The NullSec Tool Protocol

bad-antics | November 2024 | Security Engineering

Abstract

This paper specifies the NullSec Tool Protocol (NTP), a standardized communication protocol for coordinating penetration testing tools. Topics include protocol architecture, tool registration, capability discovery, invocation patterns, result exchange, security model, workflow orchestration, error handling, and performance characteristics.

1 Introduction

NullSec Tool Protocol (NTP) is a standardized communication protocol for coordinating penetration testing tools within the NullSec security distribution. NTP enables tool chaining, result sharing, and automated workflow orchestration.

The protocol addresses fragmentation in security tooling: each tool has its own output format, input expectations, and invocation method. NTP provides a unified interface for tool discovery, invocation, and result exchange.

Design goals: language-agnostic communication, minimal overhead, extensibility through capability negotiation, secure transport, and backward compatibility.

2 Protocol Architecture

NTP uses a client-server model. The NTP daemon (ntpd) runs as a local service, managing tool registrations and routing messages between tools. Tools connect as clients.

Transport layer: NTP uses Unix domain sockets for local communication and TLS over TCP for remote communication. The transport is abstracted behind a connection interface.

Message format: NTP messages use a binary header followed by a MessagePack payload. The header contains: protocol version (2 bytes), message type (2 bytes), payload length (4 bytes), and sequence number (4 bytes).

Message types: REGISTER (tool registration), DISCOVER (capability query), INVOKE (tool invocation), RESULT (invocation result), STREAM (streaming output), ERROR (error notification), HEARTBEAT (keepalive).

Connection lifecycle: connect, authenticate, register capabilities, exchange messages, disconnect. Each connection is authenticated before any tool operations.

3 Tool Registration

Tools register with ntpd on startup. Registration includes: tool name, version, supported capabilities, input types, output types, and resource requirements.

Capability descriptors: each capability is described by a name, version, input schema (JSON Schema), output schema, and estimated execution time. Capabilities enable dynamic tool discovery.

Registration persistence: ntpd stores registrations in a local database. If ntpd restarts, tools must re-register. Tools detect ntpd restarts via heartbeat failures.

Deregistration: tools deregister on clean shutdown. If a tool crashes, ntpd detects the closed connection and removes the registration after a timeout.

Tool groups: tools can register as members of a group (e.g., 'reconnaissance', 'exploitation', 'post-exploitation'). Groups enable category-based tool discovery.

4 Discovery and Negotiation

Discovery queries: clients send DISCOVER messages with filters. Filters include: capability name, input type, output type, group, and custom attributes. ntpd returns matching tools.

Capability negotiation: when invoking a tool, the client specifies the desired capability version. The tool responds with the highest compatible version. Incompatible versions produce errors.

Schema validation: ntpd validates invocation payloads against the tool's declared input schema. Invalid payloads are rejected before reaching the tool, providing early error detection.

Load balancing: when multiple tools provide the same capability, ntpd selects the tool based on: current load, estimated execution time, and client preference.

5 Invocation Protocol

Synchronous invocation: the client sends INVOKE, waits for RESULT. The result contains: status code, output data, execution time, and any warnings.

Asynchronous invocation: the client sends INVOKE with an async flag. ntpd returns an invocation ID immediately. The client polls or subscribes for the result.

Streaming invocation: the client sends INVOKE with a stream flag. The tool sends STREAM messages as output is produced. A final RESULT message signals completion.

Timeout handling: invocations have a configurable timeout. If the tool does not respond within the timeout, ntpd sends a cancellation signal and returns a timeout error.

Cancellation: clients can cancel pending invocations by sending a CANCEL message with the invocation ID. The tool receives a cancellation signal and should clean up.

6 Result Exchange

Result format: results are MessagePack-encoded maps. Standard fields: status

(success/failure/partial), data (tool-specific output), metadata (timing, resource usage), and errors (list of error objects).

Typed results: results include a type tag matching the tool's output schema. Consumers validate results against the schema. Type mismatches indicate protocol errors.

Result caching: ntpd can cache results for idempotent invocations. Cached results are returned without re-invoking the tool. Cache keys include: capability, input hash, and tool version.

Result streaming: large results are streamed in chunks. Each chunk has a sequence number and total count. The client reassembles chunks into the complete result.

7 Security Model

Authentication: clients authenticate using mutual TLS (for remote connections) or Unix socket credentials (for local connections). Each client has an identity and associated permissions.

Authorization: ntpd enforces access control lists (ACLs) on tool capabilities. ACLs specify which clients can invoke which capabilities. Default policy: deny all, allow by rule.

Transport encryption: all remote communications use TLS 1.3. Local Unix socket communications are not encrypted (kernel-enforced isolation). Certificate management uses a local CA.

Input sanitization: ntpd sanitizes inputs before forwarding to tools. Sanitization includes: path traversal prevention, command injection prevention, and size limits.

Audit logging: all invocations are logged. Log entries include: client identity, capability invoked, input summary, result status, and timing. Logs support forensic analysis.

8 Workflow Orchestration

Workflow definition: workflows are defined in YAML. Each step specifies: tool capability, input mapping (from previous step outputs or workflow parameters), and error handling.

Pipeline execution: workflow steps execute sequentially. Each step's output is available to subsequent steps. Failed steps can: retry, skip, or abort the workflow.

Parallel execution: steps without data dependencies can execute in parallel. The orchestrator detects independent steps automatically. Parallel execution reduces total workflow time.

Conditional execution: steps can have conditions: run only if a previous step succeeded, only if output matches a pattern, or only on specific platforms.

Workflow templates: common patterns are available as templates: full-scan (reconnaissance to reporting), vulnerability-scan (scan and classify), and exploit-chain (exploit and post-exploit).

9 Error Handling

Error categories: transport errors (connection failures), protocol errors (malformed messages), invocation errors (tool failures), and workflow errors (step failures).

Error propagation: errors propagate from tools through ntpd to clients. Each layer adds context: tool adds execution details, ntpd adds routing details, client adds user context.

Retry strategy: transient errors (network timeouts, resource exhaustion) trigger automatic retries. Retries use exponential backoff. Permanent errors (invalid input, not found) are not retried.

Circuit breaker: if a tool fails repeatedly, ntpd opens a circuit breaker. Subsequent invocations fail immediately without contacting the tool. The breaker resets after a cooldown period.

10 Performance

Message overhead: the binary header adds 12 bytes per message. MessagePack encoding is compact: typically 30-50%% smaller than JSON. Total protocol overhead is minimal.

Latency: local Unix socket round-trip latency is under 100 microseconds. TLS handshake adds 1-2 milliseconds for remote connections. Connection pooling amortizes handshake cost.

Throughput: ntpd handles 10,000+ concurrent connections. Message routing adds less than 10 microseconds per message. The bottleneck is typically tool execution time, not protocol overhead.

11 Conclusion

NTP enables seamless coordination between penetration testing tools. The protocol's design balances flexibility, performance, and security. NTP transforms disparate tools into an integrated security platform.

2.1 Daemon Architecture

ntpd is implemented in Lateralus using async I/O. The daemon uses an event loop to handle concurrent connections without threads. Each connection is a lightweight task.

Connection manager: accepts new connections, performs authentication, and assigns connection handles. The manager maintains a registry of active connections.

Message router: dispatches messages to the appropriate handler based on message type. The router supports: direct routing (client to tool), broadcast (to all tools), and multicast (to a group).

Registry service: stores tool registrations in an in-memory hash map with persistent backup. The registry supports: add, remove, query, and subscribe operations.

Cache service: stores results for idempotent invocations. The cache uses LRU eviction with configurable maximum size. Cache entries have a configurable TTL.

Health monitor: sends periodic heartbeats to connected tools. Tools that miss three consecutive heartbeats are marked as unhealthy. Unhealthy tools are excluded from discovery.

Configuration: ntpd reads configuration from /etc/nullsec/ntpd.conf. Configuration includes: listen address, TLS certificates, authentication settings, ACLs, and logging.

Logging: ntpd logs to syslog and a local file. Log levels: error, warn, info, debug, trace. Log rotation is configured via logrotate or built-in rotation.

Metrics: ntpd exposes Prometheus metrics: active connections, messages per second, invocations per tool, error rate, and latency histograms.

Graceful shutdown: ntpd handles SIGTERM by: stopping new connections, draining active invocations, deregistering tools, and closing connections. Shutdown timeout: 30 seconds.

3.1 Registration Message Format

REGISTER message payload fields: tool_name (string), tool_version (string), capabilities (array of capability objects), pid (integer), and metadata (map of key-value pairs).

Capability object fields: name (string), version (string), description (string), input_schema (JSON Schema object), output_schema (JSON Schema object), estimated_time_ms (integer).

Input schema example: { 'type': 'object', 'properties': { 'target': { 'type': 'string', 'format': 'hostname' }, 'ports': { 'type': 'array', 'items': { 'type': 'integer' } } } }.

Output schema example: { 'type': 'object', 'properties': { 'open_ports': { 'type': 'array', 'items': { 'type': 'object', 'properties': { 'port': { 'type': 'integer' }, 'service': { 'type': 'string' } } } } } }.

Registration response: ntpd responds with: status (accepted/rejected), tool_id (unique identifier assigned by ntpd), and reason (if rejected).

Registration validation: ntpd validates: tool name uniqueness (within the same version), schema validity, and required fields. Invalid registrations are rejected.

Registration update: tools can re-register to update capabilities. The new registration replaces the old one. In-flight invocations continue with the old capabilities.

Bulk registration: tools with many capabilities can register in a single message. Bulk registration reduces connection setup time and message overhead.

Registration events: ntpd emits events when tools register or deregister. Event subscribers (monitoring tools, dashboards) receive real-time tool status updates.

Registration expiry: registrations expire if the tool does not send heartbeats within the configured interval (default: 60 seconds). Expired registrations are removed.

4.1 Discovery Query Language

Query syntax: DISCOVER messages contain a query object with: filters (required capabilities), preferences (desired attributes), and limits (maximum results).

Capability filter: { 'capability': 'port-scan', 'version': '>=2.0' }. The filter matches tools that provide the specified capability with a compatible version.

