nullsec Tool Protocol

The typed inter-tool communication protocol for nullsec pipeline tools

Lateralus Language

bad-antics · April 2026 · nullsec / Lateralus Language Research

ABSTRACT nullsec tools communicate via a typed protocol that enables pipeline composition without text parsing. The protocol defines: a binary-encoded message format (CBOR-based), a typed schema registry for common security data types (hosts, ports, vulnerabilities, credentials), and a streaming mode for large datasets. This paper specifies the protocol, the schema registry, and the rules for adding new schemas. It also describes how the protocol integrates with the Lateralus pipeline type system to provide compile-time validation of tool-to-tool connections.

1. Why a Typed Protocol

Traditional Unix philosophy: tools communicate via plain text (grep, awk, sed form the glue). This works for simple cases but fails when tool output is structured: nmap XML, JSON APIs, binary captures. Each tool must implement its own parser for upstream tool output, creating a fan-out of brittle parsers.

The nullsec protocol provides a typed binary message format. A tool that produces a HostScanResult need not serialize to text; the next tool receives the structured value directly. No parsers, no fragile grep patterns, no lost precision.

2. Message Format: CBOR with Type Tags

Messages are encoded in CBOR (Concise Binary Object Representation, RFC 8949) with nullsec-specific type tags. Each message starts with a nullsec header:

```
// Message envelope
struct Message {
    magic:      [u8; 4],  // 0x4E534543 ('NSEC')
    version:    u16,
    schema_id:  u32,      // identifies the payload type
    payload_sz: u32,
    payload:    CBOR,     // schema-typed CBOR value
    checksum:   u32,      // CRC32 of header + payload
}
```

The schema_id maps to an entry in the schema registry. The receiving tool looks up the schema, validates the payload against it, and deserializes into the corresponding Lateralus type.

3. Schema Registry

The schema registry is a TOML file listing all known schema IDs and their Lateralus type names:

```
# nullsec/schemas/registry.toml
[schemas]
1   = "nullsec::schema::HostScanResult"
2   = "nullsec::schema::PortResult"
3   = "nullsec::schema::VulnReport"
4   = "nullsec::schema::Credential"
5   = "nullsec::schema::DnsRecord"
6   = "nullsec::schema::HttpRequest"
7   = "nullsec::schema::HttpResponse"
8   = "nullsec::schema::CertificateInfo"
```

Adding a new schema requires: defining the Lateralus type in nullsec::schema, assigning it the next available ID, adding it to the registry TOML, and submitting a PR to the nullsec repository. The PR must include a round-trip test and a migration guide if the new schema supersedes an existing one.

4. Streaming Mode

For large datasets (full /8 port scans, packet captures), the protocol supports streaming: a stream is a sequence of messages with the same schema ID, terminated by a sentinel end-of-stream message.

```
// Streaming producer
let stream = nullsec::Stream::new(schema::HOST_SCAN_RESULT);
for host in scan_results {
    stream.send(host)?;
}
stream.finish()?

// Streaming consumer
let incoming = nullsec::Stream::receive(schema::HOST_SCAN_RESULT);
for result in incoming {
    let r: HostScanResult = result?;
    process(r);
}
```

The streaming API backpressures automatically: the producer blocks if the consumer's receive buffer is full. This prevents memory exhaustion when scanning large networks.

5. Pipeline Integration

The nullsec protocol integrates with the Lateralus pipeline type system: a tool's input and output schema IDs are part of its function signature. The compiler checks that adjacent tools in a pipeline have compatible schemas.

```
// Tool signatures (from the nullsec crate documentation)
fn port_scan(targets: Vec<IpNet>) -> Stream<HostScanResult>  // schema 1
fn service_detect(host: HostScanResult) -> HostScanResult    // schema 1 in/out
fn vuln_scan(host: HostScanResult) -> Stream<VulnReport>     // schema 3

// Pipeline: compiler verifies schema compatibility at each boundary
let report = ["192.168.1.0/24"]
    |>> port_scan
    |>  service_detect
    |>> vuln_scan
```

6. Authentication and Integrity

In multi-host deployments (distributed scanning from multiple nodes), messages are authenticated with Ed25519 signatures. Each node has a long-term key pair; the signature covers the full message envelope including the header.

```
struct AuthenticatedMessage {
    message:   Message,
    sender_id: Ed25519PublicKey,
    signature: Ed25519Sig,
}

fn verify(msg: &AuthenticatedMessage) -> Result<(), AuthError> {
    let body = encode_cbor(&msg.message);
    ed25519::verify(&msg.sender_id, &body, &msg.signature)
}
```

The signing key is stored in the nullsec trust store and rotated every 30 days. Nodes that have not rotated their keys are automatically quarantined by the cluster manager.

7. Versioning and Forward Compatibility

Schema versions are included in the schema ID encoding: the upper 16 bits of the 32-bit schema ID are the schema version number, and the lower 16 bits are the schema identifier. A tool that receives a schema version higher than it supports can still process the known fields (forward compatibility is guaranteed by the CBOR map structure: unknown fields are ignored).

```
// Schema ID encoding
const HOST_SCAN_RESULT_V1: u32 = 0x0001_0001;
const HOST_SCAN_RESULT_V2: u32 = 0x0002_0001;  // adds geolocation field

// A v1 consumer receiving a v2 message:
// - reads schema_id 0x0002_0001
// - knows only v1 fields (host, open_ports, scan_time)
// - ignores unknown geolocation field (CBOR map, forward-compatible)
// - parses successfully
```

8. Reference Implementation

The reference implementation of the nullsec protocol is in nullsec::proto, a Lateralus library that provides the Message, Stream, and AuthenticatedMessage types, plus the CBOR serialization and deserialization for all registered schemas.

The reference implementation is approximately 600 lines of Lateralus and is tested against 50 round-trip property tests (generated using the quickcheck library) and interoperability tests with a Python reference client.

Lateralus is an open-source, zero-dependency programming language. Project home: https://lateralus.dev. Source: github.com/bad-antics/lateralus-lang. Released under CC BY 4.0.