NATS Server

Security Assessment

February 27, 2025

Prepared for:
Ginger Collison, Synadia Communications, Inc.
Organized by Open Source Technology Improvement Fund, Inc.

Prepared by: **Spencer Michaels, Sam Alws, Emilio López, Cliff Smith, and Travis Peters**

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

Jeff Braswell, Project Manager
jeff.braswell@trailofbits.com

The following engineering director was associated with this project:

David Pokora, Engineering Director, Application Security
david.pokora@trailofbits.com

The following consultants were associated with this project:

Sam Alws, Consultant
sam.alws@trailofbits.com

Emilio López, Consultant
emilio.lopez@trailofbits.com

Spencer Michaels, Consultant
spencer.michaels@trailofbits.com

Cliff Smith, Consultant
cliff.smith@trailofbits.com

Travis Peters, Consultant
travis.peters@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
March 18, 2024	Pre-project kickoff call
March 25, 2024	Status update meeting #1
April 1, 2024	Delivery of report draft
April 1, 2024	Report readout meeting
February 27, 2025	Delivery of final comprehensive report
April 17, 2025	Delivery of updated final comprehensive report

Executive Summary

Engagement Overview

The Open Source Technology Improvement Fund (OSTIF) engaged Trail of Bits to review the security of the NATS Server project, which provides messaging for distributed systems.

A team of five consultants conducted the review from March 18 to March 29, 2024, for a total of six engineer-weeks of effort. Our testing efforts focused on security concerns related to account isolation for multitenant configurations; authentication and authorization in both static and dynamic modes, including callouts; TLS support in intercomponent connections; encryption at rest for Stream data; and input parsing. With full access to source code and documentation, we performed static and dynamic testing of the codebase, using automated and manual processes.

Observations and Impact

Within the scope of security concerns reviewed in this audit, the NATS Server has an overall satisfactory security posture. While we found 10 security issues during the review, the majority are of informational severity—that is, not exploitable or of negligible impact—and the remaining issues are of medium or low severity. Notably, the issues described in this report do not appear to indicate systemic flaws in the NATS team's development process; rather, they are more likely to be the result of isolated mistakes. Such occasional mistakes are inevitable in a sufficiently large codebase, but their likelihood can be reduced by lessening code complexity and by routinely employing static analysis.

Complexity is a notable weak point of the NATS codebase: files for major functionality such as authentication can be thousands of lines long, the functions within handling multiple distinct features across many hundreds of lines. Even if this kind of code is not vulnerable in its current state, there is a risk that developers making future changes could have difficulty fully understanding it and introduce logic errors. We recommend refactoring these parts of the codebase to split up long functions and large files to make the individual blocks of logic contained within them easier to grasp.

We also recommend using static analyzers such as Semgrep and `errcheck` to identify issues such as unchecked errors and type assertions, as our own scans with those tools found a large number of these issues. In a sufficiently large codebase, the introduction of such flaws over time is inevitable simply due to human error, but it is fairly easy to detect them using static analysis at build time or before pushing a commit in order to prevent them from making their way into production code.

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that the NATS development team take the following steps:

- **Remediate the findings disclosed in this report.** These findings should be addressed as part of a direct remediation or any refactor that may occur when addressing other recommendations.
- **Investigate and remediate instances of unchecked type assertions and ignored error values.** See appendix D for a list of unchecked type assertions. A list of ignored error values can be obtained using the `errcheck` tool.
- **Reduce the size and complexity of security-related functions, such as authentication handlers.** Conditional-heavy functions that are many hundreds of lines long are difficult to understand and audit, making it easier to accidentally introduce logic errors while making future changes and harder to detect those issues after the fact. If a function handles several distinct concerns, such as different types of authentication, refactor it into multiple self-contained, single-purpose functions and call out to the necessary handler as appropriate for each incoming authentication request.
- **Regularly perform static analysis on the codebase using Semgrep, CodeQL, and actionlint and integrate these tools into the CI pipeline.** See appendix F for further instructions.

Finding Severities and Categories

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

Severity	Count
High	0
Medium	3
Low	1
Informational	6
Undetermined	0

CATEGORY BREAKDOWN

Category	Count
Configuration	2
Data Exposure	1
Data Validation	1
Denial of Service	2
Error Reporting	1
Patching	1
Timing	1
Undefined Behavior	1

Project Goals

The engagement was scoped to provide a security assessment of the NATS system. Specifically, we sought to answer the following non-exhaustive list of questions:

- Are there circumstances under which potentially sensitive data such as message contents could be leaked from one account to another in a multitenant environment?
- Are there any cases in which access controls could behave differently depending on whether an account is statically or dynamically configured, sourced from an authentication callback, and so on?
- Is TLS properly configured on client, Leaf Node, route, and Gateway connections?
- Is Stream data encrypted at rest, and are the encryption methods in use reasonably secure?
- Can any untrusted user input trigger a panic, crash a server, or cause a connection between cluster members to be dropped?

Project Targets

The engagement involved a review and testing of the targets listed below.

NATS Server

Repository	https://github.com/nats-io/nats-server
Version	`121169ea86756a853a418446b9c7591df761b49d` (tag v2.10.12)
Type	Go application
Platform	Multiple

NATS JWT Implementation

Repository	https://github.com/nats-io/jwt
Version	`c2d30e2ffc632a1ea64030467e5a40e02e4158be` (tag v2.5.5)
Type	Go library
Platform	Multiple

NATS Keys

Repository	https://github.com/nats-io/nkeys
Version	`c865baf4058b0ae6529eeb82fbe86bd8c21f4a36` (tag v0.4.7)
Type	Go library
Platform	Multiple

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches included the following:

- Manual review of critical code paths related to the following features:
    - Multitenancy
    - Authentication and authorization
    - TLS support
    - Encryption at rest
    - Parser defined in `server/parser.go`
- Use of Semgrep, CodeQL, and actionlint static analysis tools on the `nats-server` repository and triaging of results
- Dynamic testing and analysis of application features related to the above features, where feasible

Threat Model

As part of the audit, Trail of Bits conducted a lightweight threat model, drawing from Mozilla's "Rapid Risk Assessment" methodology and the National Institute of Standards and Technology's (NIST) guidance on data-centric threat modeling (NIST 800-154). We began our assessment of the design of NATS Server by reviewing published system documentation. During the week of February 20, 2024, we held a series of three discovery calls with NATS engineers to analyze the system's internal design and learn about typical use cases.

Threat Model Scope

This threat model covered the open-source NATS Server software project, documented at https://docs.nats.io/ and hosted at https://github.com/nats-io/nats-server/. The commercial Synadia Cloud–hosted NATS service offering and the NATS Execution Engine add-on were not included in the scope.

Data Types

NATS clients and Servers communicate using a custom protocol consisting of text-based messages, transmitted over TCP, which may or may not be encrypted. Configuration files are written in a custom format that resembles markup languages such as JSON and YAML. In some configurations, authentication messages consisting of JWTs are exchanged between clients, Servers, and external authentication providers.

The fundamental architecture of NATS is based on the publish-subscribe pattern: each message has a designated subject that determines its delivery destinations. All operations, including Cluster and Supercluster management and the JetStream persistence layer, are built on top of this model.

Data Flow

Below, we depict known connections between system components of a hypothetical NATS deployment. Developers and system administrators have great freedom in designing a system running on NATS, and there can be substantial variations between architectures. The deployment described herein is a basic installation that includes at least one instance of each of the major components of the optional NATS Server components. Further details are discussed in the Components and Trust Zones and Trust Zone Connections report subsections.

In the diagrams below, the dotted red lines indicate trust boundaries separating zones, where the system enforces (or should enforce) interstitial controls and access policies. Where additional message-specific authentication or authorization information is needed, we label the connection with it.

A NATS system (figure 1) consists of one or more Servers, to which any number of clients may connect. Typically, such a system will feature multiple Servers joined together into a mesh network called a Cluster. Servers in a Cluster communicate with each other via Routes. Multiple Clusters can be further aggregated into a Supercluster by designating one or more Servers within each Cluster as a Gateway, providing an inter-Cluster communication channel.

In addition to requiring that clients authenticate upon connecting to a Server, a NATS deployment can create security domains in the form of Accounts (figure 2), which act as namespaces of subjects. Servers group individual users into Accounts, and each user can access only subjects within their own Account. Administrators can selectively loosen this boundary by configuring Imports and Exports that expose specified resources in one Account to users in a different Account. Additionally, user-level ACLs can also exercise fine-grained control over subject authorizations through allowlists and denylists.

Authentication in NATS supports configurations involving an external source of truth. If an external Account Resolver is in use (figure 4), an external database provides the root of trust for Accounts. If an auth callout service is in use (figure 3), both authentication and authorization decisions rely on logic implemented outside of the NATS Cluster.

When a client attempts to authenticate through an auth callout service, the service handling the client's initial connection request publishes an authentication request through a system service. That request is picked up by another client that has network access to an arbitrary external service that provides authentication. The handling client then publishes an authentication response, and the message is received by the original server's callout service.

For other account resolvers, users can be authenticated against multiple data sources. A Server may have a list of accounts directly in memory, loaded from a local configuration file; it may call out to another server using NATS native account resolution; or it may call out to an external resolver URL.

The Account concept can be extended to create a security boundary between two Servers through the use of Leaf Nodes, which are Servers configured to run in an auxiliary or satellite relationship to a Cluster. When a Leaf Node authenticates to a Cluster through a Server's Leaf Node Listener, the Leaf Node's access is limited to one Account. As with normal users, the Server can also apply ACLs to further limit the Leaf Node's access to the chosen Account. Clients connecting to the Leaf Node authenticate according to the Leaf Node's local policy, but no client connecting through a Leaf Node can obtain greater access to the Cluster than the Leaf Node's own credentials permit.

Clients may connect to a NATS Cluster using a variety of protocols, including the NATS native text-based protocol, WebSockets, and MQTT. Clients using only Core NATS features have access to ephemeral publish-subscribe messaging, under which messages are never queued for delivery to offline clients or stored for later retrieval. A subsystem called JetStream provides a flexible persistence layer that enables administrators to configure storage and delayed delivery of messages according to an administrator-defined policy.

The NATS service itself is a single binary; the distinctions between Servers running in various capacities (Leaf Node, Gateway, etc.) are determined by the features that are enabled on each instance of the NATS service.

NATS Servers may expose Monitor and Profiler endpoints that provide performance and debugging information over an HTTP connection.

