

# RIFT Import Enforcement Guide

## OBINexus Computing - AEGIS Project Implementation

**Version:** 1.0.0-dev

**Stage:** Implementation Gate

**Classification:** Git-RAF Enforced

**Author:** AEGIS Core Engineering Team

**Date:** June 22, 2025

### Executive Summary

The RIFT Import Enforcement system implements cryptographically verified, slot-based module resolution with mandatory semantic versioning constraints. This specification ensures all module imports undergo governance validation while preventing URI injection vulnerabilities and maintaining deterministic dependency resolution across all compilation stages.

### Semantic Versioning Architecture (smever.x)

#### Bracketed Syntax Requirements

All RIFT imports must utilize bracketed semantic versioning syntax to prevent URI injection attacks and ensure cryptographic verification of module dependencies.

#### Canonical Import Format

rift

```
// ✅ CORRECT: Bracketed semantic versioning with slot resolution
from rift_std.core.sync[stable@1.4.2] import RiftLock
from rift_std.crypto.hash[beta@2.1.0-rc1] import Blake3Context
from rift_std.governance.policy[alpha@0.9.5-dev] import PolicyValidator
```

```
// ❌ FORBIDDEN: Unsafe @ symbol usage (URI injection vulnerability)
from rift_std.core.sync@stable-1.4.2 import RiftLock
from rift_std.crypto.hash@beta/2.1.0 import Blake3Context
```

#### Version Constraint Syntax

rift

```
// Exact version constraint
from module.path[stable@1.4.2] import Component

// Compatible version range (semantic versioning rules)
from module.path[stable@^1.4.0] import Component // >= 1.4.0, < 2.0.0
from module.path[beta@~2.1.0] import Component // >= 2.1.0, < 2.2.0

// Pre-release channel specification
from module.path[alpha@1.0.0-rc1+build.123] import Component
```

## Slot-Hash Lookup Architecture

### Cryptographic Module Resolution

Each module import resolves through a cryptographically verified slot-hash lookup system that maps semantic versions to content-addressed storage.

```
c

// Slot-hash Lookup structure
typedef struct {
    char module_namespace[128]; // Fully qualified module namespace
    char version_constraint[32]; // Semantic version constraint string
    char channel_identifier[16]; // Release channel (stable/beta/alpha)
    uint8_t content_hash[32]; // BLAKE3 hash of module content
    uint8_t signature[64]; // Ed25519 signature of module metadata
    uint64_t build_timestamp; // UTC timestamp of module build
} rift_module_slot_t;

// Module Lookup cache structure
typedef struct {
    rift_module_slot_t slots[1024]; // Pre-allocated slot array
    uint32_t slot_count; // Current number of populated slots
    uint8_t cache_integrity_hash[32]; // BLAKE3 hash of entire cache
    uint8_t cache_signature[64]; // Ed25519 signature of cache state
} rift_module_cache_t;
```

### Slot Resolution Protocol

c

```
// Resolve module import to content hash
int rift_resolve_module_import(
    const char* namespace_path,
    const char* version_constraint,
    const char* channel,
    rift_module_slot_t* resolved_slot
);

// Validate resolved module against governance policies
int rift_validate_module_governance(
    const rift_module_slot_t* slot,
    const uint8_t* policy_requirements,
    size_t policy_length
);
```

## Canonical Import Path Specifications

### Namespace Hierarchy Requirements

```
rift_std.{domain}.{subdomain}[{channel}@{version}]
├─ rift_std.core.*           # Core language runtime components
├─ rift_std.crypto.*         # Cryptographic primitives and protocols
├─ rift_std.governance.*     # Policy enforcement and validation
├─ rift_std.io.*             # Input/output and networking
├─ rift_std.concurrent.*     # Thread-safe concurrency primitives
└─ rift_std.hardware.*       # Hardware abstraction layer
```

## Channel Classification System

```
rift

// Stable channel - production-ready, governance-validated
from rift_std.core.types[stable@2.1.4] import SafeInteger

// Beta channel - feature-complete, pre-production validation
from rift_std.experimental.quantum[beta@1.0.0-beta.3] import QuantumContext

// Alpha channel - development features, governance review pending
from rift_std.research.neural[alpha@0.8.2-dev] import NetworkLayer

// Internal channel - organization-specific, restricted access
from obnx_internal.accounting.ledger[internal@1.2.1] import TransactionValidator
```