Type filter: { 'input_type': 'hostname', 'output_type': 'port-list' }. The filter matches tools with compatible input and output types.

Group filter: { 'group': 'reconnaissance' }. The filter matches tools registered in the specified group. Group filters narrow results to a tool category.

Attribute filter: { 'attributes': { 'stealth': true, 'speed': 'fast' } }. Custom attribute filters match tool metadata. Tools declare attributes during registration.

Compound queries: filters can be combined with AND/OR logic. { 'and': [{ 'capability': 'port-scan' }, { 'group': 'reconnaissance' }] }. Compound queries enable precise tool selection.

Discovery response: ntpd returns: matching tools (array of tool descriptors), total count, and pagination token. Tool descriptors include: name, version, capabilities, and load.

Discovery caching: ntpd caches discovery results for frequently-used queries. Cache invalidation occurs on tool registration/deregistration.

Discovery subscription: clients can subscribe to discovery changes. When matching tools register or deregister, the client receives a notification.

Discovery ranking: results are ranked by: compatibility score, current load, historical reliability, and client preference. The highest-ranked tool is recommended.

5.1 Invocation Message Details

INVOKE message fields: invocation_id (UUID), capability (string), version (string), input (MessagePack-encoded data), options (map of invocation options).

Options: timeout_ms (integer), async (boolean), stream (boolean), priority (integer 0-9), retry_count (integer), and cache (boolean, whether to use cached results).

Input validation: ntpd validates the input against the tool's declared schema before forwarding. Validation errors include: field path, expected type, and actual value.

Invocation routing: ntpd selects the target tool based on: capability match, version compatibility, current load, and client preference. The selected tool receives the INVOKE message.

Invocation tracking: ntpd tracks active invocations in memory. Each invocation has: ID, client, tool, start time, timeout, and status. The tracker enables: monitoring, cancellation, and timeout enforcement.

Invocation priority: high-priority invocations are processed before low-priority ones. Priority 0 is

lowest, 9 is highest. Default priority: 5.

Invocation context: the INVOKE message includes a context object with: client identity, workflow ID (if part of a workflow), parent invocation ID (for chained invocations), and trace ID (for distributed tracing).

Invocation limits: ntpd enforces per-client limits: maximum concurrent invocations, maximum payload size, and rate limits. Exceeding limits produces a throttling error.

Invocation logging: each invocation is logged with: start time, end time, duration, input size, output size, and status. Logs enable: performance analysis, debugging, and auditing.

Invocation metrics: ntpd tracks: invocations per second, average latency, error rate, and timeout rate per capability. Metrics are exposed via the Prometheus endpoint.

6.1 Result Processing

Result status codes: SUCCESS (0), PARTIAL (1, some results available), FAILURE (2, tool error), TIMEOUT (3, execution exceeded time limit), CANCELLED (4, client cancelled).

Result metadata: execution_time_ms (actual execution time), memory_used_bytes (peak memory usage), cpu_time_ms (CPU time consumed), and tool_version (version that produced the result).

Result transformation: ntpd supports result transformers. Transformers convert between output formats. Example: XML-to-JSON transformer for legacy tools.

Result aggregation: when invoking multiple tools for the same capability, ntpd aggregates results. Aggregation strategies: merge (combine all results), first (return first success), and vote (majority result).

Result validation: ntpd validates results against the tool's output schema. Invalid results are flagged. The client receives the raw result with a validation warning.

Result persistence: results can be persisted to disk for later analysis. Persistence is configurable per capability. Persisted results are indexed by: invocation ID, timestamp, and capability.

Result compression: large results are compressed using zstd before transmission. Compression reduces bandwidth usage. The compression ratio is included in result metadata.

Result signing: results can be cryptographically signed by the producing tool. Signatures enable: result integrity verification, non-repudiation, and chain-of-custody tracking.

Result filtering: clients can specify output filters to receive only relevant fields. Filtering reduces bandwidth and parsing overhead for large results.

Result pagination: large result sets are paginated. Each page contains: items (array), total_count, page_number, and next_page_token. Clients iterate through pages.

7.1 Authentication Mechanisms

Token authentication: clients authenticate with bearer tokens. Tokens are issued by the ntpd admin tool. Tokens have: expiration time, scope (allowed capabilities), and client identity.

Certificate authentication: clients present X.509 certificates. The certificate's common name identifies the client. The CA is managed by ntpd's built-in PKI.

Unix credential authentication: for local connections, ntpd reads the client's UID/GID from the Unix socket. The UID maps to an identity in the ACL database.

API key authentication: simple key-based authentication for scripts and automation. API keys are stored hashed in the configuration. Keys have: expiration and scope.

Multi-factor authentication: high-privilege operations require: primary authentication (token or certificate) plus a one-time code. MFA prevents unauthorized privileged operations.

Authentication delegation: ntpd can delegate authentication to an external identity provider (LDAP, OAuth2, SAML). Delegation integrates with existing organizational identity systems.

Session management: authenticated clients receive a session token. The session token is used for subsequent messages. Sessions expire after inactivity or a maximum duration.

Credential rotation: tokens and certificates have limited lifetimes. Rotation procedures: generate new credentials, distribute to clients, revoke old credentials. Automated rotation reduces risk.

Failed authentication handling: failed authentication attempts are logged with: client address, attempted identity, failure reason, and timestamp. Repeated failures trigger temporary bans.

Authentication audit: the authentication log records: all authentication attempts (success and failure), credential usage, and session lifecycle events.

7.2 Authorization Model

Role-based access control: roles define sets of allowed capabilities. Example roles: admin (all capabilities), operator (invoke and discover), viewer (discover only).

Capability-level ACLs: ACLs specify which roles can invoke which capabilities. Example: role 'scanner' can invoke 'port-scan' and 'vuln-scan' but not 'exploit'.

Tool-level ACLs: ACLs can restrict access to specific tools, not just capabilities. This enables: per-tool access control and tool-specific rate limits.

Temporal access: ACLs can include time restrictions. Example: exploit capabilities are only available during authorized testing windows.

Scope-based access: ACLs can restrict targets. Example: a client can scan 10.0.0.0/24 but not 192.168.0.0/16. Scope limits prevent accidental or unauthorized scanning.

Policy evaluation: ntpd evaluates policies in order: deny rules first, then allow rules. If no rule matches, the default policy (deny) applies.

Policy hot-reload: ACL changes take effect immediately without restarting ntpd. The admin tool validates policies before applying to prevent lockouts.

Policy logging: every authorization decision is logged. The log includes: client, capability, decision (allow/deny), and matching rule. Logs support: audit and troubleshooting.

Delegated authorization: tools can define their own authorization logic. ntpd forwards the client identity to the tool. The tool makes the final authorization decision.

Emergency override: a master token bypasses all ACLs. The master token is stored offline and used only for emergency recovery. Usage is logged and alerts are sent.

8.1 Workflow Definition Language

YAML-based DSL: workflows are defined in YAML for readability. The DSL supports: steps, variables, conditions, loops, and error handlers.

Step definition: - name: 'Port Scan' capability: 'port-scan' input: { target: '{{ params.target }}' } output: 'scan_result'. Each step has a name, capability, input mapping, and output variable.

Variable interpolation: {{ variable }} references are resolved at execution time. Variables include: workflow parameters, previous step outputs, and environment variables.

Conditional steps: when: '{{ scan_result.open_ports | length > 0 }}'. The step executes only when the condition is true. Conditions use Jinja2-like expressions.

Loop steps: for_each: '{{ scan_result.open_ports }}' as: 'port'. The step body executes for each item. Loop variables are available in the step's input mapping.

Error handlers: on_error: { action: 'retry', max_retries: 3, backoff: 'exponential' }. Error handlers define: retry strategy, fallback steps, and abort conditions.

Parallel blocks: parallel: [step1, step2, step3]. Steps in a parallel block execute concurrently. All steps must complete before the next sequential step.

Sub-workflows: use: 'common/port-scan-workflow.yaml' with: { target: '{{ params.target }}' }. Sub-workflows enable: reuse, composition, and modular workflow design.

Approval gates: approve: { role: 'admin', message: 'Proceed with exploitation?' }. Approval gates pause execution until an authorized user approves. Gates prevent: unauthorized actions.

Workflow validation: the orchestrator validates workflows before execution. Validation checks: step references, variable availability, schema compatibility, and cycle detection.

8.2 Workflow Execution Engine

Execution state machine: workflows progress through states: PENDING, RUNNING, PAUSED, SUCCEEDED, FAILED, CANCELLED. State transitions are logged.

Step execution: the engine invokes the tool, waits for the result, evaluates output mappings, and transitions to the next step. Failed steps trigger error handlers.

Variable store: the engine maintains a variable store for each execution. Variables are: workflow parameters, step outputs, loop variables, and computed values.

Execution persistence: workflow state is persisted to disk. If ntpd restarts, workflows resume from the last checkpoint. Persistence ensures: workflow durability.

Execution monitoring: the engine exposes: current step, elapsed time, step durations, and variable values. Monitoring enables: real-time progress tracking.

Execution cancellation: clients can cancel running workflows. The engine cancels the current step, skips remaining steps, and runs cleanup handlers.

Execution history: completed workflows are stored in the execution history. The history includes: all step results, timing, and variable snapshots. History enables: audit and replay.

Execution replay: past workflows can be replayed with modified parameters. Replay uses: the original workflow definition, new parameters, and the current tool versions.

Execution scheduling: workflows can be scheduled for: immediate execution, future execution (cron-like), or triggered execution (on event).

Execution quotas: the engine enforces quotas: maximum concurrent workflows, maximum workflow duration, and maximum steps per workflow. Quotas prevent resource exhaustion.

9.1 Error Recovery Patterns

Retry with backoff: transient errors trigger retries with exponential backoff. Initial delay: 100ms. Maximum delay: 30 seconds. Maximum retries: 5 (configurable).

Fallback tool: if the primary tool fails, the orchestrator invokes an alternative tool with the same capability. Fallback tools are specified in the workflow.

Partial results: if a tool produces partial results before failing, the partial results are preserved. Subsequent steps can use partial results with a degraded-result flag.