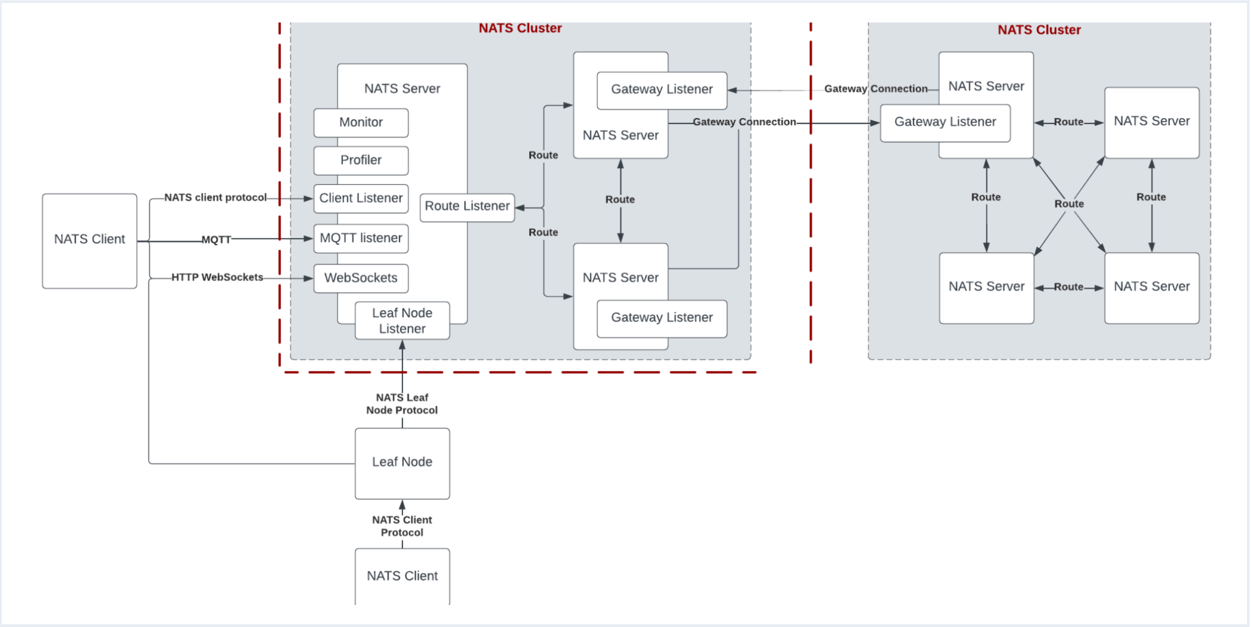

Figure 1: Architecture for a hypothetical NATS Server Supercluster, including two Clusters, three Gateways, and one Leaf Node. All supported protocols are represented.

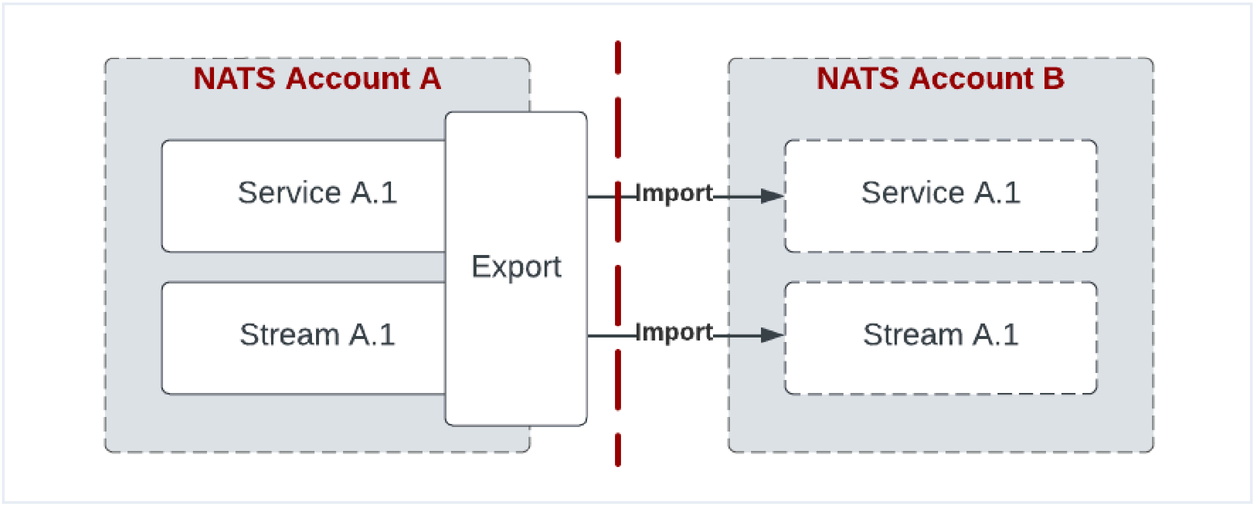

Figure 2: Accounts as a logical trust zone as they exist throughout a Supercluster. Services and Streams in Account A are selectively made available to other accounts through an Export. Account B enables its clients to interact with these resources by importing them.

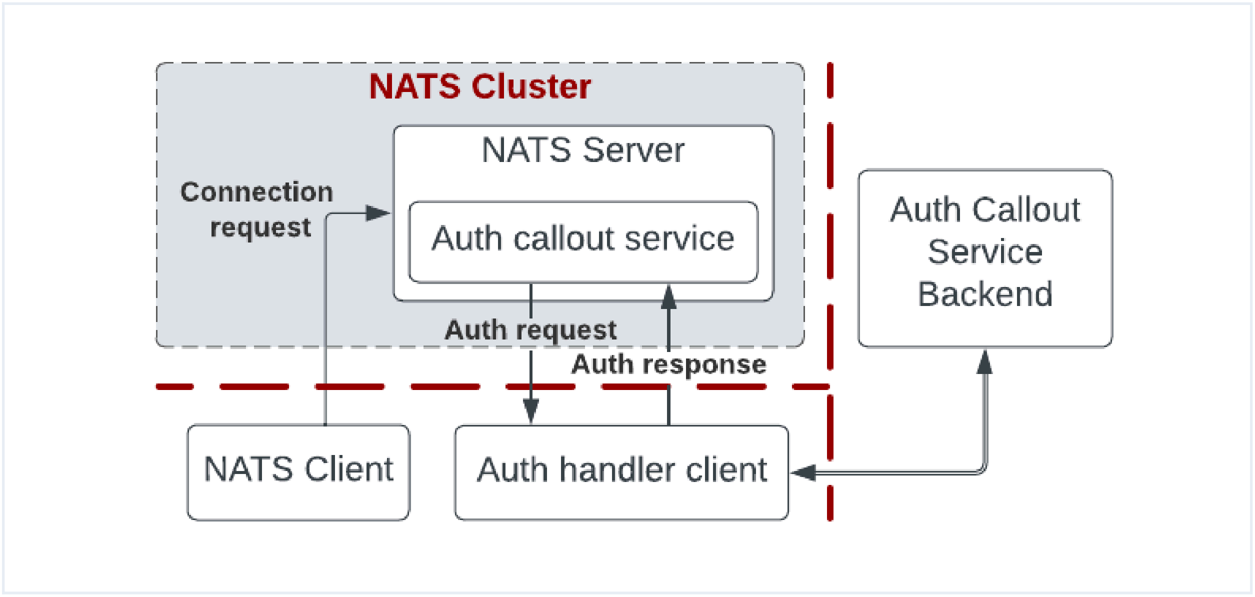

Figure 3: A NATS client authenticating via an external callout service

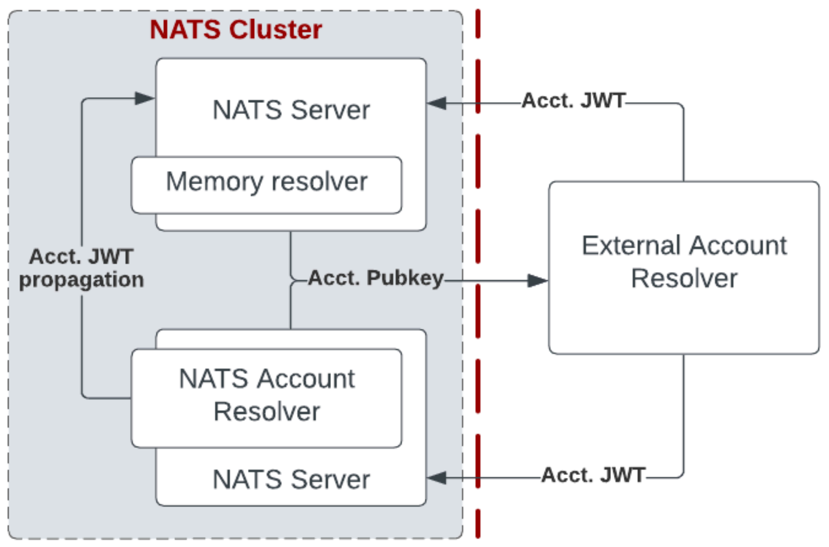

Figure 4: The various ways in which a NATS Cluster can be configured to resolve accounts

Components and Trust Zones

The following table describes the components that make up the NATS system, as well as the external dependencies on which they rely. These system elements are further classified into *trust zones*—logical clusters of shared functionality and criticality, between which the system enforces (or should enforce) interstitial controls and access policies.

Component	Description
External Clients	External clients interact with a NATS Cluster, including publishing and subscribing to message subjects, without joining a Cluster as Servers.
Online Clients	Some NATS clients, including the NATS CLI and client SDK frameworks, are used to interact with a live Cluster, including sending and receiving messages and monitoring the Cluster's performance. Additionally, some trusted users will connect to the monitoring and profiling endpoints using web browsers.
Offline Utilities	The NATS project publishes utility software, including nsc and nk, that generate keys and other configuration data offline instead of interacting with a live Cluster over the network.
Cluster	A Cluster is a group of NATS Servers that replicate data across a full mesh of route connections.

NATS Server	A Server is a node within a NATS mesh network; its core function is that of a message broker, receiving and distributing arbitrary payloads called messages that are categorized under textual tags called subjects. In addition to its TCP listener for client connections under the standard NATS protocol, a Server may expose various additional listeners for Server-to-Server communication purposes depending on its role within the system.
Client Listener	Each Server runs a TCP listener that accepts connections using the native text-based publish-subscribe protocol.
Leaf Node Listener	A Leaf Node Listener is a Server listener that accepts incoming connections from Leaf Nodes and exchanges messages between Leaf Nodes and the Server nodes in the Cluster.
Route Listener	A Route Listener is a Server component that accepts incoming connections from other Servers in the same Cluster, establishing a mesh.
Gateway Listener	A Gateway Listener is a Server component that accepts incoming connections from Servers in other Clusters operating as Gateways. Such cross-Cluster connections establish a Supercluster.
Monitor	A Monitor is a Server component that exposes Server information and statistics over HTTPS.
Profiler	A Profiler is a Server component that exposes a `pprof` endpoint for performance profiling.
Stream	The JetStream subsystem is a persistence layer that runs on top of Core NATS, and a Stream is a persistent representation of the message history for a subject. The contents of a Stream are exposed to end users through consumers, which provide a view of the Stream contents that client applications can subscribe to on a push or pull basis. Streams can be exposed to other Accounts through Exports.
Service	A Service is an entity that follows the request-reply pattern; clients can send the Service a message representing an action or a query and then await a reply. Services can be exposed to other Accounts through Exports.
MQTT Bridge	An MQTT Bridge is a component that enables MQTT devices to communicate with NATS Servers, bridging MQTT and NATS topics.