## Third-Party Module Integration

rift

```
// External module import with cryptographic verification
from external.{org}.{module}[{channel}@{version}+{build_hash}] import Component

// Example: External cryptographic library
from external.sodium.crypto[stable@1.0.18+blake3.a1b2c3d4] import SecretBox

// Example: External data processing library
from external.apache.arrow[beta@12.0.1+blake3.e5f6g7h8] import Table
```

## Version Lock Enforcement

### Deterministic Dependency Resolution

```
c

// Version lock file structure (.rift.Lock)
typedef struct {
    char lock_format_version[8];           // Lock file format version
    uint32_t dependency_count;             // Number of Locked dependencies
    uint64_t lock_generation_time;         // UTC timestamp of Lock generation
    uint8_t lock_integrity_hash[32];       // BLAKE3 hash of all dependencies
    uint8_t lock_signature[64];           // Ed25519 signature of Lock file
} rift_version_lock_header_t;

// Individual dependency Lock entry
typedef struct {
    char namespace[128];                   // Module namespace path
    char resolved_version[32];             // Exact resolved version
    char channel[16];                      // Resolved channel identifier
    uint8_t content_hash[32];              // BLAKE3 hash of module content
    uint8_t dependency_signature[64];      // Ed25519 signature of dependency
} rift_dependency_lock_t;
```

### Lock File Generation and Validation

c

```
// Generate version lock file from dependency resolution
int rift_generate_version_lock(
    const char* project_root,
    const rift_dependency_lock_t* dependencies,
    uint32_t dependency_count,
    const char* output_lock_file
);

// Validate project against existing lock file
int rift_validate_version_lock(
    const char* project_root,
    const char* lock_file_path,
    bool* lock_valid,
    char* validation_errors,
    size_t error_buffer_size
);
```

## Import Validation Pipeline

### Stage 1: Syntax Validation

c

```
// Validate import statement syntax
typedef enum {
    RIFT_IMPORT_VALID,
    RIFT_IMPORT_INVALID_NAMESPACE,
    RIFT_IMPORT_INVALID_VERSION,
    RIFT_IMPORT_UNSAFE_URI_CHARS,
    RIFT_IMPORT_MISSING_BRACKETS,
    RIFT_IMPORT_INVALID_CHANNEL
} rift_import_validation_result_t;

int rift_validate_import_syntax(
    const char* import_statement,
    rift_import_validation_result_t* validation_result,
    char* error_details,
    size_t error_buffer_size
);
```

### Stage 2: Cryptographic Verification

c

```
// Verify module cryptographic integrity
int rift_verify_module_integrity(
    const rift_module_slot_t* resolved_slot,
    const uint8_t* module_content,
    size_t content_length,
    bool* integrity_valid
);

// Verify module signature chain
int rift_verify_module_signature_chain(
    const rift_module_slot_t* slot,
    const uint8_t* trust_anchor_public_key,
    bool* signature_valid
);
```

## Stage 3: Governance Policy Enforcement

c

```
// Policy enforcement for module imports
typedef struct {
    bool allow_alpha_channel;           // Allow alpha channel imports
    bool allow_external_modules;        // Allow external organization modules
    uint32_t max_dependency_depth;      // Maximum dependency chain depth
    char* allowed_namespaces[64];      // Whitelist of allowed namespaces
    uint32_t allowed_namespace_count;  // Count of allowed namespaces
} rift_import_policy_t;

int rift_enforce_import_policy(
    const rift_module_slot_t* slot,
    const rift_import_policy_t* policy,
    bool* policy_compliant,
    char* policy_violation_details,
    size_t details_buffer_size
);
```

## Build System Integration

### Dependency Resolution Build Phase

```
makefile
```

```
# RIFT dependency resolution integration
```

```
RIFT_RESOLVER = rift-resolve-deps
```

```
RIFT_VALIDATOR = rift-validate-imports
```

```
# Dependency resolution target
```

```
%.deps.resolved: %.rift %.rift.lock
```

```
$(RIFT_RESOLVER) resolve --project=$< --lock-file=$*.rift.lock --output=$@
```

```
$(RIFT_VALIDATOR) verify-lock --project=$< --lock-file=$*.rift.lock
```

```
# Import validation target
```

```
%.imports.validated: %.deps.resolved
```

```
$(RIFT_VALIDATOR) validate-imports --deps-file=$< --policy=strict --output=$@
```

```
rift-bridge.exe validate-stage --stage=1 --input-hash=$< --output-hash=$@
```

## Continuous Integration Requirements

```
bash
```

```
# CI/CD import validation pipeline
```

```
rift-ci-validate-imports:
```

