Security Audit - Firedancer v0.4

conducted by Neodyme AG

Lead	Thomas Lambertz
Auditor	Simon Klier
Auditor	Benno Fünfstück
Auditor	Alain Rödel
Auditor	Florian Schweins

April 01, 2025

Table of Contents

Appendices

1 | Executive Summary

Neodyme was engaged to conduct a time-boxed security audit of Firedancer's major update to version 0.4, carried out during late February and March 2025. A prior audit of version 0.1 was performed in 2024 and included comprehensive threat modeling and architectural analysis.

Given that this engagement focused on an update rather than a fresh implementation, much of the original threat model remains valid. As such, this audit concentrated on **code changes**, especially in areas that diverged from v0.1, and on **compatibility concerns** arising from changes in the upstream Agave client.

Consistent with the previous audit, **no remote code execution (RCE) vulnerabilities** were identified. Most findings related to denial-of-service (DoS) scenarios, particularly in the QUIC networking stack. One high-severity issue was discovered, involving an unintuitive Agave feature activation that could lead to runtime desynchronization. This issue was already fixed upstream by the time of reporting. **No critical-severity vulnerabilities were found.**

Overall, Firedancer continues to demonstrate a strong security posture. The validator is well-architected and maintains a clear focus on **defense-in-depth**, with **sandboxing** standing out as a particularly well-implemented safeguard.

According to Neodymes Rating Classification, the audit uncovered **2 security relevant** and **3 informational** findings, distributed across severity levels as shown below:

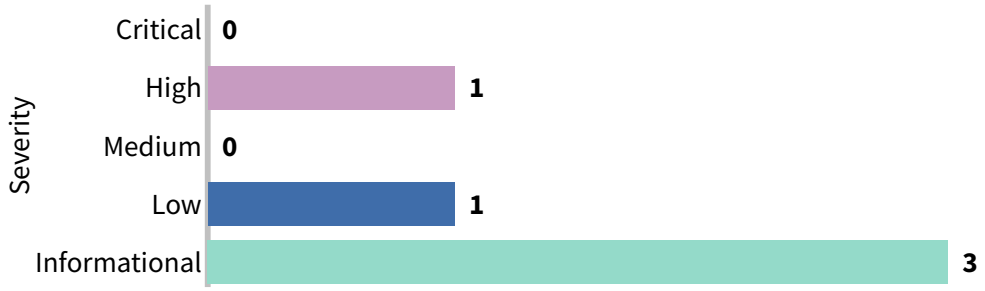

Figure 1: Overview of Findings

All findings were reported to the Firedancer development team and were addressed promptly. Neodyme has verified the security fixes for correctness and completeness.

2 | Introduction

Jump Crypto engaged Neodyme to conduct a comprehensive security audit of the Firedancer Solana validator, with a specific focus on the changes introduced in version 0.4. The audit was carried out over the course of late February and March 2025 by a team of five senior security researchers, each focusing on different aspects of the validator stack. Several members of the team have an extensive track record in identifying critical vulnerabilities in both Solana programs and validator implementations across various blockchains. Collectively, their prior disclosures have prevented the loss of over 1 billion USD in digital assets.

This engagement builds on Neodyme's previous audit of Firedancer version 0.1, conducted in 2024. Much of the original threat modeling and architectural analysis from that audit remains valid, as the core design of Firedancer has not significantly changed. As such, this audit concentrated on **codebase changes, newly introduced functionality**, and **updates in Agave** that could introduce new incompatibilities or attack surfaces.

Firedancer is a high-performance, alternative implementation of the Solana validator client. While it continues to rely on the original Agave client for certain components — including the runtime, blockstore, consensus, and gossip — it replaces or extends others, such as networking, transaction ingress and validation, and leader operations like block packing.

Version 0.4 introduces a variety of improvements across the codebase, ranging from foundational refactors to major features like bundle support, live identity switching, QUIC enhancements, and plugin infrastructure. These changes increase Firedancer's capability and complexity, warranting renewed scrutiny.

For a detailed examination of the overall architecture and threat landscape, we refer readers to the Firedancer v0.1 audit report. This report focuses exclusively on new and modified components introduced in version 0.4.

3 | Scope

This audit focuses specifically on the updates made between Firedancer versions 0.1 and 0.4. While the majority of the codebase was finalized at the outset of the audit, some components were still undergoing internal review and had not been fully integrated. As a result, our analysis began with the fully completed sections and expanded to newly finalized components as they became available. Initial attention was directed toward the new QUIC implementation and identity-changing functionality. Subsequently, we performed a review of all modified components.

Source Code Revisions

The source code analyzed during this audit is available at https://github.com/firedancer-io/firedancer/. Specifically, the audit focused on the Firedancer v0.4 parts of the code mentioned above, excluding components such as the runtime, blockstore, consensus, and gossip components, which remain part of Agave. Throughout the audit, the Firedancer codebase underwent several changes. We regularly updated our analysis to include the latest commits, though only important changes were backported to v0.4. The relevant source code revisions for this audit are:

- `untagged versions` · Start of the audit
- `v0.403.20113` · First real v0.4 audited
- `v0.410.20113` · Version at end of audit

4 | High Level Changes

We began the audit with a high-level review of all changes introduced in version 0.4, coordinating the scope. As with previous audits, we prioritized components—referred to as tiles—that are network-reachable or handle untrusted user input, as these represent the most exposed and risk-prone parts of the system.