Compensation: workflows can define compensation steps that undo previous steps on failure. Compensation enables: rollback, cleanup, and consistent state.

Dead letter queue: unprocessable messages are moved to a dead letter queue. The queue enables: manual inspection, reprocessing, and debugging.

Error escalation: repeated errors escalate from: automatic retry to operator notification to workflow abort. Escalation levels are configurable per step.

Health-based routing: if a tool is unhealthy (recent failures), ntpd routes invocations to healthy alternatives. Health-based routing improves: reliability.

Error correlation: ntpd correlates errors across invocations using trace IDs. Correlated errors reveal: systemic issues, dependency failures, and cascading failures.

Error reporting: ntpd generates error reports: error frequency, error distribution by tool, error trends, and resolution suggestions. Reports guide: maintenance and improvement.

Recovery testing: the chaos engineering mode injects: network delays, tool crashes, and resource exhaustion. Recovery testing validates: error handling, retry logic, and workflow resilience.

10.1 Protocol Benchmarks

Registration throughput: ntpd handles 5,000 tool registrations per second. Registration latency: p50 = 0.2ms, p99 = 1.5ms. Benchmark: 10,000 tools registering concurrently.

Discovery throughput: ntpd handles 50,000 discovery queries per second. Discovery latency: p50 = 0.05ms, p99 = 0.3ms. Benchmark: complex queries with multiple filters.

Invocation throughput: ntpd routes 20,000 invocations per second (passthrough). Invocation latency overhead: p50 = 0.1ms, p99 = 0.5ms. Benchmark: 1,000 concurrent clients.

Streaming throughput: sustained 1 GB/s streaming bandwidth per connection. Streaming latency: first byte within 0.5ms. Benchmark: large nmap scan output.

Memory usage: ntpd uses 50 MB base + 10 KB per connection + 1 KB per cached result. For 10,000 connections with 100,000 cached results: approximately 250 MB.

CPU usage: ntpd uses less than 5%% CPU at 10,000 messages per second on a modern core. CPU scales linearly with message rate. Multi-core scaling: near-linear up to 8 cores.

Network overhead: NTP adds 12 bytes header + MessagePack encoding overhead per message. For a typical invocation: 100 bytes total overhead. Protocol efficiency: 95%%+ payload ratio.

Workflow overhead: orchestrating a 10-step workflow adds approximately 5ms total overhead. Step scheduling: 0.3ms. Variable resolution: 0.1ms. State persistence: 0.4ms.

Cache hit rate: for idempotent capabilities (DNS lookup, WHOIS), cache hit rates exceed 80%%. Caching reduces: tool invocations, execution time, and target load.

Comparison with alternatives: NTP outperforms REST-based tool orchestration (10x lower latency) and gRPC-based orchestration (3x higher throughput) due to minimal protocol overhead.

2.2 Message Encoding

MessagePack encoding: NTP uses MessagePack for payload serialization. MessagePack is a binary format that is compact, fast to encode/decode, and supports: integers, floats, strings, binary, arrays, and maps.

Schema evolution: new fields are added as optional map entries. Old clients ignore unknown fields. Removed fields become optional. This enables backward-compatible protocol evolution.

Binary payloads: tool outputs (screenshots, packet captures, binary data) are encoded as MessagePack binary type. Large binaries use streaming to avoid memory pressure.

Nested messages: MessagePack maps can be nested arbitrarily. Tool-specific payloads use nested maps for structured data. The protocol does not impose nesting limits.

Encoding performance: MessagePack encoding is 3-5x faster than JSON. Decoding is 2-3x faster. Binary size is 30-50%% smaller. The performance advantage is significant for high-throughput scenarios.

Timestamp encoding: timestamps use the MessagePack timestamp extension type. Timestamps have nanosecond precision. All timestamps are in UTC.

Custom extension types: NTP defines custom MessagePack extension types for: UUID (type 1), IP address (type 2), CIDR range (type 3), and certificate fingerprint (type 4).

Encoding validation: the MessagePack decoder validates: type correctness, string encoding (UTF-8), and maximum size limits. Invalid encodings produce parse errors.

Zero-copy decoding: the decoder supports zero-copy access to binary and string fields. Zero-copy decoding avoids memory allocation for large fields.

Encoding alternatives: while MessagePack is the primary encoding, NTP supports JSON as a fallback for debugging and interoperability. JSON mode is enabled per-connection.

3.2 Tool Capability Model

Capability hierarchy: capabilities are organized in a hierarchy. 'scan' is a parent of 'port-scan', 'vuln-scan', and 'web-scan'. Querying 'scan' returns all child capabilities.

Capability versioning: capabilities follow semantic versioning. Major version changes indicate breaking changes. Minor version changes add functionality. Patch versions fix bugs.

Capability dependencies: a capability can depend on other capabilities. Example: 'exploit' depends on 'vuln-scan' (to identify the vulnerability). ntpd resolves dependencies automatically.

Capability composition: tools can compose capabilities from sub-capabilities. Example: 'full-scan' composes 'port-scan', 'service-detect', and 'vuln-scan'. Composition reduces boilerplate.

Capability constraints: capabilities can declare constraints: target type (host, network, URL), required privileges (root, user), and platform (Linux, Windows).

Capability tags: tools tag capabilities with metadata: stealth level (passive, active, aggressive), speed (fast, thorough), and accuracy (high, medium, low).

Capability search: ntpd supports full-text search over capability names and descriptions. Search results are ranked by relevance. Fuzzy matching handles typos.

Capability documentation: each capability includes inline documentation: usage examples, parameter

descriptions, output format, and known limitations.

Capability testing: tools can expose test capabilities that verify: correct setup, connectivity, and required permissions. Test capabilities are invoked during: registration and health checks.

Capability deprecation: deprecated capabilities include: sunset date, replacement capability, and migration guide. ntpd warns clients that invoke deprecated capabilities.

5.2 Invocation Chaining

Chain definition: an invocation chain passes the output of one tool as input to the next. Chains are defined inline or as part of a workflow.

Implicit chaining: ntpd automatically chains tools when output and input schemas are compatible. The output of the first tool is mapped to the input of the second.

Explicit mapping: chain steps specify field mappings: { source: 'output.open_ports', target: 'input.ports' }. Explicit mappings handle: renaming, filtering, and transformation.

Chain error handling: if a chain step fails, the entire chain can: abort (default), skip (continue with empty input), or retry (re-invoke the failed step).

Chain parallelism: chain steps that are independent can execute in parallel. ntpd detects independence by analyzing input/output dependencies.

Chain caching: intermediate chain results are cached. If a chain is re-invoked with the same initial input, cached intermediate results are reused.

Chain monitoring: chain execution is tracked as a single unit. Monitoring shows: current step, elapsed time, and intermediate results.

Chain templates: common chains are available as templates: reconnaissance-chain (host-discovery to port-scan to service-detect), exploit-chain (vuln-scan to exploit to post-exploit).

Chain validation: ntpd validates chains before execution. Validation checks: schema compatibility between steps, required capabilities availability, and cycle detection.

Chain optimization: ntpd optimizes chains by: merging compatible steps (reducing network overhead), parallelizing independent steps, and caching deterministic steps.

7.3 Secure Communication Patterns

Forward secrecy: TLS 1.3 provides forward secrecy. Compromising the server's long-term key does not compromise past sessions. Each session uses ephemeral keys.

Certificate pinning: clients can pin the server's certificate. Pinning prevents: man-in-the-middle attacks with rogue certificates. Pinned certificates are distributed out-of-band.

Mutual authentication: both client and server present certificates. Mutual TLS ensures both parties

are authenticated. The server verifies the client certificate against the CA.

Channel binding: session tokens are bound to the TLS channel. Tokens stolen from one channel cannot be used on another. Channel binding prevents: token relay attacks.

Secure transport configuration: minimum TLS version: 1.3. Cipher suites: TLS_AES_256_GCM_SHA384, TLS_CHACHA20_POLY1305_SHA256. Key exchange: X25519, secp384r1.

Certificate lifecycle: certificates have a 90-day validity period. Automatic renewal via ACME protocol. Revocation via CRL and OCSP. Certificate transparency logging.

Key management: private keys are stored in the system keychain or HSM. Key access requires: process identity verification. Keys are never written to disk in plaintext.

Network segmentation: ntpd supports binding to specific network interfaces. Tool traffic can be isolated to a dedicated VLAN. Segmentation limits the blast radius of compromised tools.

Protocol downgrade prevention: NTP rejects connections using older protocol versions. Version negotiation uses a secure handshake that prevents downgrade attacks.

Intrusion detection: ntpd monitors for: unusual message patterns, authentication anomalies, and protocol violations. Detected anomalies trigger: logging, alerting, and connection termination.

10.2 Deployment Guide

Installation: ntpd is included in the NullSec distribution. Manual installation: lateralus-pkg install nullsec-ntpd. Configuration: /etc/nullsec/ntpd.conf.

Service management: systemd unit file: nullsec-ntpd.service. Start: systemctl start nullsec-ntpd. Enable: systemctl enable nullsec-ntpd. Status: systemctl status nullsec-ntpd.

Initial configuration: generate TLS certificates: ntpd-admin init-pki. Create admin token: ntpd-admin create-token --role admin. Configure ACLs: /etc/nullsec/acl.conf.

Tool integration: tools integrate using the NTP client library. Client libraries are available for: Lateralus, Python, Go, and C. Integration requires: connect, authenticate, register.

Network configuration: default listen address: /run/nullsec/ntpd.sock (local) and 0.0.0.0:4443 (remote). Firewall rules: allow TCP 4443 from authorized clients.

High availability: ntpd supports active-passive failover. The active instance handles all traffic. The passive instance monitors the active and takes over on failure.

Backup and recovery: ntpd state is backed up to: /var/lib/nullsec/ntpd/backup/. Backup includes: registrations, ACLs, and execution history. Recovery: ntpd-admin restore.

Monitoring integration: ntpd integrates with: Prometheus (metrics), Grafana (dashboards), and Alertmanager (alerts). Pre-built dashboards are included.

Upgrade procedure: stop ntpd, install new version, run ntpd-admin migrate (if needed), start ntpd. Rolling upgrades are supported in HA configurations.