Supercluster	A Supercluster is an interconnected group of Clusters that share data over Gateway connections.
Gateway	A NATS Server running as a Gateway exposes, in addition to the normal Client Listener and any other enabled network services, a TCP service that accepts incoming connections from Gateway nodes in other Clusters. Gateway nodes communicate about their respective interest in different subjects, and when a Gateway node receives a message in which another Cluster has interest, it replicates the message to the other Cluster via its Gateway connection. Each Gateway connection is one way. When two Clusters connect across Gateways, each Cluster must have at least one outbound Gateway connection to the peer Cluster.
Leaf Node	A Leaf Node establishes a restricted connection to a Cluster and is responsible for authenticating and authorizing clients, routing client messages, and applying ACLs.
Leaf Node	When a Leaf Node opens an authenticated connection to a Server's Leaf Node Listener, it gains access to the Cluster's data Account associated with the Leaf Node's user credentials, subject to any ACLs the Server applies to that user. Online Clients can connect to the Leaf Node and publish or subscribe to subjects in the same Account. Leaf Nodes can also maintain JetStream persistence throughout temporary outages in the connection to the Cluster. In configurations with multiple disjoint Clusters or Superclusters, Leaf Nodes can simultaneously open connections to multiple upstream Servers using either the same Account or different Accounts.
Account	An Account is a namespace of resources, including subjects, messages, and Streams. Creating multiple Accounts in one Cluster or Supercluster enables system administrators to implement multitenancy by restricting user access to resources appropriate for that user's business role.
System Account	The System Account is a default Account that does not contain user data. Depending on the system's configuration, Servers communicate with each other using Services and subjects in the System Account. Additionally, messages comprising logging events and Cluster statistics are posted to subjects within the System Account. Clients with credentials for the System Account can read this data for monitoring purposes and, depending on their authorization, may be able to alter system configuration by posting messages to these subjects.

Non-System Account	Non-System Accounts are user-space Accounts that contain subjects, messages, and other resources that a client can interact with. If administrators do not manually configure a Non-System Account, NATS places all user data in a default Account called the global Account.
Export	A Server configuration can permit access to Services and Streams from outside their containing Accounts by defining an Export. Each Export can optionally be restricted so that only specified peer Accounts are permitted to import from it.
Import	A Server configuration can permit users in one Account to access exported Services and Streams by defining an Import. Only Services and Streams included within an Export can be imported, and the Server will enforce any Account restrictions specified in the Export.
External Network	Under some configurations, aspects of authentication and authorization can be delegated to external systems via a URL that could be hosted anywhere on the internet that is accessible from the NATS Servers.
Credential Minter	In JWT-based authentication configurations, external processes in possession of the operator and/or Account keys are responsible for generating authentication keys and JWTs. User JWTs are self-proving through their signatures and need not be maintained inside the NATS system's data store.
Account Resolver	When a Cluster is configured for decentralized JWT authentication, a Server can optionally retrieve JWTs defining an Account's root of trust from an external server identified by a URL specified in the Server's configuration.
Auth Callout Service Backend	The auth callout feature allows a Cluster to fully delegate authentication and authorization decisions to a NATS Service. In most use cases, this Service will interact with some sort of backend service, such as an LDAP service.

Trust Zone Connections

At a design level, trust zones are delineated by the security controls that enforce the differing levels of trust within each zone. Therefore, it is necessary to ensure that data cannot move between trust zones without first satisfying the intended trust requirements of its destination. We enumerate such connections between trust zones below.

Originating Zone	**Destination Zone**	**Data Description**	**Connection Type**	**Authentication Type**
External	Cluster	An Online Client can connect to any server running a reachable Client Listener, MQTT listener, or WebSockets listener. After authenticating, the client can publish or subscribe to any message subjects in their Account, subject to any applicable ACLs.	• NATS client protocol • MQTT • WebSockets	• Server-side TLS • Username and password • Static token • Mutual TLS • NKey challenges • Bearer token
	Cluster	When monitoring and profiling are enabled, clients can also use web browsers to connect to these endpoints on individual NATS Servers. Additional transport layer authentication and encryption may be added by the NATS environment's network configuration.	• Plaintext HTTP (monitoring and profiling) • HTTPS (monitoring)	• N/A
Cluster	Same Cluster	Communications between Servers in the same Cluster occur over route	• NATS route protocol	• Mutual TLS • Username and

		connections, which form a full mesh between each pair of peers in the Cluster.		password
	Different Cluster	Communications across Clusters occur over connections between Servers running as Gateway nodes. Gateway connections form a full mesh between Clusters in that there is always at least one connection between each pair of peer Clusters. Non-Gateway nodes communicate only with other Servers in the same Cluster.	● NATS Gateway protocol	● Mutual TLS
Cluster	External Network	When an external Account Resolver is in use, the NATS Server handling an authentication request retrieves the issuing Account's JWT from the Account Resolver.	● HTTPS	● Server-side TLS ● Mutual TLS
	External Network	When auth callout is configured, the Service that performs the callout will usually interact with an Auth Callout Service Backend, such as an LDAP server. The parameters of this connection are entirely up to the	● User determined	● Chosen by developer ● Request and response payloads optionally encrypted with XKeys

		developer of the auth callout service.		
Leaf Node	Cluster	When a Leaf Node connects to a Server's Leaf Node Listener, the Server limits the Leaf Node's access to Cluster data according to the authentication mechanism by which the Leaf Node authenticates itself. These limitations can include both account restrictions and fine-grained ACLs that control access to specific subjects.	• NATS Leaf Node protocol	• Server-side TLS • Username and password • Mutual TLS • NKey challenges
Account	Different Account	When Server configuration files in a Cluster define an Export covering one or more Services or Streams, other Accounts can import those Services and Streams so they are exposed to users in the importing Account.	• NATS client protocol	• N/A
System Account	Non-System Account	Messages posted to the System Account can affect data and operations in a Non-System Account through control messages posted to System Account subjects. Such messages can affect	• NATS route protocol • NATS Gateway protocol	• Mutual TLS • Username and password

		authentication processes, Account JWT signing keys, and JetStream replication.		

Threat Actors

When conducting a threat model, we define the types of actors that could threaten the security of the system. We also define other users of the system who may be impacted by, or induced to undertake, an attack. For example, in a confused deputy attack such as cross-site request forgery, a normal user who is induced by a third party to take a malicious action against the system would be both the victim and the direct attacker. Establishing the types of actors that could threaten the system is useful in determining which protections, if any, are necessary to mitigate or remediate vulnerabilities. We will refer to these actors in descriptions of the security findings that we uncovered through the threat modeling exercise.

Actor	Description
Cluster User	Users are principals authorized to directly interact with a NATS Cluster or Supercluster via an Online Client. They belong to and are issued by a specific Account and can publish or subscribe only to subjects in the containing Account, subject to any applicable ACLs.
Leaf Node User	Instead of authenticating directly into a Cluster, Leaf Node users authenticate to a Leaf Node, and their access to the Cluster is limited both by restrictions placed on the Leaf Node by the Cluster and by restrictions placed on the Leaf Node user by the Leaf Node's local authorization configuration.
Network Attacker	A network attacker is a malicious actor on an internal network containing at least one of the Server nodes within a NATS Cluster, but who has no credentials for any NATS hosts.
External Attacker	An external attacker is a malicious actor on the public internet, with no special privileges anywhere within the NATS system.

Threat Scenarios

The following table describes possible threat scenarios given the design, architecture, and risk profile of a NATS deployment.

Threat Scenario	Actor(s)	Component(s)
An attacker joins a malicious node into a Cluster via a Route Listener, such as by breaching a network perimeter and gaining access to Server nodes operating on a trusted network or by stealing authentication credentials for a Server's Route Listener. Once a malicious Server is part of a Cluster, it has full read and write access to all subjects in all Accounts, including the System Account. The presence of a malicious Server in a Cluster would amount to a total compromise of the entire NATS deployment.	● External attacker ● Network attacker	● Route Listener
An attacker joins a malicious node into a Supercluster by stealing or forging TLS authentication credentials for a Gateway Listener. Once joined into a Supercluster, there is no trust boundary between two Clusters. Therefore, a malicious Server that gained membership into a Supercluster would have achieved total compromise of the entire NATS deployment.	● External attacker	● Gateway Listener
An attacker compromises a Leaf Node. The attacker could compromise credentials for a Server's Leaf Node Listener and join a malicious Leaf Node into the Cluster, or the attacker could compromise the host running a Leaf Node that is already connected to the Cluster. Depending on the Server's configuration, the access gained may be limited to one Account and could have further limitations imposed through ACLs embedded in the compromised credentials. The attacker would have full read and write access to all subjects within the scope of the Leaf Node's access to the Cluster.	● External attacker	● Leaf Node Listener

An attacker connects to a Leaf Node using credentials that are valid for the Leaf Node's local authentication configuration. In addition to being limited by the configuration of the Leaf Node's connection to the Cluster, the attacker's access would be limited by any access controls implemented in the Leaf Node's own local authentication configuration.	• Leaf Node user	• Client Listener on Leaf Node
An unauthorized actor gains read and/or write access to a Non-System Account. Access gained through this attack will depend on the credentials used and the system's authorization configuration. If the Online Client's credentials are based on a JWT with limited authorization claims, the attacker's access may be limited to certain subjects. Any Exports accessible to the attacker's credentials could be leveraged to gain access to resources hosted in different Accounts.	• Cluster user • Leaf Node user • External attacker	• Client Listener • Export/Import
A malicious user gains access to the System Account through misconfiguration of a low-privilege user or theft of credentials for a high-privilege user. Since the System Account houses subjects that NATS Servers use to manage authentication, system life cycle events, and JetStream replication, write access to the System Account enables total compromise of the entire NATS environment. By default, users authenticating through the client listener cannot gain access to the System Account. Manual configuration is required to permit any entity aside from a Server authenticated to the same Cluster to directly publish or subscribe to System Account subjects.	• Cluster user • Leaf Node user • External attacker	• Client Listener
An attacker compromises the Credential Minter, either by stealing its NKey or by obtaining the ability to execute commands on the container or Server housing the JWT creation process. The attacker could use the compromised private key to sign arbitrary JWTs, effectively enabling compromise of all resources downstream of that private key. If the affected Credential Minter housed one Non-System Account's signing key, the compromise would be	• External attacker • Network attacker	• Credential Minter

limited in scope to that Account. If an operator key (i.e., a root trust key for the entire NATS environment) were affected, the attacker could create new Accounts and create new users for all Non-System Accounts, leading to a breach of the entire system.		
An attacker executes a cross-site scripting attack against a browser that contains a JWT used as a bearer token. By stealing this token, the attacker could connect to any Server's WebSockets listener and gain access to all subjects for which the underlying user has authorization. By default, user JWTs cannot be used as a sole means of authentication as bearer tokens, and user JWTs are not stored in cookies. Thus, multiple configuration settings must be manually changed in order for this attack to be possible.	● External attacker ● Cluster user ● Leaf Node user	● WebSockets listener
An attacker requests Account token data from an external Account Resolver. These tokens will enable the attacker to profile the user base and authentication configuration of the Cluster in question.	● External attacker	● Account Resolver
An attacker gains access to the Monitor and/or Profiler endpoints on one or more Servers. Although information obtainable through these endpoints does not contain message payloads, they will at least contain subject names along with usage data sufficient to broadly fingerprint Cluster traffic.	● Network attacker ● External attacker	● Monitor ● Profiler

Recommendations

Trail of Bits recommends that the NATS team take the following steps to improve the system's overall security:

- **Lock down the profiling and monitoring endpoints.** Currently, they default to being exposed on all interfaces and are served over plaintext HTTP. These endpoints should be exposed only on `localhost` by default and/or should support HTTPS.