As in previous audits, it was clear that the codebase had already undergone substantial internal scrutiny. Since the high-level attack surface remained broadly consistent with earlier versions, this engagement emphasized the security implications of implementation changes, as well as the evolving Agave dependency — that could introduce new incompatibilities or unexpected behavior.

The following section outlines the most significant architectural and implementation changes relevant to this audit.

QUIC

The QUIC tile has undergone significant performance improvements in v0.4. It has also become more secure: the Firedancer team identified that the previously implemented `sign`-tile access was unnecessary, as server identity is not actively verified in practice. As a result, that pathway was removed.

We re-evaluated the QUIC implementation and found it still has some susceptibility to denial-of-service (DoS) attacks, compared to a very simple volumetric-DoS that is always possible, but much less so than before. See Findings ND-FD04-LO-01, ND-FD04-IN-01 and ND-FD04-IN-02 for details.

QUIC Fuzzing

We assessed the fuzzing and test coverage of QUIC and found it to be strong overall, with much of the coverage coming from unit tests. At the beginning of the audit, the available fuzzer was relatively shallow—it operated statelessly and only fuzzed individual functions, rather than full connection lifecycles.

We prototyped an improved fuzzer capable of handling full QUIC connections. Despite increased depth and state awareness, we did not identify any issues through this method. Another more advanced QUIC fuzzer developed by FD/AR exists but is not yet publicly available.

Net/XDP Rewrite

The network path of Firedancer was rewritten in a projected called net 2.0. The design, packet flow and security are thoroughly documented in the repository under `book/guide/internals/*.md` (Docs: Net Tile), so we omit the details here.

Plugins

Firedancer now includes a flexible plugin system. Many tiles can optionally emit messages to plugin consumers. Each tile-type has it's own plugin link: `poh_plugin`, `replay_plugi`... A lot of them aren't really used in the "Frankendancer" setup where Agave still provides the runtime.

A central `plugin` tile aggregates output from all participating tiles into a unified `plugin_out` link. This allows plugin consumers to access the variety of internal events—such as slot completions, gossip updates, validator state, balances, and block engine activity — without needing to interact with each tile individually.

All plugin configuration is hardcoded and requires recompilation to change. But the interface is set, and plugins don't have to touch other tiles directly to gather info. Right now, the only plugin consumer is the new GUI tile. It exposes the messages and metrics it receives from all over the validator over HTTP, so consumers can see them on an informative web UI.

Graphical User Interface (GUI)

Firedancer now has a GUI in form of a browser-based dashboard, hosted by the `fd_gui_tile`. The tile bundles static assets, and serves events and metrics over websockets.

The GUI tile connects to the global metrics metadata as well as the `plugin_out` link, allowing it to display information from across the validator. Since the GUI may be externally reachable, we assessed potential risks, including impact from potential remote code execution (RCE). While metrics are writable, they are not critical to core validator functionality. The other links are joined read-only, and the GUI is in it's own sandboxed process. One point of concern is the shared `fseq` in the metrics workspace: Finding ND-FD04-IN-03.

Bundles

One of the most significant new features in Firedancer v0.4 is support for **bundles**, specifically those provided by the Jito block engine. Jito supplies validators with atomic bundles—groups of transactions that must be replayed together. If any transaction within the bundle fails, none of them may be included in the block. Validators that correctly handle bundles receive additional rewards, facilitated through auxiliary "cranking" transactions that must be generated and included in the block by the validator itself.

While the concept of bundles is generic, the current implementation is tightly coupled to Jito-specific behaviors, as Jito is presently the only supported bundle ingress mechanism.

Firedancer's transaction-centric architecture posed challenges for integrating bundles. The solution was to maintain transaction-based processing, but attach metadata to each transaction indicating:

- Whether it is part of a bundle
- A bundle ID
- The total number of transactions in the bundle

This metadata enables the pack tile to reconstruct and schedule bundles appropriately.

To maintain correctness and reduce complexity, firedancer chose a design in which all bundles must progress through the validator pipeline in order. Specific tiles enforce this ordering — for example, only the first verify and bank tiles process bundled transactions. This avoids races where bundles are processed out of sequence.

Bundle execution introduces a unique requirement: **all-or-nothing execution**. If any transaction in a bundle fails — due to runtime errors or conditions like prior execution (e.g., replay attempts) — the entire bundle must be rolled back. This makes it impossible to know in advance if a bundle is valid without attempting execution.

Firedancer handles this with a `bundle stage` inside the Agave runtime, adapted from Jito's implementation. It executes the full bundle and commits only if all transactions succeed. This ensures compatibility with Jito's semantics and integrates cleanly with Firedancer's execution model.