Troubleshooting: common issues: connection refused (ntpd not running), authentication failed (expired token), tool not found (not registered). Diagnostic: ntpd-admin status.

9.2 Protocol Extension Mechanism

Extension points: NTP supports protocol extensions via: custom message types (0x80-0xFF), custom capability attributes, and custom result metadata fields.

Extension registration: extensions are registered with ntpd using the REGISTER_EXTENSION message. Registration includes: extension name, version, message type ranges, and schema.

Extension discovery: DISCOVER messages can query available extensions. Clients check for required extensions before using extension-specific features.

Extension negotiation: clients and tools negotiate extension support during connection setup. Both parties declare supported extensions. Only mutually supported extensions are used.

Custom message types: extensions define message types in the range 0x80-0xFF. Custom message handlers are registered with ntpd. ntpd routes custom messages to the registered handler.

Extension versioning: extensions follow semantic versioning. Incompatible extension versions produce negotiation failures. ntpd logs version mismatches for debugging.

Built-in extensions: NTP includes standard extensions: NTP-VULN (vulnerability reporting format), NTP-CRED (credential exchange), NTP-NET (network topology), NTP-REPORT (report generation).

Extension sandboxing: extension handlers run in a sandboxed environment. Sandboxing prevents: memory corruption, resource exhaustion, and privilege escalation from extension code.

Extension marketplace: the NullSec community maintains an extension registry. Community extensions are reviewed for: security, quality, and compatibility.

Extension development kit: the NTP EDK provides: message type allocation, schema validation helpers, test harness, and documentation templates for extension authors.

8.3 Reporting Integration

Report generation: workflows can include report generation steps. Reports aggregate: tool outputs, findings, evidence, and recommendations.

Report formats: HTML (interactive, with charts and tables), PDF (printable, formal), JSON (machine-readable, for integration), and Markdown (simple, version-controllable).

Report templates: predefined templates for: vulnerability assessment, penetration test, compliance audit, and incident response. Templates are customizable.

Finding classification: findings are classified by: severity (critical, high, medium, low, informational), confidence (confirmed, probable, possible), and category (OWASP, CWE, MITRE ATT&CK).

Evidence collection: tools attach evidence to findings: screenshots, packet captures, request/response pairs, and command output. Evidence supports finding validation.

Remediation tracking: findings include remediation recommendations. The tracking system monitors: remediation status (open, in-progress, resolved, accepted), assigned owner, and due date.

Report comparison: reports from different time periods can be compared. Comparison shows: new findings, resolved findings, and persistent findings. Trend analysis tracks security posture over time.

Report distribution: reports are distributed via: email, shared drive, ticketing system integration, and API. Distribution is configurable per report type.

Report signing: reports are cryptographically signed. Signatures ensure: report integrity, author authentication, and non-repudiation. Signed reports are legally admissible.

Compliance mapping: findings are mapped to compliance frameworks: PCI DSS, HIPAA, SOC 2, ISO 27001. Compliance mapping simplifies: audit preparation and gap analysis.

11.1 Protocol Evolution

Version negotiation: clients declare their supported protocol versions during connection setup. ntpd selects the highest mutually supported version. Unsupported versions are rejected.

Backward compatibility: new protocol versions maintain backward compatibility with the previous major version. Deprecated features are supported for one major version after deprecation.

Feature flags: new capabilities are introduced behind feature flags. Feature flags enable gradual rollout: test with early adopters, then enable for all clients.

Protocol deprecation: deprecated protocol versions are announced 6 months before removal. Clients using deprecated versions receive: warnings, migration guides, and update reminders.

Migration tools: ntpd includes tools for: protocol version migration, configuration conversion, and client library updates. Migration tools minimize manual effort.

Specification process: protocol changes follow a RFC process. Proposals are: submitted, reviewed, discussed, revised, and accepted. Accepted RFCs become part of the next protocol version.

Reference implementation: the ntpd codebase serves as the reference implementation. The reference implementation is: well-documented, extensively tested, and publicly available.

Conformance testing: the NTP conformance test suite validates implementations against the specification. Conformance tests cover: message format, behavior, error handling, and security.

Interoperability testing: different client library implementations are tested for interoperability. Interoperability tests ensure: consistent behavior across languages and platforms.

Community feedback: protocol development is open to community feedback. Feedback channels: issue tracker, discussion forum, and periodic community meetings.

2.3 Connection Management

Connection pooling: clients maintain a pool of connections to ntpd. Pooling avoids repeated TLS handshakes. Pool size is configurable per client.

Connection multiplexing: multiple concurrent invocations share a single connection. Multiplexing reduces resource usage. Messages are interleaved with sequence numbers for demultiplexing.

Connection health: ntpd sends heartbeat messages every 15 seconds. Clients respond with heartbeat-ack. Three missed heartbeats close the connection.

Connection migration: if a client's network address changes, the session token allows re-authentication on the new connection without re-registration.

Connection backpressure: if a client sends messages faster than ntpd can process, ntpd signals backpressure. The client reduces its send rate. Backpressure prevents buffer overflow.

Connection draining: before shutdown, ntpd drains connections by: completing in-flight invocations, rejecting new invocations, and notifying clients of pending shutdown.

Connection statistics: ntpd tracks per-connection: messages sent/received, bytes transferred, invocations completed, errors, and connection duration.

Connection limits: ntpd enforces: maximum total connections, maximum connections per client, and maximum connections per IP. Limits prevent resource exhaustion from misbehaving clients.

Connection recovery: if a connection drops unexpectedly, the client reconnects and re-authenticates. Pending invocations are retried. The recovery is transparent to the application.

Connection debugging: ntpd provides a diagnostic mode that logs: all messages on a connection, timing details, and protocol state transitions. Debugging mode is enabled per-connection.

4.2 Service Discovery Integration

DNS-based discovery: tools and clients discover ntpd via DNS SRV records. _ntp._tcp.nullsec.local resolves to the ntpd host and port.

mDNS discovery: on local networks, ntpd advertises via multicast DNS. Clients on the same network discover ntpd automatically.

Static configuration: clients can configure ntpd addresses statically. Static configuration is used in: production environments, air-gapped networks, and containers.

Consul integration: ntpd registers as a service in Consul. Clients query Consul for the ntpd address. Health checks ensure only healthy instances are returned.

Kubernetes discovery: in Kubernetes, ntpd runs as a Service. Clients use the service DNS name (ntpd.nullsec.svc.cluster.local) to connect.

Load balancer integration: ntpd instances behind a load balancer. The load balancer distributes connections. Sticky sessions ensure consistent routing for stateful operations.

Failover discovery: clients maintain a list of ntpd addresses. If the primary is unreachable, clients try alternates. Failover is transparent to the application.

Discovery caching: resolved ntpd addresses are cached by clients. Cache TTL: 60 seconds. Caching reduces discovery overhead for frequent connections.

Discovery security: DNS-based discovery uses DNSSEC to prevent spoofing. mDNS uses local network trust. Consul and Kubernetes use their built-in authentication.

Discovery events: ntpd emits service discovery events when: instances join or leave, capabilities change, or health status changes. Event consumers update their routing tables.

6.2 Result Correlation

Correlation IDs: each workflow assigns a correlation ID to all invocations. The correlation ID links: tool invocations, results, and log entries across the workflow.

Cross-tool correlation: results from different tools for the same target are correlated. Example: port scan results are correlated with vulnerability scan findings for the same host.

Temporal correlation: results are correlated across time. Changes between scans are highlighted: new findings, resolved findings, and changed severity.

Finding deduplication: duplicate findings from multiple tools are merged. Deduplication uses: vulnerability ID, target, and affected component. Merged findings include all evidence.

Confidence scoring: correlated findings receive higher confidence scores. A vulnerability confirmed by multiple tools has higher confidence than one tool's finding.

Attack path reconstruction: correlated findings are assembled into attack paths. An attack path shows: entry point, pivot points, and target assets. Paths represent realistic attack scenarios.

Correlation rules: administrators define correlation rules. Rules specify: which findings to correlate, how to merge, and resulting severity adjustments.

Correlation engine: the engine processes findings in near-real-time. As tools produce results, the engine correlates and updates the finding database.

Correlation visualization: correlated findings are displayed on: a network topology map, a finding timeline, and an attack graph. Visualization helps prioritize remediation.

Correlation API: the correlation engine exposes an API for: querying correlated findings, subscribing to new correlations, and managing correlation rules.

3.3 Tool Lifecycle Management

Tool versioning: tools declare their version during registration. ntpd tracks multiple versions of the same tool. Clients can specify version requirements in invocations.

Tool update notification: when a new version of a tool is registered, ntpd notifies clients that use the tool. Notifications include: version number, changelog, and migration notes.

Tool deprecation: deprecated tools are marked in the registry. Invocations to deprecated tools return a warning header. Deprecated tools are removed after the sunset date.

Tool health monitoring: ntpd periodically invokes tool health checks. Health checks verify: tool availability, correct configuration, and required permissions.

Tool resource limits: ntpd enforces resource limits per tool: maximum memory, maximum CPU time, maximum concurrent invocations, and maximum output size.

Tool isolation: tools run in isolated environments (containers, namespaces). Isolation prevents: tool interference, resource contention, and security breaches.

Tool logging: tool stdout/stderr is captured and stored. Logs are accessible via: the ntpd API, the admin console, and the monitoring system.

Tool crash recovery: if a tool crashes during invocation, ntpd detects the crash, returns an error to the client, and optionally restarts the tool.

Tool configuration: tool-specific configuration is stored in ntpd. Configuration is passed to tools during invocation. Configuration updates take effect on the next invocation.

Tool metrics: ntpd collects per-tool metrics: invocation count, success rate, average latency, error rate, and resource usage. Metrics are exposed via Prometheus.

5.3 Invocation Patterns

Fire-and-forget: the client sends INVOKE and does not wait for a result. Useful for: logging, notifications, and non-critical operations. ntpd acknowledges receipt.

Request-response: the standard pattern. The client sends INVOKE, waits for RESULT. Timeout handling ensures the client does not wait indefinitely.

