`crypto/ecdsa` with P-256, P-384, or P-521
`secp256k1/v4` (see below)	secp256k1 is not a NIST-approved curve	`crypto/ecdsa` with P-256, P-384, or P-521
`github.com/go-ldap/ldap/v3`	Carries non-approved bind mechanisms, and a connection value cannot be partially restricted	`api.AttributeProvider` / `api.IdentityProvider`; see Directory lookup and authentication
`github.com/go-jose/go-jose/v3`	Algorithms are named by arbitrary string, so no approved subset can be enforced	No in-extension alternative; see Signatures and tokens
`github.com/go-jose/go-jose/v3/jwt`	Same	No in-extension alternative; see Signatures and tokens

The secp256k1 row above is the import path `github.com/decred/dcrd/dcrec/secp256k1/v4`, shortened to fit the table.

> **Warning**
>
> go-jose is withheld and has no in-extension substitute. An extension that uses it to sign, encrypt, verify, or decrypt JWTs cannot run on a FIPS Orchestrator. See Signatures and tokens for what to do instead.

12.2 Symbols withheld from packages that remain

These packages stay importable; only the listed symbols are gone, and referencing one fails at configuration load with `has no symbol`.

Package	Withheld	Why	Use instead
`crypto`	`MD4`, `MD5`, `MD5SHA1`, `RIPEMD160`, `SHA1`	Non-approved hash identifiers	`crypto.SHA256`, `crypto.SHA384`, `crypto.SHA512`

Package	Withheld	Why	Use instead
`crypto`	`RegisterHash`	Rebinds a hash implementation for the entire process, including code outside your extension	None
`crypto/cipher`	`NewCFBEncrypter`, `NewCFBDecrypter`, `NewOFB`	CFB and OFB are not approved cipher modes	`NewCBCEncrypter` ↪ or `NewCTR`
`crypto/cipher`	`NewGCM`, `NewGCMWithNonceSize`, `NewGCMWithTagSize`	These take the nonce from the caller, and an AES-GCM nonce must be constructed as SP 800-38D requires	`NewCBCEncrypter` ↪ or `NewCTR`; see AES-GCM
`github.com/google/uuid`	`NewMD5`, `NewSHA1`	Version 3 and version 5 UUIDs are defined in terms of MD5 and SHA-1	`uuid.New()` (version 4, random)

12.2.1 The `crypto` package still fails on these identifiers

Importing `crypto` is allowed (it also carries `crypto.Signer` and the approved hash identifiers you need), but referencing a withheld identifier still fails:

```
import "crypto"

func Example() {
    _ = crypto.MD5
}
```

Same shape for `MD4`, `MD5SHA1`, `RIPEMD160`, `SHA1`, and `RegisterHash`; only the identifier name changes.

> **Warning**
>
> This does not fully close the route. See Do not build a `crypto.Hash` by arithmetic or conversion.

12.3 Approved alternatives

12.3.1 Directory lookup and authentication

Withheld:

```
conn, err := ldap3.DialURL(ldapURL, ldap3.DialWithTLSConfig(tlsCfg))
if err != nil {
    return err
}
defer conn.Close()

if err := conn.Bind(bindDN, password); err != nil {
    return err
}
```

Attribute lookup: use an attribute provider configured in the Identity Fabric.

```
ap, err := api.AttributeProvider("corp-ldap")
if err != nil {
    return err
}

attrs, err := ap.Query(subject, []string{"mail", "memberOf"})
if err != nil {
    return err
}
```

Credential verification: use an identity provider; the Orchestrator performs the bind over enforced `ldaps://`.

```
idp, err := api.IdentityProvider("corp-ldap")
if err != nil {
    return err
}

idp.Login(rw, req)
```

12.3.2 Hashing

Withheld:

```
sum := md5.Sum(data)     // or sha1.Sum(data)
```

Use instead:

```
sum := sha256.Sum256(data)
```

`crypto/sha256` and `crypto/sha512` are approved and available. If you are hashing only to produce a short, stable identifier (a cache key or a correlation value), SHA-256 truncated to the length you need is the approved way to do it.