The **ingress** mechanism for receiving bundles is completely decoupled from their execution. It is implemented using a bundle plugin, not to be confused with firedancers other plugin system the GUI uses!

The `bundle` ingress component, while sometimes referred to as a plugin, operates differently from the GUI and standard plugin system. It has a distinct interface, deeper integration, and is partially written in Rust. It resides under `/plugin/bundle/` and is integrated as a core part of the block processing pipeline, not merely a passive observer.

A Rust-based plugin connects to the Jito block engine server and feeds incoming bundles into the validator pipeline. This design is modular—other bundle providers could be supported in the future by implementing compatible plugins.

Many tiles include optimizations for bundles to improve performance and avoid unnecessary work. For example, if any transaction within a bundle fails during verification, the validator can immediately discard the remaining transactions in the bundle—they would be rejected downstream regardless.

The `pack` tile, which prioritizes and schedules transactions, also handles bundles intelligently. Normally, pack uses a **priority-sorted treap** to select transactions that maximize reward per unit of work. With bundles, it needs to preserve the atomicity of grouped transactions.

To solve this, `pack` applies a mathematical transformation — complete with a formal correctness proof — to insert bundle transactions into the treap in a way that:

- Preserves internal bundle order
- Maintains reward/cost optimization logic
- Allows efficient scheduling without violating atomicity

There are a couple edge-cases: if the validator is flooded with bundles faster than it can process them, a wraparound of relative indices could occur. However, because the number of active transactions in

the treap is bounded, this situation does not violate bundle execution guarantees. Just the “FIFO” nature of incoming bundles might get offset, single bundles will still be atomic and in the correct order.

Duplicate transactions are another. What happens if a transaction is both in a bundle, and a normal transaction as well? Especially if the normal transaction arrived first, and the bundle later. Should you drop the bundle, or the transaction? Firedancer’s approach is simple: You don’t drop anything. Bundles and Transactions have separate deduplications, and only once they reach the runtime will the final replay-protection check reject the second time the transaction would be executed. This is slightly unintuitive on first look, but since the runtime has the final say it works out.

Address Lookup Tables

Firedancer now supports constructing leader blocks that include Agave’s `VersionedTransactions`, which make use of **Address Lookup Tables (ALTs)**. Previously, Firedancer could only replay blocks containing ALT-based transactions, but could not include them during its own leader slots.

This capability is enabled by a new tile called `resolve`, which sits between the `dedup` and `pack` tiles in the transaction pipeline. The `resolve` tile receives deduplicated and verified transactions, then resolves any ALT account references to their actual addresses.

To perform resolution, the tile must access the relevant ALT accounts — meaning it must operate with the Agave runtime and thus address space. But the resolve tile is still positioned well before the execution runtime and remains only loosely coupled to it. As a result, Firedancer takes a conservative approach, favoring false negatives (dropping transactions it can’t confidently resolve) over false positives. This design choice is reasonable and consistent with Solana’s operational assumptions.

Edge Cases There are a few scenarios where ALT resolution might fail:

- **Recently Created ALTs**: A transaction might reference an ALT that was created very recently and is not yet available to the `resolv` tile. While such cases are rare — since ALT extensions require at least one full slot to activate — they can occur. In these situations, the transaction is dropped. It cannot be easily retried with the same validator due to deduplication filtering, but it may still be accepted by the next validator in the leader schedule.
- **Deactivated ALTs**: ALTs are subject to a deactivation cooldown. To avoid race conditions or external manipulation of in-flight transactions, they are only considered fully deactivated once they fall outside of the recent slot hashes window. Firedancer, for simplicity and performance, does not check slot hashes explicitly. Instead, it conservatively considers any ALT from the past 512 slots to still be valid. While this period may be slightly shorter than Agave’s acceptance period in fork scenarios, it is a safe approximation and fully consensus-compatible.

These differences are limited to transaction inclusion during block construction and do not affect consensus or transaction replay. As such, they pose no correctness issues within the Solana protocol.

Live Identity Changing

Firedancer now supports **live validator identity switching**, a critical feature for enabling hot-failover without incurring downtime. This functionality allows a running validator instance to update its identity (e.g., keypair) while maintaining operational continuity across all relevant subsystems. The identity switch is orchestrated by a new component called `keyswitch`.

Each tile that relies on the validator identity is now wired to an individual shared-memory `keyswitch` object. This includes:

- `plugin`
- `agave / poh`
- `gui`
- `pack`
- `shred`
- `sign`

These tiles monitor the `keyswitch` state during their housekeeping routines. When a change is detected, they initiate the local update process.

The `set_identity` command handles the identity switch through careful coordination and sequencing, ensuring no partial updates or inconsistencies occur. The steps are as follows:

1. **Pause Block Production:** The PoH tile is first locked, and a halt is requested to pause any potentially ongoing block production cleanly.
2. **Flush Shreds:** The shred tile is signaled to flush all shreds corresponding to the current block up to the point where PoH stopped.
3. **Update all Other Tiles:** Once shreds are flushed, the sign tile is updated with the new identity, along with all other tiles that subscribe to identity information via the `keyswitch` mechanism.
4. **Wait for Acknowledgment:** All tiles — excluding PoH — must confirm that they have completed their identity transition.
5. **Resume PoH:** Once confirmation is received, the PoH tile is unpaused. The shred tile does not need explicit unpausing, as it will automatically resume upon receiving blocks from the newly resumed PoH stream.