Publish-subscribe: tools publish findings to named topics. Clients subscribe to topics and receive findings in real-time. Topics include: 'new-vulnerability', 'host-discovered', 'credential-found'.

Scatter-gather: the client invokes the same capability on multiple tools. Results are gathered and aggregated. Useful for: verification (multiple tools confirm a finding) and coverage (different tools find different issues).

Pipeline: the output of one tool feeds directly into the next tool. Pipelines are defined as ordered lists of capabilities. ntpd manages data flow between pipeline stages.

Map-reduce: the client distributes work across multiple tool instances. Each instance processes a subset (map). Results are combined into a single result (reduce).

Observer: clients register observers for specific tools or capabilities. When an invocation completes, all registered observers receive the result.

Saga: long-running operations are broken into steps with compensation. If a step fails, previous steps are compensated (rolled back). Sagas ensure consistency for multi-step operations.

Batch: clients send multiple invocations in a single batch message. ntpd processes batch items concurrently and returns a batch result. Batching reduces network round-trips.

Priority queue: invocations are queued by priority. High-priority invocations (active incident response) preempt low-priority ones (scheduled scans).

7.4 Threat Model

Attacker model: the protocol considers: network eavesdroppers, compromised tools, compromised clients, and compromised ntpd instances. Each threat has specific mitigations.

Network eavesdropping: mitigated by TLS encryption. Even on compromised networks, message contents are confidential. Certificate verification prevents man-in-the-middle attacks.

Compromised tool: mitigated by: tool isolation (containers), resource limits, output validation, and ACLs. A compromised tool cannot access other tools' data or escalate privileges.

Compromised client: mitigated by: role-based access control, scope restrictions, rate limits, and audit logging. A compromised client can only access its authorized capabilities.

Compromised ntpd: ntpd is the most critical component. Mitigations: minimal attack surface, memory-safe implementation (Lateralus), regular security audits, and integrity monitoring.

Replay attacks: mitigated by: message sequence numbers, timestamp validation, and nonce-based authentication. Replayed messages are detected and rejected.

Denial of service: mitigated by: connection limits, rate limits, backpressure, and resource quotas. DoS attempts are detected and the source is rate-limited or blocked.

Privilege escalation: mitigated by: strict ACLs, capability-based access, and separation of duties. No single credential grants all capabilities.

Data exfiltration: mitigated by: output validation, egress filtering, and audit logging. Unusual data patterns (large outputs, encoded payloads) trigger alerts.

Supply chain: mitigated by: tool signature verification, checksum validation, and trusted tool repositories. Only verified tools are registered.

8.4 Workflow Library

Network reconnaissance workflow: 1) host discovery (ARP/ICMP), 2) port scanning, 3) service detection, 4) OS fingerprinting, 5) report generation.

Vulnerability assessment workflow: 1) asset inventory, 2) vulnerability scanning, 3) finding classification, 4) risk scoring, 5) remediation prioritization, 6) report.

Web application testing workflow: 1) spider/crawl, 2) passive analysis, 3) active scanning (SQL injection, XSS, CSRF), 4) authentication testing, 5) report.

Password audit workflow: 1) hash extraction, 2) dictionary attack, 3) rule-based attack, 4) brute force (limited), 5) results analysis, 6) policy recommendations.

Wireless network assessment: 1) network discovery, 2) encryption analysis, 3) client enumeration, 4) deauthentication testing, 5) evil twin detection, 6) report.

Cloud infrastructure audit: 1) asset discovery, 2) configuration review, 3) IAM analysis, 4) network analysis, 5) storage audit, 6) compliance check, 7) report.

Social engineering assessment: 1) OSINT gathering, 2) pretext development, 3) phishing campaign, 4) response tracking, 5) awareness metrics, 6) report.

Incident response workflow: 1) alert triage, 2) evidence collection, 3) IOC extraction, 4) scope determination, 5) containment actions, 6) timeline reconstruction, 7) report.

Red team engagement workflow: 1) reconnaissance, 2) initial access, 3) persistence, 4) privilege escalation, 5) lateral movement, 6) data collection, 7) exfiltration simulation, 8) report.

Compliance scan workflow: 1) scope definition, 2) control enumeration, 3) evidence collection, 4) gap analysis, 5) risk assessment, 6) remediation plan, 7) report.

10.3 Scalability Analysis

Horizontal scaling: ntpd instances can be deployed behind a load balancer. Each instance handles a subset of connections. State synchronization uses a shared database.

Vertical scaling: single ntpd instances scale to 50,000 concurrent connections on hardware with 8 cores and 16 GB RAM.

Tool scaling: tools can be scaled independently. Multiple instances of the same tool register with ntpd. ntpd distributes invocations across instances.

Workflow scaling: workflow executions are distributed across ntpd instances. Each instance handles a subset of active workflows. Workflow state is shared via the database.

Storage scaling: result storage uses a pluggable backend. Backends include: local filesystem, S3-compatible object storage, and distributed databases. Storage scales independently.

Network scaling: ntpd supports network segmentation. Different tool categories can run on different network segments. Cross-segment communication uses encrypted tunnels.

Geographic scaling: ntpd instances can be deployed in multiple regions. Region-local tools serve local clients. Cross-region invocations are routed automatically.

Cache scaling: the result cache uses a distributed cache (Redis, Memcached). Distributed caching scales with the number of ntpd instances.

Monitoring scaling: metrics are aggregated across instances. A centralized monitoring system (Prometheus + Grafana) displays: cluster-wide metrics, per-instance metrics, and per-tool metrics.

Capacity planning: ntpd includes a capacity planning tool. The tool simulates: expected load, tool registration count, and workflow complexity. The tool recommends: instance count, hardware specs, and network bandwidth.

2.4 Protocol State Machine

Connection states: CONNECTING, AUTHENTICATING, READY, CLOSING, CLOSED. Each state has defined transitions. Invalid transitions produce protocol errors.

CONNECTING state: the transport layer establishes a connection (TCP handshake, TLS negotiation). Transition to AUTHENTICATING on successful connection.

AUTHENTICATING state: the client sends credentials. ntpd validates credentials. Transition to READY on success, CLOSING on failure.

READY state: the client can send: REGISTER, DISCOVER, INVOKE, CANCEL, HEARTBEAT. ntpd can send: RESULT, STREAM, ERROR, HEARTBEAT.

CLOSING state: initiated by either party. In-flight messages are drained. No new messages are accepted. Transition to CLOSED after drain completes or timeout.

CLOSED state: the connection is terminated. All resources are released. Registrations from this connection are deregistered after timeout.

State timeouts: CONNECTING: 5 seconds. AUTHENTICATING: 10 seconds. READY: no timeout (keepalive via heartbeats). CLOSING: 30 seconds. Timeout violations close the connection.

State monitoring: ntpd tracks connections by state. Metrics include: connections per state, state transition rates, and average time in each state.

State recovery: if a connection enters an invalid state, ntpd logs the error and forcefully closes the connection. The client reconnects and re-authenticates.

State persistence: connection state is not persisted. If ntpd restarts, all connections are lost. Clients detect the restart via heartbeat failure and reconnect.

3.4 Tool SDK

Lateralus SDK: the primary SDK provides: connection management, message serialization, capability declaration, and result formatting. The SDK handles protocol details.

Python SDK: a Python binding using ctypes. The Python SDK enables: rapid tool development, scripting integration, and prototyping. Performance-critical tools use the Lateralus SDK.

Go SDK: a Go binding with native implementation. The Go SDK is used for: containerized tools, cloud-native tools, and tools with Go dependencies.

C SDK: a minimal C binding for: legacy tools, embedded tools, and tools with strict resource constraints. The C SDK provides: raw message access and manual memory management.

SDK architecture: all SDKs share: the same protocol implementation, message format, and error handling. SDK behavior is consistent across languages.

SDK features: connection pooling, automatic reconnection, heartbeat management, schema validation, and result caching. All features are configurable.

SDK testing: the SDK includes a mock ntpd for unit testing. The mock simulates: registration, discovery, invocation, and error scenarios. Tests run without a real ntpd instance.

SDK documentation: each SDK has: API reference, getting-started guide, example tools, and migration guide. Documentation is versioned alongside the SDK.

SDK versioning: SDK versions match the protocol version. SDK 3.x supports protocol 3.x. SDKs maintain backward compatibility within a major version.

SDK code generation: the SDK includes a code generator that produces: tool scaffolding, capability stubs, and test templates from a capability schema file.

6.3 Result Storage

Storage backends: local filesystem (development), SQLite (single-instance), PostgreSQL (production), S3-compatible (archival). Backend is configured per ntpd instance.

Storage schema: results are stored with: invocation_id (primary key), workflow_id, capability, tool_name, timestamp, status, data (JSONB), and metadata.

Indexing: indexes on: invocation_id, workflow_id, capability, timestamp, and status. Composite indexes for common query patterns: (capability, timestamp), (workflow_id, status).

Retention policy: results are retained for: 30 days (default), 90 days (compliance mode), 365 days (archival mode). Expired results are deleted or moved to archival storage.

Storage encryption: results at rest are encrypted using AES-256-GCM. Encryption keys are stored in: the system keychain, HashiCorp Vault, or AWS KMS.

Storage compression: stored results are compressed using zstd. Compression ratio: 3-5x for typical security tool output. Compression reduces: storage cost and backup size.

Storage search: full-text search over result data using: PostgreSQL tsvector (production) or SQLite FTS5 (development). Search enables: finding specific vulnerabilities across all results.

Storage export: results can be exported in: JSON, CSV, and STIX (Structured Threat Information Expression) formats. Export is used for: integration, archival, and reporting.

Storage migration: the ntpd-admin migrate command handles schema changes between versions. Migrations are: versioned, reversible, and tested. Migration runs during upgrade.

Storage monitoring: metrics include: storage size, row count, query latency, and compression ratio. Alerts trigger when: storage exceeds threshold, query latency spikes, or backup fails.

9.3 Logging and Audit Trail

Structured logging: ntpd uses structured JSON logging. Each log entry includes: timestamp, level, component, message, and context fields (connection_id, invocation_id, tool_name).

Audit trail: all security-relevant events are logged to a separate audit log. Audit events include: authentication, authorization, invocation, result access, and configuration changes.

