Security Audit – Firedancer v0.1

conducted by Neodyme AG

Auditor:	Alain Rödel
Auditor:	Felipe Romero
Auditor:	Konstantin Bücheler
Auditor:	Nico Gründel
Auditor:	Ruben Gonzales
Auditor:	Simon Klier
Auditor, Administrative Lead:	Thomas Lambertz

June 30^{th} 2024

Table of Contents

Executive Summary

Neodyme was contracted to do a time-boxed audit of **Firedancer's** first milestone version 0.1, during the end of April, May and June 2024.

The audit covers functionality, design, attack surface, and sandbox isolation. Potential exploit paths and their impact were investigated, patches to Agave were reviewed for robustness, and the interaction between Agave and Firedancer during leader slots was analyzed. Additionally, the four custom tiles (sign, quic, pack, and shred) and the internal data structures used by Firedancer were examined.

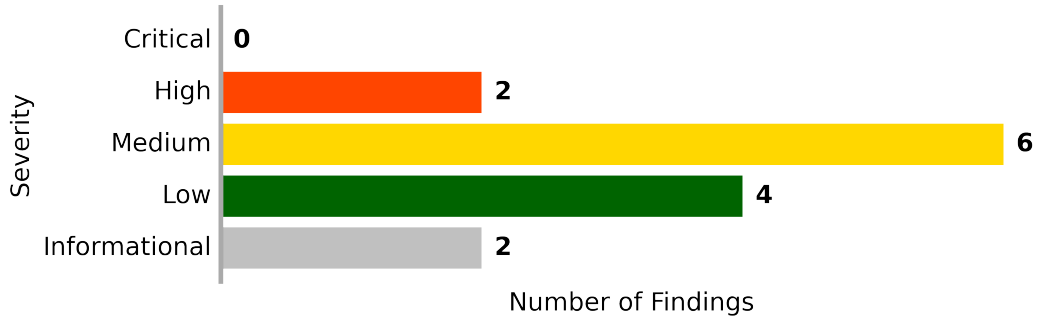


Figure 1: Overview of Findings

No remote code execution vulnerabilities were found. The findings mainly involved denial-of-service in various forms and **mismatches between Agave and Firedancer** implementation behavior. No critical vulnerabilities were identified. Two high-severity vulnerabilities were found, one of which would allow an attacker to crash Firedancer remotely.

Overall, Firedancer was found to be **well-designed**, with most issues stemming from the lack of a solid Solana validator specification, leading to slight mismatches in behavior. Firedancer's developers employ a defense in depth approach, of which the excellently implemented sandboxing is an integral part.

1 Introduction

Jump Crypto engaged Neodyme to do a detailed security analysis of their Firedancer Solana validator – specifically, the first Milestone, Version 0.1. The audit was conducted by seven Senior security researchers, each focusing on different aspects, between April 22nd and June 30th 2024. Multiple auditors have a long track record of finding critical and other vulnerabilities in both Solana programs and Validator implementations on multiple chains. Combined, they have prevented the theft of at least $1b worth of tokens through reports of critical bugs.

Firedancer, a second implementation of a Solana validator client, still relies heavily on the original client, Agave. Custom components include networking, transaction ingress and validation, and much of the leader functionality around block packing.

This report first describes relevant parts of Firedancer's design, providing a security-focused overview of its architecture. Using this foundation, an overview of potential attack surfaces is presented and the sandboxes for individual functionalities are investigated.

The report examines the potential impact of remote code execution (RCE) in various components and considers how an attacker might move between sandboxes. The patches to Agave are briefly investigated, highlighting the interaction between Agave and Firedancer during leader slots.

The four most security-critical functionalities – Signing, Quic, Pack, and Shred – are examined in-depth, followed by a review of the data structures used. All findings and their mitigations are presented.

2 | Scope

The scope of this audit can be succinctly summarized as encompassing everything relevant to Firedancer v0.1. Given the time-boxed nature of the audit, efforts were made to be time-efficient, with the focus shifting over time based on initial findings.

Audit Approach

We began with an in-depth examination of the underlying communication architecture. It was apparent that many of the individual components either had alread undergone significant scrutiny, or were too reliant on the caller integration to audit on their own. Finding them to be robust, we then moved on to explore higher-level, functional attack vectors. Our primary focus shifted to identifying bugs that could cause practical issues both in the current implementation and in future iterations of Firedancer once it operates independently of Agave.