- **Make TLS-first handshakes standard behavior for Leaf Nodes.** The recently added `handshake_first` option should be enabled by default.

- **Deprecate weaker authentication modes for Leaf Node connections,** including bearer token and username/password authentication, in favor of mutual TLS or NKey challenge authentication. If these need to be retained for backward compatibility reasons, consider disabling them by default and requiring the user to explicitly enable "legacy auth" through a specific flag.

- **Disallow mixed TLS and non-TLS endpoints in WebSocket configurations.** For instance, a WebSocket endpoint configured with TLS encryption (wss://) should not be able to feature an http://-scheme origin in its `allowed_origins` list.

- **Ensure that all security controls supported in the Server software are documented at the time of implementation.** For example, NATS engineers mentioned to us that the use of user JWTs as bearer tokens can be disallowed at the account level, and this feature appears to have been implemented in PR #3127. However, we did not see any mention of this feature during our review of the documentation, so most users are likely unaware of how to take advantage of it.

- **Add support for client-side authentication for outbound connections to URL-based external Account Resolvers using NKeys or an API token.** The Account Resolver protocol permits unauthenticated users to download Account tokens, which enable some fingerprinting of the NATS environment. Adding client-side authentication to these communications will help protect this data in environments where mutual TLS authentication is difficult to implement.

- **Consider implementing an end-to-end testing utility that allows administrators to quickly gauge what subjects a user has read and write access to when entering the Cluster through direct Online Client–to–Server connections and/or through Leaf Nodes.** NATS is an abstract platform that supports a variety of architectures and topologies, and sophisticated deployments may grow to a point where it is difficult to ascertain how much access a user might have in a particular context. If NATS published a software tool or example scripts that help perform that testing at scale, it would help developers validate and debug access issues in complex environments.

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	The NATS Server functionality that we considered during this audit does not feature security-relevant arithmetic operations.	**Not Applicable**
Auditing	The NATS Server consistently logs critical actions, errors, warnings, and so on.	**Satisfactory**
Authentication / Access Controls	We did not identify any issues in NATS's access controls or authentication functionality. However, NATS's code concerning these areas is quite complex, and the range of possible authentication cases relatively large, making it difficult to audit exhaustively; therefore, the maintainability and auditability of these parts of the codebase would benefit from refactoring.	**Satisfactory**
Complexity Management	The NATS codebase has a relatively flat structure, with some code files reaching thousands of lines in length. Functions for some critical functionality are extremely long (reaching many hundreds of lines) and cover numerous distinct cases that should be split into separate functions.	**Moderate**
Configuration	Due to time and scope constraints, configuration of third party components was not considered during this audit, aside from triaging Semgrep results relating to it (see TOB-NATS-7, TOB-NATS-9, and TOB-NATS-10).	**Not Considered**
Cryptography and Key Management	The NATS Server employs appropriate TLS configurations for connections between all of its component types (Gateway, client, Leaf Node, etc.). We discovered one issue involving a non–constant time operation used to compare passwords (TOB-NATS-3), but its presence does	**Satisfactory**

	not appear to indicate a systemic trend. In addition, we found no problems with NATS's implementation of encryption-at-rest, aside from an informational-level issue related to error handling (TOB-NATS-1).	
Data Handling	Within the limited focus areas of the audit, incoming data to the NATS Server appears to be handled safely and appropriately. We found one inconsistency (TOB-NATS-5) in the parser defined in `server/parser.go`, which could lead to a potential, but unlikely, dropped connection; the presence of this bug does not appear to indicate a systemic trend.	**Satisfactory**
Documentation	The NATS documentation is comprehensive and regularly updated.	**Strong**
Maintenance	Due to time and scope constraints, patching mechanisms were not considered during this audit.	**Not Considered**
Memory Safety and Error Handling	NATS includes many instances of ignored error values and unchecked type assertions. See appendix D for a list of unchecked type assertions; there are 214 of them. There are around 3,600 instances of ignored error values, which would be too large to fit into an appendix, but this list can be obtained using the `errcheck` tool. In one issue, we found that ignored error values could lead to faulty or missing encryption at rest, improper data erasure, or a null pointer dereference panic (see TOB-NATS-1).	**Moderate**
Testing and Verification	Due to time and scope constraints, test coverage was not considered during this audit.	**Not Considered**

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Ignored error values during file store operations	Error Reporting	**Informational**
2	User and NKeyUser clone() methods can fail to deep copy an empty allowed connection types list	Undefined Behavior	**Informational**
3	Non–constant time comparison of plaintext passwords	Data Exposure	**Medium**
4	Risk of denial of service when restoring Streams	Denial of Service	**Medium**
5	Inconsistent behavior around \r character in parser	Data Validation	**Informational**
6	Use of unpinned third-party workflow	Patching	**Medium**
7	Use of non-TLS download in Travis CI configuration file	Configuration	**Informational**
8	Missing mutex unlocks before return statements	Timing	**Informational**
9	Windows DLL loading susceptible to DLL hijacking attacks	Configuration	**Informational**
10	HTTP servers are vulnerable to Slowloris denial-of-service attacks	Denial of Service	**Low**

Detailed Findings

1. Ignored error values during file store operations	
Severity: **Informational**	Difficulty: **High**
Type: Error Reporting	Finding ID: TOB-NATS-1
Target: `server/filestore.go`	

Description

There are multiple instances in `filestore.go` in which error values are ignored, which could lead to the following issues:

- Faulty or missing encryption-at-rest
- Improper data erasure
- A null pointer dereference panic

This issue is rated as informational severity because it is highly unlikely for an error to happen in the relevant code paths.

In the `genEncryptionKey` function, the error value returned by `aes.NewCipher` is put into the `e` variable, which is then checked, but the `err` variable is returned instead (see figure 1.1). This means that, in the case of an error, `nil, nil` will be returned. Functions that call `genEncryptionKey` would assume that it returned successfully and then use the first returned `nil` value as a `cipher.AEAD` interface. This can lead to a null pointer dereference panic if the `Seal` or `Open` methods are called on the first `nil` return value. More importantly, if the first `nil` return value is assigned into the `fs.aek` property, then encryption-at-rest would not be used, and the user would have no indication of that fact.

```
func genEncryptionKey(sc StoreCipher, seed []byte) (ek cipher.AEAD, err error) {
	if sc == ChaCha {
		ek, err = chacha20poly1305.NewX(seed)
	} else if sc == AES {
		block, e := aes.NewCipher(seed)
		if e != nil {
			return nil, err
		}
		ek, err = cipher.NewGCMWithNonceSize(block, block.BlockSize())
	} else {
		err = errUnknownCipher
	}
	return ek, err
}
```

Figure 1.1: The `genEncryptionKey` function (`nats-server/server/filestore.go:638–651`)

In multiple places, the error value returned by the `rand.Read` function is ignored when a nonce for encryption is generated (see figure 1.2). This can cause the same nonce to be reused multiple times, potentially allowing the encrypted data to be decrypted by an adversary.

```
	// Generate our nonce. Use same buffer to hold encrypted seed.
	nonce := make([]byte, kek.NonceSize(), kek.NonceSize()+len(seed)+kek.Overhead())
	rand.Read(nonce)

	bek, err = genBlockEncryptionKey(sc, seed[:], nonce)
	if err != nil {
		return nil, nil, nil, nil, err
	}

	return aek, bek, seed, kek.Seal(nonce, nonce, seed, nil), nil
...
	nonce := make([]byte, fs.aek.NonceSize(), fs.aek.NonceSize()+len(b)+fs.aek.Overhead())
	rand.Read(nonce)
	b = fs.aek.Seal(nonce, nonce, b, nil)
...
	nonce := make([]byte, fs.aek.NonceSize(), fs.aek.NonceSize()+len(buf)+fs.aek.Overhead())
	rand.Read(nonce)
	buf = fs.aek.Seal(nonce, nonce, buf, nil)
...
	nonce := make([]byte, o.aek.NonceSize(), o.aek.NonceSize()+len(buf)+o.aek.Overhead())
	rand.Read(nonce)
	return o.aek.Seal(nonce, nonce, buf, nil)
...
	nonce := make([]byte, cfs.aek.NonceSize(), cfs.aek.NonceSize()+len(b)+cfs.aek.Overhead())
	rand.Read(nonce)
	b = cfs.aek.Seal(nonce, nonce, b, nil)
```

Figure 1.2: Unchecked error values from `rand.Read` (`nats-server/server/filestore.go`)

In the `mb.eraseMsg` method, the error value returned by the `rand.Read` function is ignored when data is overwritten (see figure 1.3). This can cause the data to be improperly erased, with no indication to the user that this has happened.

```
// Randomize record
data := make([]byte, rl-emptyRecordLen)
rand.Read(data)
```

Figure 1.3: Unchecked error value from `rand.Read` in `mb.eraseMsg` method (`nats-server/server/filestore.go:4124–4126`)

Recommendations

Short term, add checks to ensure errors returned by `rand.Read` are non-null. In addition, change the code shown in figure 1.1 so that the `e` variable is returned rather than the `err` variable.

Long term, use the `errcheck` tool to find other instances of unchecked errors and ensure that none of them can lead to issues.

2. User and NKeyUser clone() methods can fail to deep copy an empty allowed connection types list

Severity: **Informational**	Difficulty: **N/A**
Type: Undefined Behavior	Finding ID: TOB-NATS-2
Target: `nats-server/server/auth.go:69-108`	

Description

The `User` and `NKeyUser` structs both feature an `AllowedConnectionTypes` member, a map value. When this map is present but empty, the structs' `clone()` methods will not deep-copy the map. Therefore, if a `User` or `NKeyUser` object is cloned, both the original and the copy's `AllowedConnectionTypes` field will point to the same object; if a change is made to either one, it will be propagated to the other as well. This could result in unexpected behavior when either type of user object is cloned and subsequently their associated `AllowedConnectionTypes` list is altered.