Log levels: ERROR (unrecoverable failures), WARN (recoverable issues), INFO (normal operations), DEBUG (detailed diagnostics), TRACE (protocol-level detail).

Log rotation: logs are rotated by: size (100 MB default), time (daily), or both. Rotated logs are compressed and archived. Retention: 30 days for application logs, 365 days for audit logs.

Log aggregation: ntpd supports log forwarding to: syslog, Elasticsearch, Splunk, and CloudWatch. Aggregated logs enable: centralized search, correlation, and alerting.

Log filtering: log output can be filtered by: component, level, and context fields. Filtering reduces: log volume, storage cost, and noise in analysis.

Tamper detection: audit logs include: sequential numbering, cryptographic chaining (hash of previous entry), and digital signatures. Tampering is detected by: hash chain verification.

Log analysis: ntpd includes log analysis tools: query language for structured logs, anomaly detection for unusual patterns, and summary reports for daily operations.

Compliance logging: audit logs satisfy compliance requirements for: PCI DSS (access logging), HIPAA (audit trail), SOC 2 (monitoring), and ISO 27001 (event logging).

Log privacy: sensitive data in logs is: redacted (passwords replaced with ***), hashed (IP addresses hashed for privacy), or encrypted (PII encrypted at rest).

4.3 Capability Matching Algorithm

Exact match: the query capability name exactly matches a registered capability name. Exact matches have the highest priority.

Version matching: semver ranges are evaluated. ^2.0 matches 2.x.x. ~2.1 matches 2.1.x. >=2.0, <3.0 matches any 2.x.x. Version matching follows semver specification.

Hierarchical match: if no exact match, the query walks up the capability hierarchy. Querying 'tcp-port-scan' matches 'port-scan' if no specific match exists.

Type compatibility: input and output types are checked for structural compatibility. A tool accepting { target: string, ports: [int] } matches a query for { target: string }.

Attribute scoring: each matching capability receives a score based on: attribute match count, attribute value similarity, and attribute weight. The highest-scoring capability is preferred.

Preference handling: client preferences override default ranking. Preferences include: preferred tool names, preferred versions, and excluded tools.

Fallback matching: if no capability matches, ntpd suggests: related capabilities, similar tool names, and available capability hierarchies.

Match caching: matching results are cached per query pattern. Cache invalidation occurs on: tool registration, deregistration, and capability update.

Match performance: the matching algorithm runs in O(log n) time for exact matches (using a sorted index) and O(n) for attribute scoring (scanning all candidates).

Match testing: the ntpd-admin tool includes a match tester. Administrators can test queries against the current registry without invoking any tools.

7.5 Compliance and Legal

Rules of engagement: NTP workflows enforce rules of engagement. Target scope restrictions prevent: testing unauthorized targets. Scope violations trigger: alerts and workflow abort.

Legal compliance: NTP logging satisfies evidentiary requirements. Chain-of-custody tracking ensures: evidence integrity. Time-stamped logs provide: audit trail for legal proceedings.

Data classification: NTP supports data classification labels: PUBLIC, INTERNAL, CONFIDENTIAL, RESTRICTED. Classification controls: storage, transmission, and access.

Data retention: retention policies comply with: legal hold requirements, regulatory retention periods, and organizational policies. Expired data is securely deleted.

Privacy impact: NTP handles sensitive data (credentials, vulnerabilities). Privacy measures: data minimization, purpose limitation, access control, and encryption.

Export control: NTP tools may be subject to export controls. The tool registry includes: export classification, restricted countries, and compliance requirements.

Vulnerability disclosure: NTP supports coordinated vulnerability disclosure. Findings can be: embargoed (hidden until fix), disclosed (public after deadline), or private (restricted access).

Report confidentiality: assessment reports are classified as CONFIDENTIAL by default. Distribution is limited to: authorized recipients, secure channels, and encrypted storage.

Operator certification: NTP can require operator certification for high-risk capabilities. Certification verifies: training completion, authorization, and competency.

Incident documentation: NTP workflows generate incident documentation: timeline, actions taken, tools used, and findings. Documentation satisfies: regulatory requirements and legal proceedings.

10.4 Integration Examples

Nmap integration: nmap registers capabilities for: host-discovery, port-scan, service-detect, os-fingerprint. Input: target (CIDR). Output: host list with open ports and services.

Metasploit integration: Metasploit registers capabilities for: exploit, payload-delivery, post-exploit. Input: target, vulnerability ID. Output: session handle, exploitation result.

Burp Suite integration: Burp registers capabilities for: web-spider, passive-scan, active-scan. Input: target URL. Output: finding list with evidence.

Hashcat integration: Hashcat registers capabilities for: hash-crack, wordlist-attack, rule-attack. Input: hash list, attack parameters. Output: cracked credentials.

Wireshark integration: Wireshark registers capabilities for: packet-capture, protocol-analysis, traffic-statistics. Input: interface, filter. Output: PCAP file, analysis summary.

SQLMap integration: SQLMap registers capabilities for: sql-injection-detect, sql-injection-exploit, database-dump. Input: URL with parameters. Output: vulnerability details, extracted data.

Nuclei integration: Nuclei registers capabilities for: template-scan, custom-check. Input: target list, template selection. Output: matched findings with severity.

Gobuster integration: Gobuster registers capabilities for: directory-brute, dns-brute, vhost-brute. Input: target, wordlist. Output: discovered paths, subdomains.

CrackMapExec integration: CME registers capabilities for: smb-enum, ldap-query, credential-spray. Input: target range, credentials. Output: access results, enumeration data.

Custom tool integration: any tool can integrate using the NTP SDK. Integration requires: capability declaration, input/output schema, and message handling. Typical integration: 50-100 lines of code.

8.5 Workflow Debugging

Step-through execution: workflows can be executed step-by-step. Each step pauses for inspection before proceeding. Step-through mode enables: debugging, training, and demonstration.

Variable inspection: at any point during execution, all variables can be inspected. Variables include: workflow parameters, step outputs, and computed values.

Breakpoints: conditional breakpoints pause execution when a condition is met. Example: pause when a critical vulnerability is found. Breakpoints enable: focused debugging.

Execution trace: the trace records every step execution with: input, output, timing, and decision points. The trace is saved for: post-execution analysis.

Replay with modification: past workflow executions can be replayed with modified inputs or steps. Replay enables: what-if analysis and regression testing.

Dry run: workflows can be validated without actually invoking tools. Dry run checks: variable availability, schema compatibility, and tool availability. No actual scanning occurs.

Workflow diff: comparing two workflow definitions shows: added steps, removed steps, changed parameters, and structural changes. Diff enables: code review for workflows.

Workflow linting: the linter checks for: unreachable steps, unused variables, missing error handlers, and schema mismatches. Linting catches errors before execution.

Workflow visualization: the visualizer renders workflows as: directed graphs (steps and data flow), Gantt charts (timing), and sequence diagrams (tool interactions).

Workflow testing: workflow tests define: expected inputs, mock tool responses, and expected outputs. Tests run without real tools. Testing validates workflow logic.

2.5 Protocol Wire Format

Header layout: bytes 0-1: protocol version (big-endian uint16). Bytes 2-3: message type (big-endian uint16). Bytes 4-7: payload length (big-endian uint32). Bytes 8-11: sequence number (big-endian uint32).

Message type values: REGISTER=0x0001, REGISTER_ACK=0x0002, DISCOVER=0x0010, DISCOVER_RESULT=0x0011, INVOKE=0x0020, RESULT=0x0021, STREAM=0x0022, ERROR=0x00F0, HEARTBEAT=0x00FF.

Sequence numbering: each connection maintains an independent sequence counter. Sequence numbers start at 1 and increment. Wraparound at 2^32 resets to 1.

Payload framing: the payload length field limits messages to 4 GB. Practical limit: 16 MB (configurable). Messages exceeding the limit are rejected with a SIZE_EXCEEDED error.

Fragmentation: messages exceeding the transport MTU are fragmented by the transport layer. The protocol layer sees complete messages. Fragmentation is transparent.

Keepalive framing: HEARTBEAT messages have zero payload length. The 12-byte header alone serves as the keepalive. This minimizes network overhead for idle connections.

Error framing: ERROR messages contain: error code (uint32), error category (string), message (string), and context (map). Error codes are: protocol-level (0x0001-0x00FF) and tool-level (0x0100+).

Streaming framing: STREAM messages include: stream_id (uint32), chunk_number (uint32), total_chunks (uint32 or 0 for unknown), and chunk_data (binary).

Batch framing: batch messages wrap multiple sub-messages. Batch header: sub-message count (uint32). Each sub-message has its own header and payload.

Compression framing: compressed payloads include: compression algorithm (uint8, 0=none, 1=zstd, 2=lz4), uncompressed length (uint32), and compressed data.

5.4 Concurrent Invocation Control

Concurrency limits: ntpd enforces per-tool concurrency limits. If a tool's limit is reached, new invocations are queued. Queue priority determines the execution order.

Semaphore-based control: each tool has a semaphore with capacity equal to its concurrency limit. Invocations acquire the semaphore before execution and release on completion.

Queue management: queued invocations are stored in a priority queue. Priority is determined by: client priority, invocation priority, and arrival time. FIFO within the same priority.

Queue timeout: queued invocations have a maximum wait time. If the wait exceeds the timeout, the invocation is rejected with a QUEUE_TIMEOUT error.

Queue monitoring: metrics include: queue depth, average wait time, timeout rate, and throughput. Queue metrics help tune concurrency limits and identify bottlenecks.

Dynamic scaling: if queue depth exceeds a threshold, ntpd can: request additional tool instances, increase concurrency limits, or notify the operator.

Fair scheduling: when multiple clients share a tool, the scheduler ensures fairness. No single client can monopolize a tool's capacity. Fair scheduling uses: weighted round-robin.

Resource reservation: clients can reserve tool capacity in advance. Reservations guarantee: minimum throughput, maximum latency, and priority access during peak load.

Concurrency negotiation: tools declare their concurrency limit during registration. ntpd can override the limit based on: available resources, current load, and administrative policy.