Key Focus Areas

One main areas of focus for the audit were thus the functional tiles, namely Net, Quic, Verify, Dedup, Pack, Shred, Sign, Metric, Bank, Poh and Store. Other important areas were:

- Behavior differences between Firedancer and Agave
- Sandbox implementation and hardening
- Cryptography
- The Agave FFI interface

Source Code Revisions

The source code located at https://github.com/firedancer-io/firedancer/ is in scope for this audit. Specifically, the audit focused on the Firedancer v0.1 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. The relevant source code revisions for this audit are:

Relevant source code revisions are:

- `89da411981808437f094de2682e70ee8d7e44d52` • Start of the audit
- `fdd89735be64a4d8e14e8db27b85630f52c473b1` • Last revision considered

3 | Firedancer v0.1 Design

The Solana network relies on validators to maintain the network's state, verify transactions, and facilitate the proof-of-stake consensus mechanism. Currently, the only existing validator implementation is the Agave validator, developed by Anza and formerly referred to simply as 'validator' by Solana Labs.

Firedancer will be an alternative validator client, maintaining compatibility with the existing network. As such, it is imperative that Firedancer replicates the functional nuances of the Agave validator. Presently, Agave's implementation is entirely defined by its existing framework, lacking a formal specification. This absence poses significant challenges in developing a compatible alternative.

The initial phase of Firedancer's development, sometimes referred to as Firedancer v0.1, introduces partial replacements in the Agave validator infrastructure. It substitutes major components like the networking stack, proof-of-history, and most leader functionalities, including block-packing. However, other critical components such as storage, runtime, replay, and consensus continue to utilize the original Agave code.

Notably, Firedancer is programmed in C, diverging from Agave's use of Rust. This strategic choice in programming language aims to minimize the potential impact of compiler bugs and language-specific behaviors on network stability. Given the priority for high performance, C was selected as the most suitable programming language for Firedancer.

Detailed Firedancer Architecture

Firedancer adopts a very different architectural approach compared to Agave, favoring a more streamlined and explicitly defined dependency structure. Operations in Firedancer are segmented into tasks housed within 'Tiles'. These tiles operate on shared memory areas, termed 'workspaces', which facilitate high-performance, optionally flow-controlled memory queues, known as 'Links'.

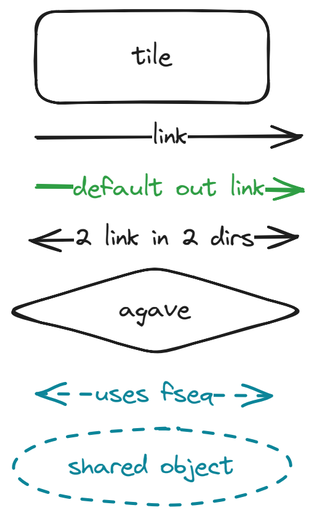
Most tiles are sandboxed, each running in a separate process. This initial setup does a lot of work to ensure that each tile can only access the required shared memory with the correct read or write permissions. The exception from this is the Agave part, which remains largely unconstrainted.

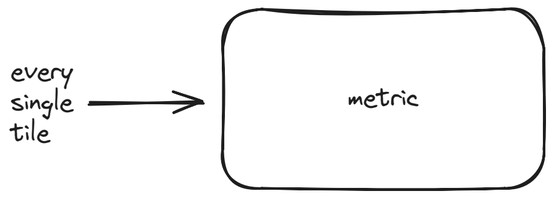
Firedancer is composed of ten primary tiles, each serving distinct functions:

- **Sign (aka Keyguard)**: Manages sensitive key material, providing signing services to other tiles.
- **Net**: Handles network ingress and egress.
- **Quic**: Processes QUIC protocol traffic from the Net tile.
- **Verify**: Verifies signatures and contents of incoming transactions.
- **Dedup**: Removes duplicate transactions.