The most complex scenarios arise when the validator is actively leader — or will soon become one — at the time of the identity switch. In such cases, careful handling is required to avoid producing blocks under an outdated or inconsistent identity. The `set_identity` orchestration explicitly accounts for these scenarios by halting block production and flushing state prior to any identity transition.

Miscellaneous Changes

In addition to the major features described above, Firedancer v0.4 includes a number of smaller but noteworthy updates. This list is non-exhaustive and highlights selected changes of interest:

Tile Topography Refactor: The main topography between tiles was refactored. While functionally similar to previous versions, much of the underlying code of the inter-tile communication has moved or been slightly altered. The main abstraction layer now used is called `STEM`, replacing `mux`. It serves the same purpose—inter-tile communication — but uses macros instead of callbacks for efficiency and simplicity.

Shred Enhancements: The `shred` tile now supports receiving both Merkle shreds and resigned shreds. Outbound (sending) support for these formats is not yet implemented.

HTTP Server and WebSocket Layer: A custom HTTP server is now in-scope and is used by several tiles, including the GUI and metrics systems. The server features a custom WebSocket implementation and uses `picohttpparser` for HTTP parsing. As the GUI is expected to be publicly accessible for some validators, the HTTP server constitutes an externally exposed attack surface. Both the GUI and HTTP server components have been separately audited by Cure53.

Sandboxing Improvements: The validator sandbox now includes integration with the Linux Landlock API, further tightening the runtime's attack surface by restricting filesystem access at the kernel level.

Library and Cryptography Updates: Several supporting libraries have been added or improved:

- A new AES-GCM implementation
- Updates to lthash, bmtree, and BitInt
- Refactors and optimizations to the ed25519 cryptographic code

5 | Findings

This section outlines all of our findings. They are classified into one of five severity levels, detailed in Appendix B. All findings are listed in Table 1 and further described in the following sections.

Identifier	Name	Severity	Status
ND-FD04-LO-01	QUIC Handshake Denial of Service	LOW	Resolved
ND-FD04-IN-01	QUIC Spam with 1200 PING Frames per Packet	INFORMATIONAL	Acknowledged
ND-FD04-IN-02	Slowloris Attacks not entirely mitigated	INFORMATIONAL	Acknowledged
ND-FD04-IN-03	Flowcontrol of links can be written cross-tile	INFORMATIONAL	Acknowledged
ND-FD04-HI-01	Unwritable Accounts List Incomplete	HIGH	Fixed before report

Table 1: Findings

[ND-FD04-LO-01] QUIC Handshake Denial of Service

Severity	Impact	Affected Component	Status
LOW	Attacker could block others from sending TX to Validator	QUIC Server	Resolved

Per default, Firedancer allows up to 4096 handshakes to occur simultaneously. This is fairly low, because handshakes are inherently very expensive.

Internally, whenever a new handshake is initiated, a `fd_quic_tls_hs_t` object occupies an additional slot in the object pool structure `state->hs_pool`. From the perspective of the server, a new connection and subsequently a new handshake slot is occupied when the server receives an initial packet that cannot be linked to an existing connection (ignoring retry protection for simplicity here). The handshake pool slot stays occupied until the handshake is completed.

A malicious client may choose to never complete the handshake process while preventing its connection from being closed due to an idle timeout at the same time. It can do so e.g. by sending further 1-RTT packets consisting of PING frames only. Doing this for 4096 connections at the same time is cheap and prevents any other legitimate client from executing a full handshake.

If an attacker manages to block all connections except for his own, he has free-reign to select what transactions get included in a block, by simply just sending those to the Validator.

Recommendations:

- Limit the amount of connections per IP (mitigates slowloris attacks as well, recommended by the standard)
- Optionally implement a handshake time out that is separate from the idle timeout
- separate connection limit for “authenticated” IPs (known good IPs from other RPC nodes and validators).

Resolution

The Firedancer Team resolved this in PR#4669 quic: tls hs eviction, by allowing the oldest incomplete handshake to be evicted when no handshake-slots are available.

[ND-FD04-IN-01] QUIC Spam with 1200 PING Frames per Packet

Severity	Impact	Affected Component	Status
INFORMATIONAL	QUIC unexpected CPU Usage	QUIC Server	Acknowledged

While investigating resource exhaustion on the QUIC tile, we thought about the following theoretical approach:

QUIC Frame parsing is somewhat expensive, since every frame has to run through the parser and call the respective handler function. The smallest message you can fit into a frame is a `PING` message with a 1 byte frame size.

Implementing this in a simple QUIC client is straight forward. For testing we disabled the QUIC crypto in firedancer `FD_QUIC_DISABLE_CRYPTO`, so some encryption overhead might apply when implementing this in a full blown QUIC stack.