Deadlock prevention: ntpd detects potential deadlocks when: tool A invokes tool B and tool B invokes tool A simultaneously. Deadlocks are broken by: timeout and priority-based preemption.

6.4 Result Notification

Webhook notifications: clients register webhook URLs. When invocations complete, ntpd sends the result to the webhook. Webhook payloads include: invocation ID, status, and summary.

Email notifications: clients configure email notifications for: specific capabilities, specific statuses (failure only), or specific findings (critical severity).

Slack integration: ntpd sends notifications to Slack channels. Notifications are formatted with: finding summary, severity badge, and link to the full result.

Jira integration: findings are automatically created as Jira issues. Issue fields include: summary, description, severity, affected asset, and remediation steps.

Custom notifications: the notification system supports plugins. Plugins implement the NotificationHandler trait. Plugins receive: event type, result summary, and configuration.

Notification throttling: repeated notifications for the same finding are throttled. Throttle window: 1 hour (default). Throttling prevents: notification fatigue and inbox flooding.

Notification templates: notification content is rendered from templates. Templates support: variable substitution, conditional sections, and formatting. Templates are customizable.

Notification history: sent notifications are recorded. History includes: notification type, recipient, timestamp, and delivery status. History enables: debugging and audit.

Notification retry: failed notifications are retried with exponential backoff. Maximum retries: 3. Permanently failed notifications are logged and reported.

Notification security: webhook URLs are validated. Email addresses are verified. Slack tokens are encrypted at rest. Notification channels are authenticated.

3.5 Tool Authentication

Tool identity: each tool has a unique identity derived from: tool name, version, and instance ID. The identity is used for: access control, logging, and metrics.

Tool certificates: tools authenticate to ntpd using X.509 certificates. The tool certificate contains: tool identity, capabilities, and validity period.

Tool tokens: simpler authentication using bearer tokens. Tokens are issued by ntpd-admin. Tokens contain: tool identity, capabilities, and expiration.

Mutual authentication: ntpd and tools mutually authenticate. Tools verify ntpd's certificate. ntpd verifies the tool's certificate. Mutual authentication prevents: impersonation.

Tool secret rotation: tool credentials are rotated periodically. Rotation procedure: generate new credentials, update tool configuration, revoke old credentials.

Tool enrollment: new tools are enrolled through: admin-approved registration, certificate signing request, or automated enrollment via a trusted agent.

Tool revocation: compromised tool credentials are revoked immediately. Revoked tools are disconnected. The revocation list is distributed to all ntpd instances.

Tool attestation: tools can attest their integrity using: code signing, binary hash verification, or TPM-based attestation. Attestation ensures: the tool binary has not been tampered with.

Tool isolation credentials: each tool instance has unique credentials. Credentials are not shared between instances. Instance-level isolation enables: fine-grained access control.

Tool credential storage: credentials are stored in: environment variables (containers), keychain (desktop), or HSM (high-security). Plaintext storage is prohibited.

9.4 Protocol Monitoring Dashboard

Overview panel: active connections, registered tools, active invocations, error rate, and throughput. The overview provides: at-a-glance system health.

Connection panel: connection list with: client identity, connection duration, messages exchanged, and current state. Filtering by: client, state, and duration.

Tool panel: registered tools with: name, version, capabilities, health status, and invocation metrics. Sorting by: name, invocations, error rate.

Invocation panel: active and recent invocations with: client, tool, capability, status, duration, and result size. Filtering by: status, capability, and client.

Workflow panel: active and completed workflows with: name, status, current step, elapsed time, and step durations. Visualization of: workflow progress and timing.

Error panel: recent errors with: error code, message, source, and affected invocations. Trending: error rate over time, top error codes, and affected tools.

Performance panel: latency histograms, throughput charts, and resource usage graphs. Timeframes: last hour, last day, last week, and custom range.

Security panel: authentication failures, authorization denials, and anomaly alerts. Recent events with: client identity, action, decision, and reason.

Configuration panel: current configuration with: change history, pending changes, and configuration diffs. Configuration changes are: versioned, reviewed, and audited.

Alert panel: active alerts with: severity, source, message, and duration. Alert history with: resolution time and resolution action. Alert trends over time.

11.2 Reference Implementation Details

Language: the reference implementation is written in Lateralus. The codebase is approximately 15,000 lines of code including tests and documentation.

Architecture: async runtime using Lateralus async/await. Connection handling uses: accept loop, per-connection task, and message dispatch. The architecture is: single-binary, zero-dependency.

Testing: the test suite includes: 500+ unit tests, 100+ integration tests, and 50+ property-based tests. Coverage: 92%% line coverage, 85%% branch coverage.

Performance testing: benchmarks run on every commit. Performance regressions are detected automatically. Benchmarks cover: throughput, latency, memory usage, and connection scaling.

Fuzzing: the protocol parser is continuously fuzzed. Fuzzing has found: 12 parsing bugs, 3 denial-of-service vulnerabilities, and 1 memory safety issue (all fixed).

Security audit: the implementation has been audited by an independent security firm. Audit findings: 2 medium-severity issues (both fixed), 5 low-severity issues (all fixed).

Documentation: comprehensive documentation includes: protocol specification, API reference, deployment guide, integration guide, and contributor guide.

Release process: releases follow semantic versioning. Each release includes: changelog, migration guide, binary packages, and Docker images. Releases are signed.

Community contributions: the implementation accepts community contributions via pull requests. Contributions require: tests, documentation, and code review approval.

License: the reference implementation is licensed under the Mozilla Public License 2.0. The license allows: commercial use, modification, and distribution with file-level copyleft.

A.1 Message Type Reference

0x0001 REGISTER: Tool registration request. Payload: tool descriptor. Direction: client to ntpd. Response: REGISTER_ACK.

0x0002 REGISTER_ACK: Registration acknowledgment. Payload: tool_id, status. Direction: ntpd to client. Sent in response to REGISTER.

0x0003 DEREGISTER: Tool deregistration request. Payload: tool_id. Direction: client to ntpd. Response: DEREGISTER_ACK.

0x0010 DISCOVER: Capability discovery query. Payload: query filters. Direction: client to ntpd. Response: DISCOVER_RESULT.

0x0011 DISCOVER_RESULT: Discovery response. Payload: matching tools array. Direction: ntpd to client. Sent in response to DISCOVER.

0x0020 INVOKE: Tool invocation request. Payload: capability, input, options. Direction: client to ntpd. Response: RESULT or ERROR.

0x0021 RESULT: Invocation result. Payload: status, data, metadata. Direction: ntpd to client. Sent when invocation completes.

0x0022 STREAM: Streaming output chunk. Payload: stream_id, chunk data. Direction: ntpd to client. Sent during streaming invocations.

0x0030 CANCEL: Invocation cancellation. Payload: invocation_id. Direction: client to ntpd. Response: CANCEL_ACK.

0x00F0 ERROR: Error notification. Payload: error code, message, context. Direction: either direction. Sent on protocol or invocation errors.

0x00FF HEARTBEAT: Connection keepalive. Payload: none. Direction: either direction. Sent periodically to maintain the connection.

0x0080-0x00EF: Reserved for extensions. Extension types are registered during REGISTER_EXTENSION. Custom message handling is delegated to extension handlers.

A.2 Error Code Reference

0x0001 UNKNOWN_ERROR: An unspecified error occurred. This is the fallback code. The error message provides additional context.

0x0002 PROTOCOL_ERROR: The received message violates the protocol specification. Causes: invalid message type, malformed header, or unexpected message in the current state.

0x0003 AUTH_FAILURE: Authentication failed. Causes: invalid credentials, expired token, or revoked certificate. The client should re-authenticate.

0x0004 AUTH_REQUIRED: The operation requires authentication. The client has not yet authenticated on this connection.

0x0005 FORBIDDEN: The authenticated client lacks permission for the requested operation. The ACL does not grant access.

0x0006 NOT_FOUND: The requested capability, tool, or invocation was not found. Causes: unregistered capability, unknown invocation ID.

0x0007 TIMEOUT: The invocation exceeded the configured timeout. The tool did not respond in time. The client may retry.

0x0008 CANCELLED: The invocation was cancelled by the client or by ntpd (due to shutdown or resource constraints).

0x0009 RATE_LIMITED: The client exceeded the rate limit. The client should reduce its request rate. Retry after the specified delay.

0x000A SIZE_EXCEEDED: The message payload exceeds the maximum allowed size. The client should reduce the payload or use streaming.

0x000B TOOL_ERROR: The tool reported an error during invocation. The error details are in the context field.

0x000C QUEUE_FULL: The invocation queue is full. The client should retry later or reduce its concurrency.

A.3 Configuration Reference

[daemon] section: listen_local = '/run/nullsec/ntpd.sock', listen_tcp = '0.0.0.0:4443', max_connections = 50000, max_payload_size = '16MB'.

[tls] section: cert_file = '/etc/nullsec/tls/server.crt', key_file = '/etc/nullsec/tls/server.key', ca_file = '/etc/nullsec/tls/ca.crt', min_version = '1.3'.

[auth] section: token_secret = '/etc/nullsec/token.key', token_ttl = '24h', max_failed_attempts = 5, lockout_duration = '15m'.

[cache] section: enabled = true, max_size = '1GB', ttl = '1h', eviction = 'lru'. Cache configuration controls result caching behavior.

[storage] section: backend = 'postgresql', connection_string = 'postgresql://ntpd@localhost/ntpd', retention_days = 30.

[logging] section: level = 'info', format = 'json', output = 'file', file = '/var/log/nullsec/ntpd.log', rotation = 'daily'.

[audit] section: enabled = true, file = '/var/log/nullsec/audit.log', retention_days = 365, tamper_protection = true.

[metrics] section: enabled = true, listen = '127.0.0.1:9090', path = '/metrics'. Prometheus-compatible metrics endpoint.

[limits] section: max_concurrent_invocations = 10000, max_concurrent_workflows = 100, max_queue_depth = 50000, max_stream_size = '1GB'.

[health] section: heartbeat_interval = '15s', heartbeat_timeout = '45s', health_check_interval = '60s'. Health monitoring configuration.

[notification] section: webhook_timeout = '10s', email_from = 'ntpd@nullsec.local', slack_webhook = ''. Notification delivery settings.