- **Pack**: Packs blocks when the validator is a leader.
- **Bank**: Processes blocks from the Pack tile using the Agave runtime and updates the state.
- **PoH (Proof of History)**: Manages Solana's proof-of-history component.
- **Shred**: Splits blocks created by the runtime into fragments (shreds), and reassembles shreds coming in from the network.
- **Store**: Connects to the Agave blockstore for shred storage.

Here is a full architecture diagram, including all links between different tiles.





INCOMING: network packets from web (net_rx_aio_send).
Sanity checks ipv4/udp, branches on Port:
- quic -> redirect to quic tile,
- legacy UDP -> redirect to quic tile
- shred -> redirec to shred tile

INCOMING, from shred: shreds -> send

INCOMING, from quic: quic-packets -> send

INCOMING: quic or legacy packet from net
- round-robins based on "hash(source_ip)+port"
- if quic, does quic engine + callbacks (eg quic_stream_receive).
- Each quick stream gets reassembled into one TX. Then forwards TX to verify
- if legacy, does legacy_stream_notify, which forwards TX to verify

Spins on sign tile during quic conn establishment.
Before forwarding, appends 'descriptor' to tx: append_descriptor() (essentially a 'parsed' tx)

INCOMING: [raw solana transactions (V1/2), descriptor (parsed repr of tx), len of orig_tx]
round-robins on sequence-id
verifies length, trusts parsed TX otherwise, verifies signatures.
All TXs that are valid get passed on to dedup.
each tile has own dedup tcache via first 8 tx-sig bytes. Only *new* successful sigs are passed on.

INCOMING: same raw tx as verify, but knowing sig checks out
uses local tcache for dedup, trusting the same ha_dedup_tag verify generated and passed on.
Set mcache sig tag to 0, so pack can determine source of TXs

INCOMING: Transactions from Dedup and Gossip. And 'leader' notifications from PoH.
- Gossip: Parse tx, if parses include in microblock
- Dedup: Just include in microblock
Has: 'Extra Queue'
Has: Packing algo. Has set amount of CU for votes.
When leader: Packs microblocks, respecting CU/locking/etc.
When Microblocks are full (or timed-out), forwards them to bank (if one is idle)
Sends packing-done messages via pack_bank link to PoH

INCOMING: Microblocks from pack.
- Pack chooses which bank-tile to send a microblock to
- Chosen bank executes microblock:
- fd_bank_abi_txn_init -> convert TXs into solana bank format
- fd_ext_bank_load_and_execute_txns -> try to execute transactions
- fd_ext_bank_commit_txns -> commit TXs to solana store
- hash transactions
- publish microblock plus tx-hash to PoH

INCOMING: microblocks for executed TXs from bank
- runs the PoH hash-chain synchronously (in after_credit)
- keeps track of empty 'skipped slot' ticks, in case they're needed when becoming leader
- if leader, publishes (empty) tick
- if leader, publishes microblocks in after_frag (from bank).
- This also unlocks the used accounts.
- only contains the 'executed' transactions (ie no lock failure)

shred_version

INCOMING:
- shreds from NET -> redistribute
- batches from PoH -> genreate shreds, FEC, publish to net.
- once a full set -> publish to blockstore
- CRDS contact data -> for contact info on where to send shreds
- stake_in -> for weighted distribution on where to send shreds
- sign: for signing shreds

Publishes fd_shred34_t to store, shreds to net.

INCOMING:
- shreds+FEC from shredder.
- publishes shreds to agave blockstore with `fd_ext_blockstore_insert_shreds`

4 Attack Surface Overview

With this rough understanding of what Firedancer v0.1 does, we can roughly think about the attack surface.

Remote Code Execution (RCE)

Given that Firedancer v0.1 is implemented in C, a language known for its susceptibility to bugs like buffer overflows that can lead to RCE, this represents a significant risk, much more so than for Agave. Firedancer aims to be very high performance and hence does not do automated bounds-checks on everything, just where required. An RCE vulnerability could allow an attacker to execute arbitrary code on the validator, potentially compromising the entire node. The exact impact is investigated in more detail later in this report in Section Exploit Investigation.

Denial of Service (DoS)

Firedancer v0.1 might also be vulnerable to DoS attacks, which could render the validator slow or unresponsive. This might happen via resource exhaustion attacks, where an attacker deliberately consumes excessive computational or memory resources. It may also occur due to errors such as null-pointer dereference, which would crash the tile and, with it, the whole validator. Firedancer v0.1 lacks the capability to autonomously recover from crashes in any of its tiles, which means any crash could cause node unresponsiveness. While some components mainly impact the validator's leader capabilities, others could disrupt overall validator operations, including voting.