Implementing this attack with a local sender and `netns`, we can utilize the QUIC tile to 99.98% with a bandwidth of 1.8 GBit/s on a single client. For a benchmark how this compares to a "stupid" QUIC DoS, see Benchmark below.

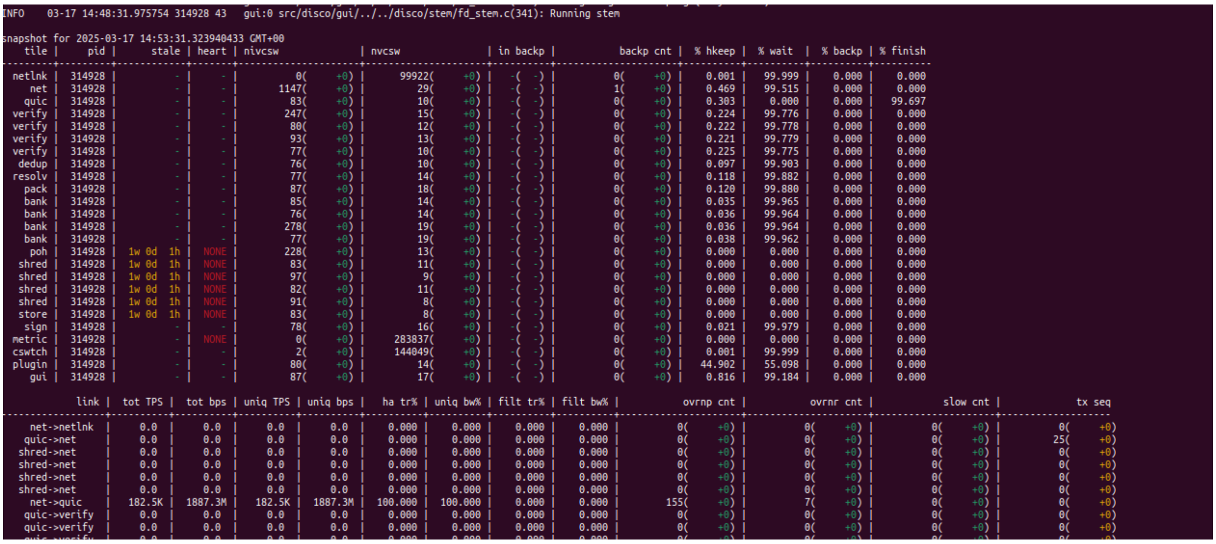

Figure 2: Tile usage when PING spammed. Note the QUIC tile utilization.

Implementation

- Client: INITIAL
- Server: HANDSHAKE
- Loop:
    - Client: 1-RTT Paket with CID from handshake, 1200 PING frames (don't receive answer, just spam)

Implementing this is possible in python with a custom patched `qh3` library that skips the QUIC encryption.

Benchmark

Comparing to a "stupid" UDP DoS with random packets (after a valid `INITIAL <=> HANDSHAKE`) sequence, we can conclude:

1. Optimized Spam: 1.8 GBits, 99.98% utilization of QUIC tile
2. "Stupid" Spam (just random bytes): 5.8 GBits, 15% utilization of QUIC tile

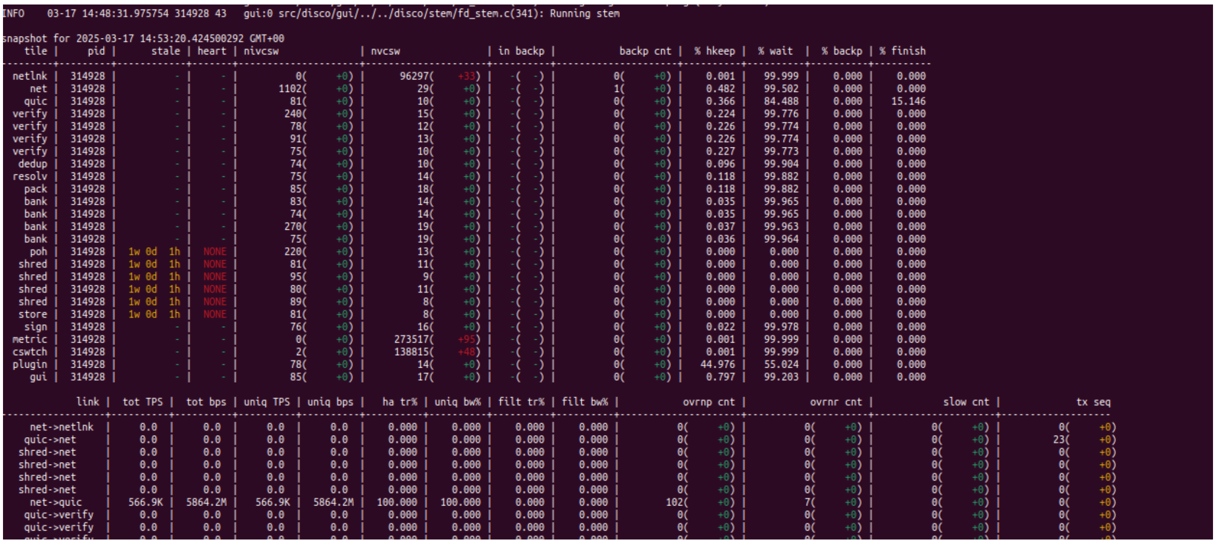

Figure 3: Spam with just random bytes in QUIC message (get discarded during parsing)

Possible Solutions:

- Limit number of frames per UDP packet.
- Implement quick path for frames that don't get consumed anyway.