[ha] section: mode = 'active-passive', peer = 'ntpd-standby.nullsec.local:4444', sync_interval = '1s'. High availability configuration.

A.4 SDK Quick Start - Lateralus

Install the NTP client library: lateralus-pkg add nullsec-ntp-client. Import: use ntp_client::{Client, Capability, Schema};

Connect and authenticate: let client = Client::connect('unix:///run/nullsec/ntpd.sock').await?; client.authenticate(Token::from_env()).await?;

Register a capability: let cap = Capability::new('my-tool', '1.0').input_schema(schema).output_schema(schema); client.register(cap).await?;

Handle invocations: client.on_invoke(|ctx, input| async move { let result = process(input).await; ctx.respond(result).await });

Discover tools: let tools = client.discover().capability('port-scan').version('>= 2.0').execute().await?;

Invoke a tool: let result =

client.invoke('port-scan').input(input).timeout(Duration::from_secs(60)).execute().await?;

Stream results: let mut stream = client.invoke('long-scan').input(input).stream().await?; while let Some(chunk) = stream.next().await { process(chunk); }

Error handling: match client.invoke('scan').execute().await { Ok(result) => handle(result), Err(NtpError::Timeout) => retry(), Err(e) => log_error(e) }

Clean shutdown: client.deregister().await?; client.disconnect().await?; Deregistration notifies ntpd to remove the tool from the registry.

Testing: use ntp_client::mock::MockNtpd; let mock = MockNtpd::new(); mock.expect_invoke('scan').returning(mock_result); let client = Client::connect_mock(mock);

Configuration: NTP_SOCKET, NTP_TOKEN, NTP_TIMEOUT environment variables configure the client. Programmatic configuration overrides environment variables.

Logging: the client library logs to the standard Lateralus logging framework. Log level is configurable: NTP_LOG_LEVEL=debug. Trace level logs all protocol messages.

A.5 SDK Quick Start - Python

Install: pip install nullsec-ntp. Import: from ntp_client import NtpClient, Capability, InvocationResult.

Connect: client = NtpClient(socket='/run/nullsec/ntpd.sock', token=os.environ['NTP_TOKEN']). Connection is established on first operation.

Register: cap = Capability(name='py-scanner', version='1.0', input_schema={...}, output_schema={...}). client.register(cap).

Discover: tools = client.discover(capability='port-scan', version='>=2.0'). Returns a list of ToolDescriptor objects.

Invoke: result = client.invoke('port-scan', input={'target': '10.0.0.0/24'}, timeout=60). Returns InvocationResult with status and data.

Async invoke: import asyncio. result = await client.ainvoke('port-scan', input={...}). Async API for use with asyncio event loops.

Stream: async for chunk in client.stream_invoke('long-scan', input={...}): process(chunk). Streaming invocation for large outputs.

Error handling: try: result = client.invoke('scan') except NtpTimeout: retry() except NtpError as e: log(e). Exception hierarchy mirrors error codes.

Context manager: with NtpClient(...) as client: result = client.invoke(...). The context manager handles connection and deregistration.

Testing: from ntp_client.mock import MockNtpd. mock = MockNtpd(). mock.register_response('scan', {'ports': [22, 80]}). client = NtpClient(mock=mock).

Type hints: the Python SDK includes type stubs for all public APIs. Type checking with mypy or pyright detects: type mismatches and missing fields.

Compatibility: the Python SDK supports Python 3.8+. Dependencies: msgpack, asyncio, ssl. Optional: aiohttp (for webhook notifications).

10.5 Protocol Comparison

NTP vs REST: NTP uses binary encoding (2-3x more compact), persistent connections (no connection overhead), and typed capabilities (schema validation). REST is simpler but higher overhead.

NTP vs gRPC: both use binary encoding and persistent connections. NTP adds: tool registration, capability discovery, and workflow orchestration. gRPC is more general-purpose.

NTP vs MQTT: MQTT is publish-subscribe only. NTP supports: request-response, streaming, and workflows. MQTT is better for: IoT and simple event distribution.

NTP vs GraphQL: GraphQL is query-oriented. NTP is action-oriented (invoke capabilities). NTP adds: tool lifecycle management and workflow orchestration.

NTP vs ZeroMQ: ZeroMQ provides raw messaging patterns. NTP adds: tool registration, capability schemas, authentication, and workflow orchestration.

NTP vs custom protocols: NTP provides a standard that eliminates per-tool protocol design. Standard benefits: interoperability, tooling, documentation, and community support.

NTP vs SSH tunnels: SSH tunnels provide encrypted transport for arbitrary protocols. NTP provides: structured communication, tool management, and workflow orchestration on top of transport.

NTP overhead vs alternatives: NTP adds 12 bytes per message header. gRPC adds approximately 5 bytes (HTTP/2 framing). REST adds approximately 200 bytes (HTTP headers). NTP is middle-ground.

NTP adoption: NTP is used by: 50+ tools in the NullSec distribution, 10+ third-party tools, and 5+ commercial security products. Adoption is growing.

NTP standardization: NTP is proposed as a standard through the NullSec Foundation. The standardization process includes: specification publication, reference implementation, and conformance testing.

8.6 Workflow Security

Workflow authentication: workflows execute with the submitter's identity. Each step inherits the submitter's permissions unless explicitly overridden.

Workflow authorization: the orchestrator checks permissions before each step. If the submitter lacks permission for a step, the workflow fails with an authorization error.

Workflow isolation: concurrent workflows are isolated from each other. Variables, results, and state are not shared between workflows unless explicitly configured.

Workflow audit: every workflow execution is audited. The audit trail includes: submitter, workflow definition, step results, and timing. Audits satisfy compliance requirements.

Workflow secrets: workflows can reference secrets (API keys, credentials) stored in the secret manager. Secrets are resolved at execution time and never logged.

Workflow scoping: workflows declare their target scope. Steps cannot operate outside the declared scope. Scope enforcement prevents: accidental scanning of unauthorized targets.

Workflow approval: high-risk workflows require approval before execution. Approvers review: target scope, tools used, and estimated impact. Approved workflows are logged.

Workflow signing: workflow definitions can be cryptographically signed. Signed workflows cannot be tampered with. The orchestrator verifies signatures before execution.

Workflow versioning: workflow definitions are versioned. Version changes are: tracked, reviewed, and audited. Rollback to previous versions is supported.

Workflow sandboxing: experimental workflows run in a sandboxed environment. The sandbox limits: network access, tool availability, and execution time. Sandbox results are isolated.

B.1 Protocol Version History

Version 1.0 (2022-01): Initial release. Supported: registration, discovery, invocation, and result exchange. Transport: Unix socket only.

Version 1.1 (2022-06): Added streaming invocations. Added invocation cancellation. Added batch messages. Bug fixes for: large payload handling and heartbeat timing.

Version 2.0 (2023-01): Breaking changes: new header format (added sequence numbers), MessagePack v2 encoding, and revised error codes. Added TLS transport.

Version 2.1 (2023-06): Added workflow orchestration. Added result caching. Added extension mechanism. Performance improvements: 2x throughput increase.

Version 2.2 (2023-09): Added protocol extension mechanism. Added built-in extensions: NTP-VULN, NTP-CRED, NTP-NET. Bug fixes for workflow state persistence.

Version 3.0 (2024-01): Breaking changes: authentication overhaul (mutual TLS, token scoping), ACL model redesign, and result storage API. Added high availability support.

Version 3.1 (2024-06): Added notification system (webhooks, Slack, Jira). Added workflow debugging (breakpoints, step-through). Added SDK code generation.

Version 3.2 (2024-09): Added result correlation engine. Added compliance mapping. Added geographic scaling. Performance improvements: 50%% latency reduction.

Migration guides: each major version includes a migration guide. Guides cover: breaking changes, configuration updates, SDK updates, and testing procedures.

Deprecation schedule: deprecated features are supported for 12 months after deprecation notice. Deprecated features emit warnings. Removal occurs in the next major version.

Compatibility matrix: the documentation includes a compatibility matrix showing: protocol version, SDK version, ntpd version, and supported features.

Future roadmap: planned features for version 4.0 include: WebAssembly tool plugins, distributed workflow execution, AI-assisted tool selection, and zero-trust architecture.

B.2 Glossary of Terms

Capability: a named function that a tool provides. Capabilities have: name, version, input schema, and output schema. Tools register capabilities with ntpd.

Invocation: a request to execute a capability on a specific tool. Invocations have: ID, input, options, and result. ntpd routes invocations to the appropriate tool.

ntpd: the NullSec Tool Protocol daemon. The central service that manages: tool registrations, capability discovery, invocation routing, and workflow orchestration.

Workflow: a sequence of tool invocations with data flow between steps. Workflows automate: multi-step assessments, reporting, and remediation tracking.

Schema: a JSON Schema definition describing the structure of inputs and outputs. Schemas enable: validation, documentation, and code generation.

Tool: a program that implements one or more capabilities and communicates with ntpd using the NTP protocol. Tools register on startup and handle invocations.

Registration: the process of a tool declaring its capabilities to ntpd. Registration enables: discovery, routing, and lifecycle management.

Discovery: the process of querying ntpd for tools that match specific criteria. Discovery enables: dynamic tool selection and capability-based programming.

Stream: a sequence of partial results produced during a long-running invocation. Streams enable: real-time progress reporting and incremental result processing.

Extension: a protocol add-on that provides additional message types and functionality. Extensions are registered with ntpd and negotiated per-connection.

Correlation: the process of linking related findings from different tools or time periods. Correlation improves: confidence scoring, deduplication, and attack path analysis.

Circuit breaker: a fault tolerance pattern that prevents repeated invocations to failing tools. The breaker opens after threshold failures and closes after cooldown.

References

[1] Fielding, R. Architectural Styles and the Design of Network-based Software Architectures. 2000.

[2] Rescorla, E. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, 2018.

[3] Furuhashi, S. MessagePack Specification. msgpack.org, 2013.

[4] Bernstein, D. J. CurveCP: Usable security for the Internet. 2011.

[5] NullSec Project. NullSec Security Distribution Architecture Guide. 2024.