12.3.3 UUIDs

Withheld:

```
id := uuid.NewMD5(uuid.NameSpaceURL, []byte(name))   // or uuid.NewSHA1(...)
```

Use instead:

```
id := uuid.New()
```

> ⓘ Note
>
> Version 3 and version 5 UUIDs have no approved equivalent. The RFC defines them in terms of MD5 and SHA-1 respectively, so a name-based UUID cannot be produced with an approved hash and remain the same UUID. Store a deterministic identifier in a field of your own, not a UUID-typed one.

The rest of the `uuid` package is unaffected, including `uuid.New`, `uuid.NewString`, `uuid.Parse`, and the version 6 and 7 constructors.

12.3.4 Symmetric encryption

Withheld:

```
block, _ := des.NewCipher(key)
stream := cipher.NewCFBEncrypter(block, iv)
```

Use instead:

```
block, err := aes.NewCipher(key)
if err != nil {
    return err
}

mode := cipher.NewCBCEncrypter(block, iv)   // or cipher.NewCTR(block, iv)
```

12.3.4.1 AES-GCM

Withheld:

```
nonce := make([]byte, 12)
aead, _ := cipher.NewGCM(block)   // or NewGCMWithNonceSize,
    ↪ NewGCMWithTagSize
ct := aead.Seal(nil, nonce, plaintext, nil)
```

Use instead:

```
mode := cipher.NewCBCEncrypter(block, iv)   // with a separate HMAC
```

> ⓘ Note
>
> All three withheld constructors take the nonce from the caller, and a GCM nonce must be constructed as SP 800-38D requires, which nothing can verify from outside your code. AES-GCM is therefore unavailable to an extension in approved mode; use AES-CBC or AES-CTR with a separate HMAC for authentication.

12.3.5 Signatures and tokens

Withheld: `crypto/dsa`, secp256k1, and all of go-jose.

For raw signatures, ECDSA on an approved curve:

```
key, err := ecdsa.GenerateKey(elliptic.P256(), rand.Reader)
if err != nil {
    return err
}

digest := sha256.Sum256(payload)
sig, err := ecdsa.SignASN1(rand.Reader, key, digest[:])
```

RSA with an approved hash also works, as does Ed25519; `crypto/ed25519` is inside the validated module and needs no special handling.

JWTs have no in-extension option. Restructure so the Orchestrator owns the token:

You were doing	Do this instead
Minting a JWT in an extension	Configure an OIDC app and shape its claims with `buildAccessTokenClaimsSE` / `buildIDTokenClaimsSE`
Verifying an inbound JWT in an extension	Let the OIDC app or the app policy validate it; read the resulting session attributes
Signing an assertion for an upstream	Use the SAML or OIDC app configuration rather than extension code

12.4 Extension compliance gaps

The Orchestrator enforces **which packages an extension may import, not which algorithms it may implement**. Everything under Packages withheld and Symbols withheld is enforced. The three things below are not, and cannot be by any check the Orchestrator can perform on extension source. They are your responsibility, and none of them produces an error, a panic, or a log line: each returns a real result that is not FIPS approved. Extension code must be reviewed before it is deployed to a FIPS Orchestrator.

12.4.1 Do not implement cryptography yourself

Code you write by hand inside an extension never enters the validated module, so nothing refuses it. An extension containing its own digest, cipher, or curve implementation runs perfectly and is not FIPS compliant. This includes code pasted from elsewhere or vendored into your extension file.

12.4.2 Do not build a `crypto.Hash` by arithmetic or conversion

`crypto.Hash` is an integer type, so a withheld identifier can be reconstructed from an approved one. Withholding the constants cannot prevent arithmetic on a value your extension is entitled to hold:

```
// Both of these name MD4, without the identifier appearing in your source.
byConversion := crypto.Hash(1).New()
byArithmetic := (crypto.SHA256 - 4).New()
```

None of MD4, RIPEMD160, MD5, or SHA-1 are FIPS-approved. Reaching one this way produces a working digest, with no error or log line, and makes the deployment non-compliant. Reference approved hashes by name only (`crypto.SHA256`, `crypto.SHA384`, `crypto.SHA512`), and never compute a `crypto.Hash` from an integer, a config value, or a request parameter.