- rift-resolve-deps --project=. --lock-file=rift.lock --verify-only
- rift-validate-imports --policy=production --fail-on-alpha
- rift-bridge.exe verify-policy-chain --stage=1 --strict-mode
- git-raf verify-attestation --stage=import-resolution

## Command Line Tools

### Dependency Resolution Commands

bash

#### *# Resolve project dependencies*

```
rift-resolve-deps resolve --project=. --output=rift.lock
rift-resolve-deps update --package=rift_std.core.sync --version="^2.1.0"
rift-resolve-deps add --package=rift_std.crypto.hash[stable@1.4.2]
```

#### *# Import validation commands*

```
rift-validate-imports check --project=. --policy=strict
rift-validate-imports lint --project=. --fix-unsafe-imports
rift-validate-imports audit --project=. --output=import_audit.json
```

#### *# Lock file management*

```
rift-lock-manager verify --lock-file=rift.lock --project=.
rift-lock-manager update --lock-file=rift.lock --force-refresh
rift-lock-manager diff --old=rift.lock.backup --new=rift.lock
```

## Module Cache Management

bash

#### *# Module cache operations*

```
rift-cache status --show-integrity
rift-cache clean --expired-only
rift-cache rebuild --verify-signatures
rift-cache export --output=module_cache_backup.tar.gz
```

#### *# Module integrity operations*

```
rift-integrity verify-module --namespace=rift_std.core.sync --version=1.4.2
rift-integrity check-signatures --cache-path=/var/cache/rift/modules
rift-integrity repair-cache --backup-path=module_cache_backup.tar.gz
```

## Security Considerations

### URI Injection Prevention



c

```
// Validate namespace characters against URI injection
#define RIFT_SAFE_NAMESPACE_CHARS "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ01234567

int rift_validate_namespace_safety(
    const char* namespace,
    bool* is_safe,
    char* unsafe_characters,
    size_t unsafe_buffer_size
);

// Validate version string against injection attacks
int rift_validate_version_safety(
    const char* version_string,
    bool* is_safe,
    char* safety_issues,
    size_t issues_buffer_size
);
```



## Cryptographic Requirements

- **Content Hashing:** BLAKE3 for all module content verification
- **Signature Verification:** Ed25519 for module authenticity
- **Cache Integrity:** ChaCha20-Poly1305 for cache storage protection
- **Lock File Protection:** Ed25519 signatures for lock file integrity

## Error Handling and Diagnostics

### Import Resolution Errors

c

```
#define RIFT_IMPORT_ERR_NAMESPACE_NOT_FOUND -2001
#define RIFT_IMPORT_ERR_VERSION_CONSTRAINT -2002
#define RIFT_IMPORT_ERR_SIGNATURE_INVALID -2003
#define RIFT_IMPORT_ERR_CONTENT_CORRUPTED -2004
#define RIFT_IMPORT_ERR_POLICY_VIOLATION -2005
#define RIFT_IMPORT_ERR_CHANNEL_RESTRICTED -2006
#define RIFT_IMPORT_ERR_DEPENDENCY_CYCLE -2007
```

### Diagnostic Information Generation

c

```
// Generate comprehensive import diagnostic report
int rift_generate_import_diagnostics(
    const char* project_root,
    const char* output_file,
    bool include_dependency_graph,
    bool include_signature_details
);

// Validate entire project import graph
int rift_validate_project_import_graph(
    const char* project_root,
    bool* graph_valid,
    char* validation_report,
    size_t report_buffer_size
);
```

## Testing and Quality Assurance

### Unit Test Coverage Requirements

```
bash

# Import validation test suite
test_bracketed_syntax_validation
test_slot_hash_resolution
test_version_constraint_parsing
test_cryptographic_verification
test_policy_enforcement
test_lock_file_generation
test_dependency_graph_validation
```

### Integration Test Requirements


```
bash

# End-to-end import system testing
test_full_dependency_resolution_pipeline
test_cross_stage_import_validation
test_git_raf_integration_with_imports
test_module_cache_integrity
test_emergency_import_recovery
```

### Performance Benchmarks

- **Import Resolution Latency:** < 50ms per import statement

- **Dependency Graph Resolution:** < 500ms for 1000 dependencies
- **Cryptographic Verification:** < 100ms per module signature
- **Cache Lookup Performance:** < 10ms per cache query

**Implementation Status:**  Import Enforcement Architecture Documented

**Next Target:** Gossip Language Governance Layer (.gs[n] modules)

**AEGIS Gate Progress:** Implementation Gate - Component Development Phase