[ND-FD04-IN-02] Slowloris Attacks not entirely mitigated

Severity	Impact	Affected Component	Status
INFORMATIONAL	Without additional Firewall, Firedancer could be attacked to not include user-transactions in produced blocks	QUIC Server	Acknowledged

The finding `ND-FD1-MD-03` from the previous audit describes a slowloris attack, where malicious clients could exhaust all available connections slots in order to prevent others from establishing new connections with the QUIC server.

Since then, only the maximum amount of connections per QUIC tile has increased from `2048` to `131072` forcing an attacker to create more connections (and prevent them from timing out too). Preventing a time out is relatively cheap because one only has to send e.g. one INITIAL packet every 10 seconds (based on the default idle time out) to keep a connection alive.

Benchmark

Our prototype establishes a connection using an initial packet and then subsequently sends a 26-byte (excluding IP and UDP headers) 1-RTT QUIC packet once every 5 seconds (default idle timeout divided by 2) to keep all `131072` connections alive. The packet had to be 26 bytes in total because QUIC packets with payloads less than 16 bytes get ignored.

The connection limit can be fully exhausted using less than 20 MBits.

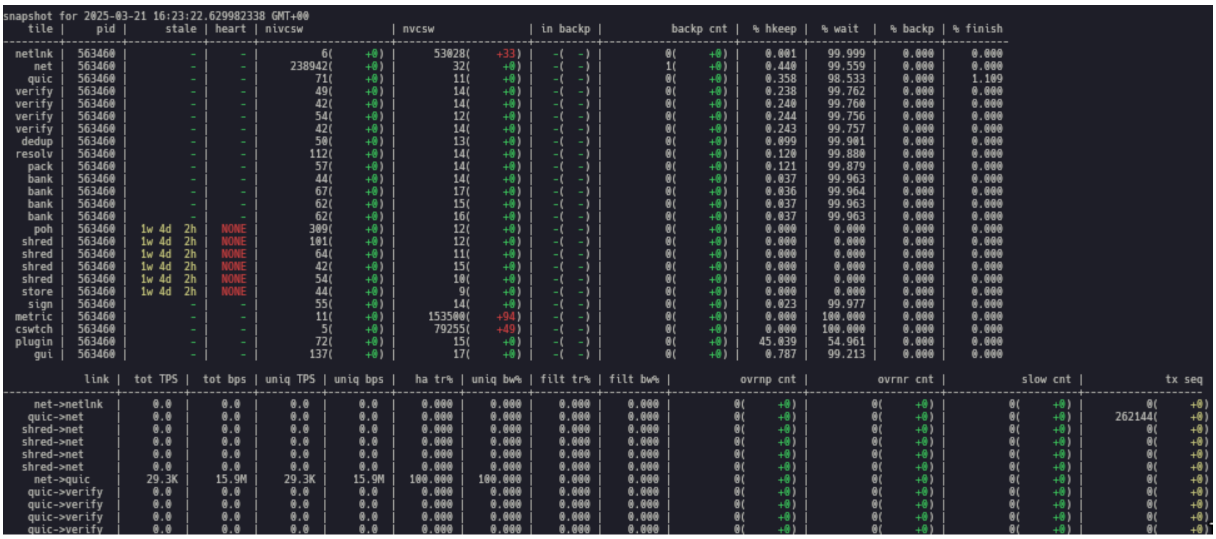

Figure 4: Tile usage while keeping 131k connections open

Note: We disabled the concurrent handshake limit because due to the way our prototype works it would run into the Handshake DoS issue before exhausting the connection limit. We also disabled retry protection for simplicity.

Recommendations:

- Last time: “A passable solution is implemented by the agave client: Agave allocates quic connections based on stake-weights of peers. If all connections are allocated, existing connections are dropped based on stake weights whenever new connections are required.”
- Also: limit number of connections per IP (recommended by the standard too)

[ND-FD04-IN-03] Flowcontrol of links can be written cross-tile

Severity	Impact	Affected Component	Status
INFORMATIONAL	Most tiles can slow Validator	Topology	Acknowledged

The links between tiles are implemented using shared memory. The data is inherently one-way, so a producer write-maps the dcache and mcache regions, while a consumer maps them read-only. But to achieve flow-control and back-pressuring, there has to be a return-path, of at least the sequence number the consumer is currently at. This is implemented with `fseq`'s. Shared-memory, atomically-written-to sequence numbers.

They currently **all** live in the **metric_in** workspace. Any tile that is allowed to write the `metric_in` workspace, which is all tiles that produce metrics, so all of them, can thus also write **all** sequence numbers of any link between any two tiles, even if that link isn't connected to the tile at all.