The conditional check in `User.clone()` beginning on line 88 (highlighted in figure 2.1) will not clone the `AllowedConnectionTypes` map if the original value has a zero length. This is the case if the underlying map value is nil, in which case no cloning is necessary, but is *also* the case when the map is non-nil but contains no items. In the latter case, the `AllowedConnectionTypes` map will not be deep-copied, resulting in the same issue as noted above for the `Account` field.

```
// NkeyUser is for multiple nkey based users
type NkeyUser struct {
	Nkey                   string              `json:"user"`
	Permissions            *Permissions        `json:"permissions,omitempty"`
	Account                *Account            `json:"account,omitempty"`
	SigningKey             string              `json:"signing_key,omitempty"`
	AllowedConnectionTypes map[string]struct{} `json:"connection_types,omitempty"`
}

// User is for multiple accounts/users.
type User struct {
	Username               string              `json:"user"`
	Password               string              `json:"password"`
	Permissions            *Permissions        `json:"permissions,omitempty"`
	Account                *Account            `json:"account,omitempty"`
	ConnectionDeadline     time.Time
`json:"connection_deadline,omitempty"`
	AllowedConnectionTypes map[string]struct{} `json:"connection_types,omitempty"`
}
```

```go
// clone performs a deep copy of the User struct, returning a new clone with
// all values copied.
func (u *User) clone() *User {
	if u == nil {
		return nil
	}
	clone := &User{}
	*clone = *u
	clone.Permissions = u.Permissions.clone()

	if len(u.AllowedConnectionTypes) > 0 {
		clone.AllowedConnectionTypes = make(map[string]struct{})
		for k, v := range u.AllowedConnectionTypes {
			clone.AllowedConnectionTypes[k] = v
		}
	}

	return clone
}

// clone performs a deep copy of the NkeyUser struct, returning a new clone with
// all values copied.
func (n *NkeyUser) clone() *NkeyUser {
	if n == nil {
		return nil
	}
	clone := &NkeyUser{}
	*clone = *n
	clone.Permissions = n.Permissions.clone()
	return clone
}
```

Figure 2.1: The `User` and `NKeyUser` structs and `clone()` methods at `nats-server/server/auth.go:59–108`

Recommendations

Short term, change the conditional check on line 88 to check for a non-nil map instead of a non-zero-length one.

```go
if u.AllowedConnectionTypes != nil { … }
```

Long term, when adding new fields to an existing struct, ensure that `clone()` and similar methods are updated accordingly.

3. Non-constant time comparison of plaintext passwords

Severity: **Medium**	Difficulty: **Medium**
Type: Data Exposure	Finding ID: TOB-NATS-3
Target: `server/auth.go:1415–1425`	

Description

When one component of the NATS system (Gateway, Leaf Node, client, etc.) authenticates to another using a password, the submitted password is compared to the stored one using the `comparePasswords()` function, shown in figure 3.1.

```
func comparePasswords(serverPassword, clientPassword string) bool {
	// Check to see if the server password is a bcrypt hash
	if isBcrypt(serverPassword) {
		if err := bcrypt.CompareHashAndPassword([]byte(serverPassword),
[]byte(clientPassword)); err != nil {
			return false
		}
	} else if serverPassword != clientPassword {
		return false
	}
	return true
}
```

Figure 3.1: The `comparePasswords()` function in `nats-server/server/auth.go:1415–1425`

In the event that the password is stored as a BCrypt hash, the passwords are compared using the constant-time `bcrypt.CompareHashAndPassword()` function. However, if not, `comparePasswords()` falls back to simple string comparison, which is not constant time.

Exploit Scenario

An administrator sets up a NATS Cluster that includes client credentials with a non-BCrypted password. A malicious client attempts to connect to the Cluster by brute forcing a password and is able to infer a valid password character by character since invalid prefixes are rejected early, while valid ones take more time.

Recommendations

Short term, use a constant-time string comparison for plaintext passwords: have the `comparePasswords()` function compare all characters of both strings and then return a result thereafter; the function should not terminate early.

Long term, enumerate all password checks within the application and ensure that they use constant-time comparison functions.

4. Risk of denial of service when restoring Streams

Severity: **Medium**	Difficulty: **High**
Type: Denial of Service	Finding ID: TOB-NATS-4
Target: `nats-server/server/stream.go:5657–5683`	

Description

The NATS Server allows Streams to be restored from S2-compressed tar archives. The implementation iterates over the files that are contained in the archive and copies them to the local filesystem. However, the sizes of the files are not checked, and as the archive itself is compressed, the sizes can be many times larger than the original archive. Uncompressing and writing these files to storage could take a significant amount of CPU time and exhaust the available storage on the local filesystem.

```
tr := tar.NewReader(s2.NewReader(r))
for {
	hdr, err := tr.Next()
	if err == io.EOF {
		break // End of snapshot
	}
	// (...)
	fd, err := os.OpenFile(fpath, os.O_CREATE|os.O_RDWR, 0600)
	if err != nil {
		return nil, err
	}
	_, err = io.Copy(fd, tr)
	fd.Close()
	if err != nil {
		return nil, err
	}
}
```

Figure 4.1: There is no size check during extraction, and the file contents are copied fully. (`nats-server/server/stream.go#5657–5683`)

Exploit Scenario

An attacker crafts a relatively small S2-compressed tar archive with several large files that can be compressed with a very high compression ratio. The attacker manages to have the NATS Server attempt to restore a Stream from it. The NATS Server decompresses the archive in a streaming fashion while writing the uncompressed contents to storage, resulting in high CPU use and increased storage use.

Recommendations

Short term, add a file size check to limit the amount of data that may be processed. Use `io.CopyN` instead of `io.Copy` to limit the amount of data actually copied to storage.

Long term, use static analysis tools such as Semgrep and integrate them into the project's CI process to detect similar issues during development. Semgrep's `go.lang.security.decompression_bomb.potential-dos-via-decompression-bomb` rule can discover this issue.

5. Inconsistent behavior around \r character in parser

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-NATS-5
Target: `server/parser.go`	

Description

The parser defined in `server/parser.go` handles the carriage return character (`\r`) inconsistently when it does not immediately precede a newline character (`\n`). If an `argBuf` buffer is used, the `\r` character is ignored, but if `argBuf` is nil, the `\r` character can be included in the final argument byte string and may cause a different character to be dropped (the character before the `\n` or the final character in the input buffer).

Figures 5.1 and 5.2 demonstrate this behavior in the parsing code for PUB message arguments. When a `\r` character is encountered, it causes the `c.drop` variable to be set to 1 (see line 412 in figure 5.1). Unlike normal characters (see line 440), the `\r` character does not get appended onto the `argBuf` buffer when `argBuf` is non-nil. When a `\n` character is encountered, the `arg` variable, which represents the full argument to the PUB message, may be taken in two different ways. When `argBuf` is non-nil, it is assigned into `arg` (see line 416). When `argBuf` is nil, a subset of `buf` is taken instead, optionally removing trailing characters as determined by the `drop` variable (see line 419). This means that a byte string that has a `\r` in the middle but not directly before the `\n` will have its last character before the `\n` removed; for example, a `buf` byte string of “PUB `abc\r 10\n`” will result in an arg value of “`abc\r 1`”, rather than “`abc\r 10`”.

Finally, in the case of a split buffer, the `c.drop` variable is used to determine how many trailing characters to remove (see line 1164 in figure 5.2). This means that a byte string of “PUB `a\rbc 123\n`” may result in an `arg` variable of “`a\rbc 12`”, “`a\rbc123`”, “`a\rb 123`”, “`a\rc 123`”, or “`abc 123`”, depending on how it is split.

```
409     case PUB_ARG:
410             switch b {
411             case '\r':
412                     c.drop = 1
413             case '\n':
414                     var arg []byte
415                     if c.argBuf != nil {
416                             arg = c.argBuf
417                             c.argBuf = nil
418                     } else {
419                             arg = buf[c.as : i-c.drop]
```

```
        }
        // ...
        c.drop, c.as, c.state = 0, i+1, MSG_PAYLOAD
        // ...
    default:
        if c.argBuf != nil {
            c.argBuf = append(c.argBuf, b)
        }
    }
```

Figure 5.1: Code for parsing arguments to PUB command (`nats-server/server/parser.go#409–442`)

```
// Check for split buffer scenarios for any ARG state.
if c.state == SUB_ARG || c.state == UNSUB_ARG ||
    c.state == PUB_ARG || c.state == HPUB_ARG ||
    c.state == ASUB_ARG || c.state == AUSUB_ARG ||
    c.state == MSG_ARG || c.state == HMSG_ARG ||
    c.state == MINUS_ERR_ARG || c.state == CONNECT_ARG || c.state == INFO_ARG {

    // Setup a holder buffer to deal with split buffer scenario.
    if c.argBuf == nil {
        c.argBuf = c.scratch[:0]
        c.argBuf = append(c.argBuf, buf[c.as:i-c.drop]...)
    }
    // Check for violations of control line length here. Note that this is not
    // exact at all but the performance hit is too great to be precise, and
    // catching here should prevent memory exhaustion attacks.
    if err := c.overMaxControlLineLimit(c.argBuf, mcl); err != nil {
        return err
    }
}
```

Figure 5.2: Code for handling split buffers (`nats-server/server/parser.go#1154–1172`)

Exploit Scenario

An authenticated attacker sends a "`PUB abc\r 0\r\n\r\n`" command to a NATS Server that has pedantic mode disabled, causing an empty message to be published to the "`abc\r`" topic. This causes "`MSG abc\r 0\r\n\r\n`" messages to be propagated throughout the NATS Cluster. This typically does not cause any issues, but when a message gets split up as "`MSG abc\r 0`" "`\r\n\r\n`", a parse error occurs, followed by a dropped connection. In this way, the attacker can cause connections to be dropped between nodes that he otherwise should have no control over.

Recommendations

Short term, add checks to the parser that throw an error if `drop` is set to 1 and a character other than `\n` is encountered.

Long term, modify NATS's parser fuzz test so that it also checks that byte strings are parsed identically regardless of how they are split. A sample fuzz test that does this is provided in appendix E.

6. Use of unpinned third-party workflow	
Severity: **Medium**	Difficulty: **High**
Type: Patching	Finding ID: TOB-NATS-6
Target: `nats-server/.github/actions/nightly-release/action.yaml`, `nats-server/.github/workflows/cov.yaml`	

Description

Some of NATS's GitHub Actions workflows use third-party dependencies whose versions are selected by Git tag rather than commit hash (see figures 6.1 and 6.2). Git tags are malleable. This means that, for example, while `jandelgado/gcov2lcov-action` is pinned to `v1.0.9`, the upstream may silently change the reference pointed to by `v1.0.9`. This can include malicious re-tags, in which case NATS's workflow will silently update to use the malicious workflow as a dependency.