12.4.3 Do not pass non-approved parameters to approved calls

These calls are all legitimate approved operations, so none can be withheld without removing capability you need. What makes a call non-approved is the *parameter*, which the Orchestrator cannot inspect at configuration load:

Call	Not approved when
`hmac.New(sha256.New, key)`	`key` is shorter than 14 bytes (112 bits)
`x509.CreateCertificate(...)`	the signature algorithm is `SHA1WithRSA`, `ECDSAWithSHA1`, or any other SHA-1 variant
`cipher.NewCBCEncrypter` / `cipher.NewCTR`	the block cipher is not AES
`ecdsa.GenerateKey(...)`	the curve is not P-256, P-384, or P-521

Every one of them succeeds. A P-224 key generates exactly as a P-256 key does, and a 4-byte HMAC key returns a working MAC: no error, no panic, no log line, and a result that is not FIPS approved. Check the parameter yourself:

```
// HMAC keys must be at least 14 bytes in approved mode.
if len(key) < 14 {
    return errors.New("hmac key too short")
}
mac := hmac.New(sha256.New, key)
```

12.5 What remains available

Service Extension SDK: unchanged in approved mode. This includes `orchestrator`, `session`, `cache`, `secret`, `log`, `router`, `idfabric`, `tai`, and `weblogic`.

> ⓘ Note
>
> `service-extension/ldap` is not the withheld package. The SDK's own `ldap` package (which LDAP provider extensions use to read the bind DN off the request context) stays available. What is withheld is the third-party `github.com/go-ldap/ldap/v3` client library. The two are easy to confuse because both are imported as `ldap`.

Approved cryptography from the standard library:

Purpose	Use
Hashing	`crypto/sha256`, `crypto/sha512`
Message authentication	`crypto/hmac` (keys of at least 14 bytes)
Symmetric encryption	`crypto/aes` with CBC or CTR (not GCM)
Signatures	`crypto/rsa`, `crypto/ecdsa` (P-256/384/521), `crypto/ed25519`
Key agreement	`crypto/ecdh`
Randomness	`crypto/rand`
Transport	`crypto/tls` at TLS 1.2 or higher
Certificates	`crypto/x509`, `encoding/asn1`

Third-party libraries: `github.com/google/uuid` (minus the two constructors above), the AWS SDK (`config`, `verifiedpermissions`, and `verifiedpermissions/types`), `github.com/go-asn1-ber/asn1-ber`, and `golang.org/x/net/html`.

The rest of the standard library, including `net/http`, `encoding/json`, `strings`, and `time`, is untouched.

Appendices

Appendix A: Approved algorithm reference

What the Orchestrator uses and accepts in approved mode.

Cryptography the Orchestrator performs

Purpose	Algorithm
JWS signing (all tokens)	RS256
SAML signing	RSA-SHA256, SHA-256 digest, exclusive canonicalization
SAML key encryption (`keyEncryptMethod`)	RSA-OAEP
SAML content encryption	AES-128/192/256-CBC
Cache encryption at rest	AES-256-GCM, random nonce
Session identifiers	Approved random source
OCSP request CertID	SHA-256
TLS	See Transport layer security

Cryptography the Orchestrator accepts

Input	Accepted
Inbound JWT signature	RS256/384/512, PS256/384/512, ES256/384/512, EdDSA (Ed25519)
DPoP proof signature	Same as an inbound JWT: RS256/384/512, PS256/384/512, ES256/384/512, EdDSA (Ed25519)
Inbound SAML signature	RSA-SHA256 or stronger, both roles (see the SAML chapter)
TLS versions	1.2, 1.3
TLS cipher suites	Six AEAD suites (see Transport layer security)
TLS key exchange	X25519MLKEM768, SecP256r1MLKEM768, SecP384r1MLKEM1024, P-256, P-384, P-521 (see Key exchange groups)
RSA keys	At least 2048 bits

Input	Accepted
ECDSA curves	P-256, P-384, P-521
CRL / OCSP signatures	SHA-256 or stronger
HMAC keys	At least 14 bytes (112 bits)

Appendix B: References

- CMVP certificate #5247
- Module security policy (dated June 12, 2026), with a pinned copy in the release bundle as `GeomysCMVPSecurityPolicy5247.pdf`
- FIPS 140-3 standard
- FIPS 140-3 Implementation Guidance
- SP 800-131A Rev 2: key length transitions
- SP 800-38D: GCM and GMAC
- SP 800-56B Rev 2: key establishment
- RFC 9449: DPoP
- Go: FIPS 140-3 compliance
- Orchestrator documentation
- Orchestrator installation
- Service Extensions reference