Take for example the `verify_dedup` link. It is it's own workspace, and created in `topology.c`:

```
1  // create workspace for link (stores actual data mcache/dcache)
2  fd_topob_wksp( topo, "verify_dedup" );
3
4  // create link. this creates m/dcache objects in the specified workspace
5  fd_topob_link(     topo,     "verify_dedup",     "verify_dedup",     config->...,
   FD_TPU_PARSED_MTU, 1UL );
6
7  // Join verify tile as output to this link. this joins VERIFY tile as read-write
   to the links m/dcaches
8  fd_topob_tile_out( topo, "verify",  i, "verify_dedup", i );
9
10 // Join dedup tile as input to this link. this joins DEDUP as read-only to the
   links m/dcaches
11 // It also allocastes a new fseq object in the metric_in workspace, and saves
   that to tile->in_link_fseq_obj_id.
12 // It later joins the metric_in workspace as read-write!
13 fd_topob_tile_in(  topo,  "dedup",      0UL,  "metric_in",  "verify_dedup",  i,
   FD_TOPOB_RELIABLE,   FD_TOPOB_POLLED );
```

Since a tile is able to modify the flow-control of all other tiles, it can simulate back-pressure upon which producers might stop producing. Many tiles in the 'hot-path' can have similar effects by simply stopping their work, but especially for "public facing" tiles such as the new GUI tile, this is an unnecessary vector in which a GUI-tile compromise could affect the whole validator.

One clean way to fix this, would be a separate workspace between all tiles just for this fseq backchannel, with appropriate read-write permission on each tile.

[ND-FD04-HI-01] Unwritable Accounts List Incomplete

Severity	Impact	Affected Component	Status
HIGH	Firedancer leader forks from the network	Account Locking	Fixed before report

A serious bug was identified in Firedancer's account locking logic, which could lead to validator forks and potential network stalls. The issue stems from an incomplete set of reserved account keys in the `fd_pack_unwritable` table, specifically missing entries introduced by the `add_new_reserved_account_keys` feature (`8U4skmMVnF6k2kMvrWbQuRUT3qQSiTYpSjqmhmgfthZu`) activated on February 16th. These keys are correctly handled in Agave (via write-demotion), but Firedancer instead drops such transactions during packing.

Without the full reserved account list, Firedancer may incorrectly treat reserved accounts as writable, leading to consensus divergence. We demonstrated this with a proof-of-concept transaction that successfully transfers lamports to a reserved account (`Sysvar1111111111111111111111111111111111111`). While Agave nodes reject the transaction, Firedancer includes it and transitions to a different state hash.

In setups where Firedancer has majority stake, this causes all Agave nodes to panic and exit due to irreconcilable state. In more typical configurations, the faulty leader forks off and continues proposing blocks alone, which will not receive votes due to invalid state. Recovery requires manual intervention (e.g., snapshot restore or ledger pruning).

The root cause lies in assumptions made in `fd_pack.c`, where the writable-account cache trusts packing logic to reject reserved writes, without accounting for newer reserved keys. **This vulnerability had already been addressed upstream before we reported it** in PR #4526, which was also backported to v0.4 and released shortly after.

Technical Details

The reserved keys in Agave are defined in `reserved_account_keys.rs`. (or in the split out `agavesdk` in the reserved-account-keys crate). With PR #4526 applied the issue is fixed.

If an account is marked as reserved, it cannot be used as a writeable input to a transaction. Agave is "nice" and write-demotes those accounts, executing the transaction as-if the account would be specified as read-only. Firedancer instead drops those transaction during pack. Due to the way the plumbing to the bank works, the "writelock demotion" path is skipped for frankendancers. The drop in pack is thus critical for consensus. If a transaction reaches the bank without having write-locks demoted, writes will be allowed. We have a PoC that transfers some lamports to a reserved account. If a frankendancer proposes a block including such a transaction, it will get to a different state-hash as all other replaying validators, who still run agaves writelock demote flow. For agave validators, the TX will be included as failing (still paying fees), whereas the frankendancer leader successfully executes it. Since a leader will never repair his own block, it is stuck until manual intervention. (or, in our testsetup where it has majority stake, all other nodes will panic since they can't reproduce the majority state).

An Agave `SanitizedTransaction` has a boolean-cache which accounts are writable. Initialized on `try_new`. This demotes programs and reserved account writability "on the fly". The actual read/writability of account is not changed, but the cache is used in `is_writable`, which the runtime trusts. Firedancer does not use `try_new`, instead, in `fd_ext_bank_load_and_execute_txns`:

```
    let txns = unsafe {
       std::slice::from_raw_parts(txns as *const SanitizedTransaction, txn_count as usize)
    };
```

where the txns are previously initialized with `fd_bank_abi_txn_init`. This is pure firedancer code, and contains

```
for( ushort i=0; i<txn->acct_addr_cnt; i++ ) {
  int is_writable = fd_txn_is_writable( txn, i ) &&
                    /* Agave does this check, but we don't need to here because pack
                       rejects these transactions before they make it to the bank.

                     !fd_bank_abi_builtin_keys_and_sysvars_tbl_contains( (const fd_acct_addr_t*)(payload + txn->acct_addr_off + i*32UL) ) */
                                      (!is_key_called_as_program(  txn,  i  )  || _is_upgradeable_loader_present);
  legacy->is_writable_account_cache[ i ] = !!is_writable;
}
```