GitHub's security hardening guidelines for third-party actions encourages developers to pin third-party actions to a full-length commit hash. Generally excluded from this are "official" actions under the `actions` organization.

```
- name: goreleaser
  uses: goreleaser/goreleaser-action@v5
  with:
    workdir: "${{ inputs.workdir }}"
    version: latest
    args: release --snapshot --config .goreleaser-nightly.yml
```

Figure 6.1: `nats-server/.github/actions/nightly-release/action.yaml#33–38`

```
- name: Convert coverage.out to coverage.lcov
  uses: jandelgado/gcov2lcov-action@v1.0.9
  with:
    infile: acc.out
    working-directory: src/github.com/nats-io/nats-server

- name: Coveralls
  uses: coverallsapp/github-action@v2
  with:
    github-token: ${{ secrets.github_token }}
    file: src/github.com/nats-io/nats-server/coverage.lcov
```

Figure 6.2: `nats-server/.github/workflows/cov.yaml#35–45`

Exploit Scenario

An attacker (or compromised maintainer) silently overwrites the `v1.0.9` tag on `jandelgado/gcov2lcov-action` with a malicious version of the action, allowing the GitHub token for the `nats-server` repository to be stolen.

An attacker (or compromised maintainer) silently overwrites the `v5` tag on `goreleaser/goreleaser-action` with a malicious version of the action, allowing both the GitHub token for the `nats-server` repository and the NATS team's username and password for Dockerhub to be stolen.

Recommendations

Short term, replace the current version tags with full-length commit hashes corresponding to the revision that each workflow is intended to use.

Long term, use Semgrep static analysis on the codebase regularly; the `yaml.github-actions.security.third-party-action-not-pinned-to-commit-sha.third-party-action-not-pinned-to-commit-sha` rule would have found this problem. Appendix F contains instructions on how to perform static analysis on the codebase using Semgrep.

7. Use of non-TLS download in Travis CI configuration file	
Severity: **Informational**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-NATS-7
Target: `nats-server/.travis.yml`	

Description

The configuration file for Travis CI files runs the GoReleaser installation script downloaded from an `http` link rather than an `https` link (see figure 7.1). This means that a machine-in-the-middle attacker may be able to provide a malicious version of the script that steals tokens or release a malicious version of the NATS Server.

```
deploy:
  provider: script
  cleanup: true
  script: curl -sL http://git.io/goreleaser | bash
  on:
    tags: true
    condition: ($TRAVIS_GO_VERSION =~ 1.21) && ($TEST_SUITE = "compile")
```

Figure 7.1: `nats-server/.travis.yml#49–55`

Recommendations

Short term, switch to GoReleaser's up-to-date recommended installation instructions. The highlighted text in figure 7.1 should be changed to `curl -sfL https://goreleaser.com/static/run | bash`. Ensure Cosign is installed in the CI environment to allow the script to check the release signatures.

Long term, use Semgrep static analysis on the codebase regularly; the `trailofbits.generic.curl-unencrypted-url.curl-unencrypted-url` rule would have found this problem. Appendix F contains instructions on how to perform static analysis on the codebase using Semgrep.

Consider manually installing a known version of GoReleaser with a known hash to reduce the risk of CI tampering due to a third-party script compromise.

8. Missing mutex unlocks before return statements

Severity: **Informational**	Difficulty: **High**
Type: Timing	Finding ID: TOB-NATS-8
Target: `nats-server/server/stream.go`, `nats-server/server/filestore.go`	

Description

Multiple functions in the NATS codebase lock a mutex but do not unlock the mutex before returning in the case of an error. This could lead to a deadlock.

Instances of this bug are shown in figures 8.1 through 8.3. In each figure, the line highlighted and marked with (1) shows the mutex lock, and the line(s) highlighted and marked with (2) show the return statement(s) that do not unlock the mutex.

```
// swapSigSubs will update signal Subs for a new subject filter.
// consumer lock should not be held.
func (mset *stream) swapSigSubs(o *consumer, newFilters []string) {
	mset.clsMu.Lock()  // (1)
	o.mu.Lock()

	if o.closed || o.mset == nil {
		o.mu.Unlock()
		return  // (2)
	}
```

Figure 8.1: `nats-server/server/stream.go#5280–5289`

```
// Truncate this message block to the storedMsg.
func (mb *msgBlock) truncate(sm *StoreMsg) (nmsgs, nbytes uint64, err error) {

	// ...

	mb.mu.Lock()  // (1)

	// ...

	// If the block is compressed then we have to load it into memory
	// and decompress it, truncate it and then write it back out.
	// Otherwise, truncate the file itself and close the descriptor.
	if mb.cmp != NoCompression {
		buf, err := mb.loadBlock(nil)
		if err != nil {
			return 0, 0, fmt.Errorf("failed to load block from disk: %w",
err)  // (2)
			}
			if mb.bek != nil && len(buf) > 0 {
				bek, err := genBlockEncryptionKey(mb.fs.fcfg.Cipher, mb.seed,
mb.nonce)
				if err != nil {
					return 0, 0, err  // (2)
				}
				mb.bek = bek
				mb.bek.XORKeyStream(buf, buf)
			}
			buf, err = mb.decompressIfNeeded(buf)
			if err != nil {
				return 0, 0, fmt.Errorf("failed to decompress block: %w", err)
// (2)
			}
			buf = buf[:eof]
			copy(mb.lchk[0:], buf[:len(buf)-checksumSize])
			buf, err = mb.cmp.Compress(buf)
			if err != nil {
				return 0, 0, fmt.Errorf("failed to recompress block: %w", err)
// (2)
			}
			meta := &CompressionInfo{
				Algorithm:    mb.cmp,
				OriginalSize: uint64(eof),
			}
			buf = append(meta.MarshalMetadata(), buf...)
			if mb.bek != nil && len(buf) > 0 {
				bek, err := genBlockEncryptionKey(mb.fs.fcfg.Cipher, mb.seed,
mb.nonce)
				if err != nil {
					return 0, 0, err  // (2)
				}
				mb.bek = bek
				mb.bek.XORKeyStream(buf, buf)
			}
			n, err := mb.writeAt(buf, 0)
			if err != nil {
				return 0, 0, fmt.Errorf("failed to rewrite compressed block:
%w", err)  // (2)
			}
			if n != len(buf) {
				return 0, 0, fmt.Errorf("short write (%d != %d)", n, len(buf))
// (2)
			}
			mb.mfd.Truncate(int64(len(buf)))
			mb.mfd.Sync()
		} else if mb.mfd != nil {
			mb.mfd.Truncate(eof)
			mb.mfd.Sync()
			// Update our checksum.
			var lchk [8]byte
		mb.mfd.ReadAt(lchk[:], eof-8)
		copy(mb.lchk[0:], lchk[:])
	} else {
		mb.mu.Unlock()
		return 0, 0, fmt.Errorf("failed to truncate msg block %d, file not
open", mb.index)
	}
```

Figure 8.2: `nats-server/server/filestore.go#4170–4279`

```
// Truncate will truncate a stream store up to seq. Sequence needs to be valid.
func (fs *fileStore) Truncate(seq uint64) error {
	// Check for request to reset.
	if seq == 0 {
		return fs.reset()
	}

	fs.mu.Lock()  // (1)

	if fs.closed {
		fs.mu.Unlock()
		return ErrStoreClosed
	}
	if fs.sips > 0 {
		fs.mu.Unlock()
		return ErrStoreSnapshotInProgress
	}

	nlmb := fs.selectMsgBlock(seq)
	if nlmb == nil {
		fs.mu.Unlock()
		return ErrInvalidSequence
	}
	lsm, _, _ := nlmb.fetchMsg(seq, nil)
	if lsm == nil {
		fs.mu.Unlock()
		return ErrInvalidSequence
	}

	// Set lmb to nlmb and make sure writeable.
	fs.lmb = nlmb
	if err := nlmb.enableForWriting(fs.fip); err != nil {
		return err  // (2)
	}
```

Figure 8.3: `nats-server/server/filestore.go#6874–6907`

Recommendations

Short term, add calls to the `Unlock()` method before each of these return statements. If it is possible, use a `defer mutex.Unlock()` statement immediately after the `Lock()`

method is called instead so that the mutex will be unlocked regardless of the code path taken.

Long term, use Semgrep static analysis on the codebase regularly; the `trailofbits.go.missing-unlock-before-return.missing-unlock-before-return` rule would have found this problem. Appendix F contains instructions on how to perform static analysis on the codebase using Semgrep.

9. Windows DLL loading susceptible to DLL hijacking attacks

Severity: **Informational**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-NATS-9

Target: `nats-server/server/certstore/certstore_windows.go`, `nats-server/server/pse/pse_windows.go`, `nats-server/server/sysmem/mem_windows.go`

Description

NATS uses DLL loading functions, which are susceptible to DLL hijacking attacks (see figures 9.1 through 9.3). An attacker may be able to cause malicious code to execute by creating a DLL in the same directory as the NATS executable or in the current working directory from which NATS is being run. In the former case, the DLL in the same directory as the NATS executable would have precedence over the Windows system DLL. In the latter case, the DLL in the current working directory would have lower precedence than the Windows system DLL and would be used only if the system DLL were not found.

If NATS is being run with elevated privileges, the DLL hijacking attack could additionally allow the attacker to perform a local privilege escalation.

```
// These DLLs must be available on all Windows hosts
winCrypt32 = windows.MustLoadDLL("crypt32.dll")
winNCrypt  = windows.MustLoadDLL("ncrypt.dll")
```

Figure 9.1: `nats-server/server/certstore/certstore_windows.go#117–119`

```
var (
	pdh                               = syscall.NewLazyDLL("pdh.dll")
```

Figure 9.2: `nats-server/server/pse/pse_windows.go#30–31`

```
func Memory() int64 {
	kernel32, err := syscall.LoadDLL("kernel32.dll")
```

Figure 9.3: `nats-server/server/sysmem/mem_windows.go#32–33`

Recommendations

Short term, replace these DLL loading functions with calls to the `windows.NewLazySystemDLL` function, which only searches for system DLLs.

Long term, use Semgrep static analysis on the codebase regularly; the `trailofbits.go.unsafe-dll-loading.unsafe-dll-loading` rule would have found this problem. Appendix F contains instructions on how to perform static analysis on the codebase using Semgrep.

10. HTTP servers are vulnerable to Slowloris denial-of-service attacks

Severity: **Low**	Difficulty: **High**
Type: Denial of Service	Finding ID: TOB-NATS-10
Target: `nats-server/server/server.go`	

Description

The NATS HTTP Servers for profiling and monitoring are vulnerable to Slowloris denial-of-service attacks. This attack takes advantage of Servers that keep connections alive for incomplete requests without defining any timeout. By flooding the Server with incomplete requests, the attacker can cause an out-of-memory error or can fill up the connection pool, causing real requests to be denied.

```
srv := &http.Server{
	Addr:           hp,
	Handler:        http.DefaultServeMux,
	MaxHeaderBytes: 1 << 20,
}
```

Figure 10.1: Server without timeout used for profiling (`nats-server/server/server.go#2750–2754`)

```
srv := &http.Server{
	Addr:           hp,
	Handler:        mux,
	MaxHeaderBytes: 1 << 20,
	ErrorLog:       log.New(&captureHTTPServerLog{s, "monitoring: "}, _EMPTY_,
0),
}
```

Figure 10.2: Server without timeout used for monitoring (`nats-server/server/server.go#2949–2954`)

Recommendations

Short term, use the `ReadTimeout` parameter on the `http.Server` struct to set a request timeout. Alternatively, use the `ReadHeaderTimeout` parameter on the `http.Server` struct to set a timeout on requests' headers and manually add timeouts at each code location in which the request body is read.

Long term, use Semgrep static analysis on the codebase regularly; the `go.net.dos.slowloris-dos.slowloris-dos` rule would have found this problem. Appendix F contains instructions on how to perform static analysis on the codebase using Semgrep.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories

Category	Description
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	**Description**
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Configuration	The configuration of system components in accordance with best practices
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Data Handling	The safe handling of user inputs and data processed by the system
Documentation	The presence of comprehensive and readable codebase documentation
Maintenance	The timely maintenance of system components to mitigate risk
Memory Safety and Error Handling	The presence of memory safety and robust error-handling mechanisms
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Code Quality Recommendations

This appendix contains findings that do not have immediate or obvious security implications or that were discovered but not fully investigated due to time constraints or scope limitations.

- **Break up long functions, such as `processClientOrLeafAuthentication()` at `server/auth.go:575-1060`:** Several security-relevant code paths involve long, multihundred-line functions handling numerous distinct functionalities (e.g., authentication methods). While they are not necessarily vulnerable in their own right, they are difficult to comprehensively audit and understand due to their complexity, which makes it more likely that logic-related security issues will be unintentionally introduced by future modifications and less likely that they will be discovered.

- **Replace `c.pa.size - LEN_CR_LF` with `c.pa.size - 2` in `server/parser.go`.** The statement `i = c.as + c.pa.size - LEN_CR_LF` is used in multiple locations in `server/parser.go` in order to jump to the last character of a message payload. The `c.as` variable represents the first character of the message payload, and the `c.pa.size` variable represents the length of the message payload. The `c.pa.size` variable does not include the added length from the trailing CRLF characters that occur after the message payload, but the use of `c.pa.size - LEN_CR_LF` implies that `c.pa.size` *does* include the CRLF characters. The reason the statement works correctly is that `c.as + c.pa.size` represents the character *after* the end of the message payload, `c.as + c.pa.size - 1` represents the character *at* the end of the message payload, `c.as + c.pa.size - 2` takes off one more character to account for the `i++` statement in the parser's `for` loop, and `c.as + c.pa.size - LEN_CR_LF` equals `c.as + c.pa.size - 2` (since CRLF is two characters long). The use of the `LEN_CR_LF` constant in this context can mislead developers as to how the code functions and can make code maintenance more difficult.

```
case PUB_ARG:
	switch b {
	case '\r':
		c.drop = 1
	case '\n':
		var arg []byte
		if c.argBuf != nil {
			arg = c.argBuf
			c.argBuf = nil
		} else {
			arg = buf[c.as : i-c.drop]
		}
		if err := c.overMaxControlLineLimit(arg, mcl); err != nil {
			return err
		}
		if trace {
			c.traceInOp("PUB", arg)
		}
		if err := c.processPub(arg); err != nil {
			return err
		}

		c.drop, c.as, c.state = 0, i+1, MSG_PAYLOAD
		// If we don't have a saved buffer then jump ahead with
		// the index. If this overruns what is left we fall out
		// and process split buffer.
		if c.msgBuf == nil {
			i = c.as + c.pa.size - LEN_CR_LF
		}
	default:
		if c.argBuf != nil {
			c.argBuf = append(c.argBuf, b)
		}
	}
```

Figure C.1: Example of misleading usage of LEN_CR_LF constant in the parser (nats-server/server/parser.go#409–442)

D. Instances of Unchecked Type Assertions

The following is a list of all instances of unchecked type assertions in the NATS codebase. Unchecked type assertions can lead to panics if the real and expected types differ.

- `server/auth.go:320`
- `server/auth.go:335`
- `server/auth_callout.go:369`
- `server/dirstore.go:363`
- `server/dirstore.go:423`
- `server/dirstore.go:433`
- `server/dirstore.go:603`
- `server/dirstore.go:623`
- `server/dirstore.go:637`
- `server/dirstore.go:638`
- `server/dirstore.go:661`
- `server/events.go:373`
- `server/events.go:908`
- `server/events.go:1330`
- `server/events.go:1420`
- `server/events.go:1432`
- `server/events.go:1437`
- `server/events.go:1480`
- `server/events.go:1634`
- `server/events.go:1670`
- `server/events.go:1958`
- `server/gateway.go:499`
- `server/gateway.go:526`
- `server/gateway.go:865`
- `server/gateway.go:1158`
- `server/gateway.go:1775`
- `server/gateway.go:1790`
- `server/gateway.go:1818`
- `server/gateway.go:1875`
- `server/gateway.go:1985`
- `server/gateway.go:2078`
- `server/gateway.go:2357`
- `server/gateway.go:2364`
- `server/gateway.go:2413`
- `server/gateway.go:2471`
- `server/gateway.go:2769`
- `server/gateway.go:2773`
- `server/gateway.go:2777`

- server/gateway.go:3034
- server/gateway.go:3061
- server/gateway.go:3225
- server/gateway.go:3226
- server/ipqueue.go:79
- server/jetstream.go:768
- server/jetstream.go:847
- server/jetstream_errors.go:87
- server/leafnode.go:703
- server/leafnode.go:728
- server/leafnode.go:1587
- server/leafnode.go:1972
- server/monitor.go:851
- server/monitor.go:997
- server/monitor.go:1038
- server/monitor.go:1138
- server/monitor.go:1718
- server/monitor.go:2010
- server/monitor.go:2025
- server/monitor.go:2289
- server/monitor.go:2300
- server/monitor.go:2539
- server/monitor.go:2584
- server/mqtt.go:496
- server/mqtt.go:620
- server/mqtt.go:995
- server/mqtt.go:1657
- server/mqtt.go:1671
- server/mqtt.go:1681
- server/mqtt.go:1694
- server/mqtt.go:1707
- server/mqtt.go:1716
- server/mqtt.go:1725
- server/mqtt.go:1739
- server/mqtt.go:1761
- server/mqtt.go:1777
- server/mqtt.go:1791
- server/mqtt.go:1804
- server/mqtt.go:1818
- server/mqtt.go:1842
- server/mqtt.go:1871
- server/mqtt.go:2116
- server/mqtt.go:2874
- server/mqtt.go:2978

- `server/raft.go:533`
- `server/raft.go:541`
- `server/raft.go:593`
- `server/raft.go:1429`
- `server/raft.go:1444`
- `server/raft.go:1988`
- `server/raft.go:2018`
- `server/raft.go:2046`
- `server/raft.go:2194`
- `server/raft.go:2228`
- `server/raft.go:2238`
- `server/reload.go:1219`
- `server/reload.go:1229`
- `server/reload.go:1231`
- `server/reload.go:1233`
- `server/reload.go:1235`
- `server/reload.go:1237`
- `server/reload.go:1239`
- `server/reload.go:1241`
- `server/reload.go:1243`
- `server/reload.go:1245`
- `server/reload.go:1247`
- `server/reload.go:1249`
- `server/reload.go:1251`
- `server/reload.go:1253`
- `server/reload.go:1263`
- `server/reload.go:1269`
- `server/reload.go:1270`
- `server/reload.go:1300`
- `server/reload.go:1302`
- `server/reload.go:1304`
- `server/reload.go:1304`
- `server/reload.go:1306`
- `server/reload.go:1308`
- `server/reload.go:1310`
- `server/reload.go:1312`
- `server/reload.go:1314`
- `server/reload.go:1316`
- `server/reload.go:1341`
- `server/reload.go:1342`
- `server/reload.go:1362`
- `server/reload.go:1363`
- `server/reload.go:1494`
- `server/reload.go:1495`

- server/reload.go:1503
- server/reload.go:1504
- server/reload.go:1518
- server/reload.go:1519
- server/reload.go:1552
- server/reload.go:1553
- server/reload.go:1563
- server/reload.go:1564
- server/reload.go:1565
- server/reload.go:1566
- server/reload.go:1567
- server/reload.go:1568
- server/reload.go:1572
- server/reload.go:1573
- server/reload.go:1584
- server/reload.go:1585
- server/reload.go:1586
- server/reload.go:1587
- server/reload.go:1588
- server/reload.go:1589
- server/reload.go:1591
- server/reload.go:1593
- server/reload.go:1599
- server/reload.go:1602
- server/reload.go:1628
- server/reload.go:1630
- server/reload.go:1632
- server/reload.go:1633
- server/reload.go:1724
- server/reload.go:1731
- server/reload.go:1907
- server/reload.go:1959
- server/reload.go:2155
- server/reload.go:2244
- server/reload.go:2370
- server/reload.go:2381
- server/route.go:150
- server/route.go:808
- server/route.go:946
- server/route.go:1192
- server/route.go:1274
- server/route.go:1386
- server/route.go:1586
- server/route.go:1595

- server/route.go:1831
- server/route.go:2458
- server/route.go:2507
- server/sendq.go:108
- server/server.go:1091
- server/server.go:1184
- server/server.go:1192
- server/server.go:1205
- server/server.go:1214
- server/server.go:1219
- server/server.go:1380
- server/server.go:1406
- server/server.go:1408
- server/server.go:1580
- server/server.go:1605
- server/server.go:1726
- server/server.go:1804
- server/server.go:1912
- server/server.go:2224
- server/server.go:2284
- server/server.go:2607
- server/server.go:2621
- server/server.go:2748
- server/server.go:2912
- server/server.go:3262
- server/server.go:3537
- server/server.go:3593
- server/server.go:3603
- server/server.go:3613
- server/server.go:3860
- server/server.go:4374
- server/stree/dump.go:35
- server/stree/stree.go:66
- server/stree/stree.go:129
- server/stree/stree.go:212
- server/stree/stree.go:234
- server/stree/stree.go:244
- server/stree/stree.go:282
- server/stree/stree.go:311
- server/stree/stree.go:356
- server/websocket.go:402
- server/websocket.go:406
- server/websocket.go:748
- server/websocket.go:1098

E. Automated Testing Artifacts

This appendix contains information about tooling used in our automated testing campaigns.