Thus the writable account cache does NOT demote writable reserved-accounts, instead it claims that pack rejects them before in comment. Indeed, the validate_transaction call does

```
for(        fd_txn_acct_iter_t        iter=fd_txn_acct_iter_init(        txn, FD_TXN_ACCT_CAT_WRITABLE );
    iter!=fd_txn_acct_iter_end(); iter=fd_txn_acct_iter_next( iter ) ) {
  writes_to_sysvar |= fd_pack_unwritable_contains( ACCT_ITER_TO_PTR( iter ) );
}
// ...
if( FD_UNLIKELY( writes_to_sysvar                                        ) )
return FD_PACK_INSERT_REJECT_WRITES_SYSVAR;
```

where `fd_pack_unwritable` is essentially a hashset that contains blacklisted keys. But that list is incomplete since activation of `add new unwritable reserved accounts` #34899 in mid-February. For example, `Sysvar1111111111111111111111111111111111111` is not reserved in Firedancer. If we send a transaction that transfers lamports to `Sysvar1111111111111111111111111111111111111` (thereby writing to it), Firedancer includes it in a block and executes it successfully, while the same transaction fails in agave. This leads to mismatched blockhashes between the Firedancer validator proposed the block and the rest of the network, since the leader modifies the account while nodes that replay do not

(all nodes have agave-based replay, which does the write-demotion “correctly”). In a localnet where a Firedancer-leader has majority-stake localnet, all followers thus crash with

```
We have tried to repair duplicate slot: 561 more than 10 times and are unable to freeze a block with bankhash `7M6wpYo7KbGzpVtFeBLCjbWz8p48HkRNFHPHDhATT7X2`, instead we have a block with bankhash `Some(98Mb8HXPa9s5TkAdH389GqgvNeWuPE4bpNHhFNDmF5dq)`.
This is most likely a bug in the runtime.
At this point manual intervention is needed to make progress.
Exiting"  location="core/src/replay_stage.rs:1550:25"  version="2.1.13 (src:00000000; feat:1725507508, client:Agave)"
```

We haven’t tested impact on “realistic” stake distributions, but assume that this just fork off the leader. The block is fully valid after-all, just all subsequent votes on this block from the proposing-validator will not be, since they vote on wrong state. Since leader stays forked off until manual restart and snapshot-pull, or manual deletion of the offending block from the ledger, this could be used to fork off vulnerable leaders one-by-one.

A | About Neodyme

Security is difficult.

To understand and break complex things, you need a certain type of people. People who thrive in complexity, who love to play around with code, and who don't stop exploring until they fully understand every aspect of it. That's us.

Our team never outsources audits. Having found over 80 High or Critical bugs in Solana's core code itself, we believe that Neodyme hosts the most qualified auditors for Solana programs. We've also found and disclosed critical vulnerabilities in many of Solana's top projects and have responsibly disclosed issues that could have resulted in the theft of over $10B in TVL on the Solana blockchain.

All of our team members have a background in competitive hacking. During such hacking competitions, called CTFs, we competed and collaborated while finding vulnerabilities, breaking encryption, reverse engineering complicated algorithms, and much more. Through the years, many of our team members have won national and international hacking competitions, and keep ranking highly among some of the hardest CTF events worldwide. In 2020, some of our members started experimenting with validators and became active members in the early Solana community. With the prospect of an interesting technical challenge and bug bounties, they quickly encouraged others from our CTF team to look for security issues in Solana. The result was so successful that after reporting several bugs, in 2021, the Solana Foundation contracted us for source code auditing. As a result, Neodyme was born.

B | Vulnerability Severity Rating

We use the following guideline to classify the severity of vulnerabilities. Note that we assess each vulnerability on an individual basis and may deviate from these guidelines in cases where it is well-founded. In such cases, we always provide an explanation.

Severity	Description
CRITICAL	Vulnerabilities that will likely cause loss of funds. An attacker can trigger them with little or no preparation, or they are expected to happen accidentally. Effects are difficult to undo after they are detected.
HIGH	Bugs that can be used to set up loss of funds in a more limited capacity, or to render the contract unusable.
MEDIUM	Bugs that do not cause direct loss of funds but that may lead to other exploitable mechanisms, or that could be exploited to render the contract partially unusable.
LOW	Bugs that do not have a significant immediate impact and could be fixed easily after detection.
INFORMATIONAL	Bugs or inconsistencies that have little to no security impact, but are still noteworthy.

Additionally, we often provide the client with a list of nit-picks, i.e. findings whose severity lies below Informational. In general, these findings are not part of the report.

Neodyme AG

Dirnismaning 55
Halle 13
85748 Garching
Germany

E-Mail: contact@neodyme.io

https://neodyme.io