Fuzzing Inconsistent Behavior from TOB-NATS-5

We used the built-in Go fuzzer to find extra example cases of inconsistent behavior described in TOB-NATS-5. The harness provided in figure E.3 can be added to `parser_test.go` and run as follows to discover new instances of differing behavior when the buffer being parsed is split differently.

```
go test -run FuzzSplit -fuzz FuzzSplit github.com/nats-io/nats-server/v2/server -v
```

Figure E.1: The command to run the harness in figure E.3

The test case seeds were collected from other tests in the file. The harness is able to find an example of inconsistent behavior in about a minute when run on a 2021 MacBook Pro. Note that this harness only lightly verifies that buffers that parse successfully as a whole can be processed in a split manner, but it does not check whether two pieces of buffer that parse correctly in succession can also be parsed in one go. It also does not check that the results from parsing are equivalent in both cases.

```
% go test -run FuzzSplit -fuzz FuzzSplit github.com/nats-io/nats-server/v2/server -v
=== RUN   FuzzSplit
fuzz: elapsed: 0s, gathering baseline coverage: 0/325 completed
fuzz: elapsed: 1s, gathering baseline coverage: 325/325 completed, now fuzzing with
10 workers
fuzz: elapsed: 3s, execs: 20184 (6726/sec), new interesting: 6 (total: 331)
// (...)
fuzz: elapsed: 33s, execs: 107827 (1577/sec), new interesting: 36 (total: 361)
fuzz: minimizing 45-byte failing input file
fuzz: elapsed: 36s, minimizing
--- FAIL: FuzzSplit (35.98s)
    --- FAIL: FuzzSplit (0.00s)
        parser_test.go:275: Second split failed with i 7 ["PUB 0\r "] ["00\n"]

    Failing input written to testdata/fuzz/FuzzSplit/5e2dcc3924dda852
    To re-run:
    go test -run=FuzzSplit/5e2dcc3924dda852
=== NAME
FAIL
exit status 1
FAIL    github.com/nats-io/nats-server/v2/server        36.953s
```

Figure E.2: Fuzz results obtained from running the command in figure E.1

```
func FuzzSplit(f *testing.F) {
	testcases := [][]byte{
		[]byte("PING\r\n"),
		[]byte("PING  \r"),
		[]byte("PING  \r  \n"),
		[]byte("PONG\r\n"),
		[]byte("PONG  \r"),
		[]byte("PONG  \r  \n"),
		[]byte("PONG\r\n"),
		[]byte("CONNECT
{\"verbose\":false,\"pedantic\":true,\"tls_required\":false}\r\n"),
		[]byte("SUB foo 1\r"),
		[]byte("PUB foo 5\r\nhello\r"),
		[]byte("PUB foo.bar INBOX.22 11\r\nhello world\r"),
		[]byte("PUB foo.bar 11\r\nhello world hello world\r"),
		[]byte("PUB foo
3333333333333333333333333333333333333333333333333333333333333333\r\n"),
		[]byte("HPUB foo 12 17\r\nname:derek\r\nHELLO\r"),
		[]byte("HPUB foo INBOX.22 12 17\r\nname:derek\r\nHELLO\r"),
		[]byte("HPUB foo INBOX.22 0 5\r\nHELLO\r"),
		[]byte("HMSG $foo foo 10 8\r\nXXXhello\r"),
		[]byte("HMSG $foo foo 3 8\r\nXXXhello\r"),
		[]byte("HMSG $G foo.bar INBOX.22 3 14\r\nOK:hello world\r"),
		[]byte("HMSG $G foo.bar + reply baz 3 14\r\nOK:hello world\r"),
		[]byte("HMSG $G foo.bar | baz 3 14\r\nOK:hello world\r"),
		[]byte("MSG $foo foo 5\r\nhello\r"),
		[]byte("RMSG $foo foo 5\r\nhello\r"),
		[]byte("RMSG $G foo.bar INBOX.22 11\r\nhello world\r"),
		[]byte("RMSG $G foo.bar + reply baz 11\r\nhello world\r"),
		[]byte("RMSG $G foo.bar | baz 11\r\nhello world\r"),
		[]byte("abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"),
		[]byte("+OK\r\n"),
		[]byte("PUB foo.bar.baz 2\r\nok\r\n"),
	}
	for _, tc := range testcases {
		f.Add(tc)
	}

	f.Fuzz(func(t *testing.T, buffer []byte) {
		c := dummyClient()
		err := c.parse(buffer)
		if err != nil {
			return
		}
		for i, _ := range buffer {
			c := dummyClient()
```

```
                    err = c.parse(buffer[:i])
                    if err != nil {
                          t.Fatalf("First split failed with i %v [%q] [%q]", i,
string(buffer[:i]), string(buffer[i:]))
                    }
                    err = c.parse(buffer[i:])
                    if err != nil {
                          t.Fatalf("Second split failed with i %v [%q] [%q]", i,
string(buffer[:i]), string(buffer[i:]))
                    }
              }
      })
}
```

Figure E.3: A fuzzing harness that tests that buffers that can be parsed correctly as a whole also get processed successfully when split in two parts

This difference in behavior can also be observed by running the found case against a live NATS Server; for example, if `localhost:4222` were a NATS Server, the following two commands would elicit a different response from the Server:

```
% (echo -en "PUB 0\r "; sleep 1; echo -en "00\n"; sleep 1) | nc localhost 4222
% (echo -en "PUB 0\r 00\n"; sleep 1) | nc localhost 4222
```

Figure E.4: Two commands that send the same string to a Server in `localhost:4222`, but the first one does it in two parts, while the second one does it all in one go

```
% docker run --rm --name nats-main -p 4222:4222 -p 6222:6222 -p 8222:8222 nats -D
# (...)
# Response logs from NATS for the first command
[1] 2024/03/31 20:21:43.003748 [DBG] 192.168.65.1:33467 - cid:8 - Client connection
created
[1] 2024/03/31 20:21:44.000094 [ERR] 192.168.65.1:33467 - cid:8 - processPub Parse
Error: "0\r00"
[1] 2024/03/31 20:21:44.003362 [DBG] 192.168.65.1:33467 - cid:8 - Client connection
closed: Protocol Violation

# Response logs from NATS for the second command
[1] 2024/03/31 20:21:55.630639 [DBG] 192.168.65.1:33468 - cid:9 - Client connection
created
[1] 2024/03/31 20:21:56.624102 [DBG] 192.168.65.1:33468 - cid:9 - Client connection
closed: Client Closed
```

Figure E.5: Output from the NATS Server when running the examples in figure E.4

F. Automated Static Analysis

This appendix describes the setup of the automated analysis tools used during this audit.

Though static analysis tools frequently report false positives, they detect certain categories of issues, such as memory leaks, misspecified format strings, and the use of unsafe APIs, with essentially perfect precision. We recommend periodically running these static analysis tools and reviewing their findings.

Semgrep

To install Semgrep, we used `pip` by running `python3 -m pip install semgrep`.

To run Semgrep on the codebase, we ran the following commands in the root directory of the project:

```
semgrep --config "r/all" --metrics=off
```

We recommend integrating Semgrep into the project's CI/CD pipeline. To thoroughly understand the Semgrep tool, refer to the Trail of Bits Testing Handbook, which offers guidance on streamlining the use of Semgrep and improving security testing effectiveness. Also, consider doing the following:

- Limit results to error severity only by using the `--severity` ERROR flag.
- Focus first on rules with high confidence and medium- or high-impact metadata.
- Use the SARIF format (by using the `--sarif` Semgrep argument) with the SARIF Explorer for Visual Studio Code extension. This will make it easier to review the analysis results and drill down into specific issues to understand their impact and severity.

CodeQL

We installed CodeQL by following CodeQL's installation guide.

After installing CodeQL, we ran the following command to create the project database for the NATS Server repository:

```
codeql database create nats.db --language=go
```

We then ran the following command to query the database:

```
codeql database analyze nats.db --format=sarif-latest
--output=codeql_res.sarif -- go-lgtm-full go-security-and-quality
go-security-experimental
```

The resulting SARIF file can be reviewed with the SARIF Explorer extension as well. To understand CodeQL more thoroughly, we recommend reviewing the CodeQL chapter in the Testing Handbook.

actionlint

We installed actionlint by following actionlint's quick start guide. We also installed its two external dependencies, shellcheck and pyflakes, using their corresponding installation guides.

After installing actionlint, we ran the following command to analyze the repository:

```
actionlint
```

D. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

On February 18, 2024, Trail of Bits reviewed the fixes and mitigations implemented by Synadia Communications for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

On April 17, 2025, Trail of Bits reviewed an additional fix for issue TOB-NATS-4.

In summary, of the 10 issues described in this report, Synadia Communications has resolved nine issues and has not resolved the remaining issue. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Status
1	Ignored error values during file store operations	Resolved
2	User and NKeyUser clone() methods can fail to deep copy an empty allowed connection types list	Resolved
3	Non–constant time comparison of plaintext passwords	Resolved
4	Risk of denial of service when restoring Streams	Resolved
5	Inconsistent behavior around \r character in parser	Unresolved
6	Use of unpinned third-party workflow	Resolved
7	Use of non-TLS download in Travis CI configuration file	Resolved
8	Missing mutex unlocks before return statements	Resolved
9	Windows DLL loading susceptible to DLL hijacking attacks	Resolved
10	HTTP servers are vulnerable to Slowloris denial-of-service attacks	Resolved

Detailed Fix Review Results

TOB-NATS-1: Ignored error values during file store operations
Resolved in PR #5248. The mistaken variable noted in figure 1.1 has been corrected, and error checks have been added for each `rand.Read` and `eraseMsg` call.

TOB-NATS-2: User and NKeyUser clone() methods can fail to deep copy an empty allowed connection types list
Resolved in PR #5246. The problematic conditional check noted in figure 2.1 has been corrected to account for empty maps.

TOB-NATS-3: Non–constant time comparison of plaintext passwords
Resolved in PR #5247. The noted passwords are now compared in constant time.

TOB-NATS-4: Risk of denial of service when restoring Streams
Resolved in commit `306781218cb0`. The restore process now checks storage limits as it uncompresses the data, and the process is stopped if the storage limit would be exceeded.

TOB-NATS-5: Inconsistent behavior around \r character in parser
Unresolved. A fix has not yet been implemented for this issue.

TOB-NATS-6: Use of unpinned third-party workflow
Resolved in PR #5837. The noted third-party workflows have all been pinned.

TOB-NATS-7: Use of non-TLS download in Travis CI configuration file
Resolved in PR #5514. The problematic plaintext URL has been replaced with an alternate HTTPS source.

TOB-NATS-8: Missing mutex unlocks before return statements
Resolved in PR #5276. The missing mutex unlocks have been added.

TOB-NATS-9: Windows DLL loading susceptible to DLL hijacking attacks
Resolved in PR #5836. The noted unsafe instances of `MustLoadDLL()` have been replaced by `NewLazySystemDLL()`, which is not susceptible to DLL hijacking.

TOB-NATS-10: HTTP servers are vulnerable to Slowloris denial-of-service attacks
Resolved in PR #5790. The HTTP servers have been configured with a five-second read timeout.

G. Fix Review Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

Fix Status	
Status	Description
Undetermined	The status of the issue was not determined during this engagement.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
497 Carroll St., Space 71, Seventh Floor
Brooklyn, NY 11215
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.