Behavior Mismatches with Agave

Discrepancies between Agave and Firedancer v0.1 present a significant area of concern. Divergences in block or transaction acceptance between the two could disrupt network consensus. We operate under the assumption that Agave, as the existing standard, correctly validates transactions and blocks.

Non-critical discrepancies during transaction handling are manageable; Firedancer v0.1 may simply limit some network functionalities, such as not accepting all transactions into blocks it creates as leader. However, divergences in block acceptance and creation are more problematic.

If **Agave creates and accepts blocks that Firedancer v0.1 rejects**, Firedancer nodes might not be able to replay the state, and diverge from the network, potentially breaking network consensus. However, because Firedancer v0.1 only introduces a subset of new functionalities, the risk associated with such mismatches is somewhat limited. Most of Firedancer v0.1's runtime, including the full replay stage and repair mechanisms, are inherited from Agave. This ensures that the Agave component

of Firedancer v0.1 retains full capability for fetching missing shreds, processing them according to Agave's logic, extracting transactions, processing them in the bank, replaying transactions, generating votes, and distributing these votes via gossip. The only aspect of the non-leadership chain influenced by the newly implemented Firedancer code involves certain Proof of History (PoH) utility functions. Importantly, PoH verification of replayed transactions remains a function of the Agave side, though all leader-required PoH functionalities, such as resetting PoH when consensus advances or when switching the current bank, are managed by Firedancer. Given this reliance on Agave for essential checks and balances, small anomalies in Firedancer v0.1's behavior often do not lead to practical issues. For example, even if Firedancer v0.1's net or shred-reassembly components erroneously drop valid data, Agave's independent repair pathways can rectify these errors, preventing the validator from falling of the network.

In the context of Firedancer v0.1/Milestone 1, a more pressing practical issue is **erroneously accepting data that should be rejected**, subsequently forming a block that the rest of the network will refuse. It is likely that even other Firedancer v0.1 nodes will reject the block, with the only big replay differentiator being shred-reassembly. However, while the leadership functionality is crucial for individual validators and impacts the number of transactions Solana can process, it is not immediately critical to overall network operation, as other validators will reject an invalid block. Theoretically, producing an invalid block is a slashable offense, which should result in the validator's stake being slashed. However, there is currently no automated protocol in place for this enforcement.

To summarize, since pretty much all of gossip, replay and repair is done in Agave, it is quite hard for practical network issues to appear when Firedancer v0.1 behaves weirdly. For replay, all checks are done in Agave, and if the net-/shred-reassembly-component of Firedancer v0.1 drops valid data, Agave can repair it via a fully independent code path.

Additionally, as long as only few nodes in the network run Firedancer v0.1, the network will survive most problems relating to it. As soon as a significant amount of stake is delegated to Firedancer v0.1 nodes, the Solana network will halt on RCE/DOS/Fork. In theory this limit is 33%, as at least 2/3 of stake needs to agree on a fork for it to be rooted. Indeed, in the past we've seen 10% of the network go offline without significant issues. However, in practice, the network will be considerably impacted well before the exact 33% boundary is reached.

Bugs in Agave

Finally, vulnerabilities present in Agave could also affect Firedancer v0.1. This aspect, however, falls outside the scope of this audit.

Recommendation for Agave Compatibility

As highlighted in the section above, it is crucial that both Firedancer and Agave accept and reject the same blocks and transactions. The necessary checks for this are not explicitly documented and are distributed throughout the Agave codebase. Firedancer implements these checks as well, but often in a different format. For example, transaction sanitization checks that are consolidated in one place in Agave are distributed across different tiles in Firedancer. Understanding why certain checks are implemented in specific places requires substantial knowledge of Agave's internals.

Despite our efforts to identify all discrepancies, it is likely that some have been missed. The distributed nature of these checks complicates the process of ensuring consistency between Agave and Firedancer.

Recommendation

We recommend adding code references, preferably using permanent links, in all places where Firedancer performs checks that have to be equivalent to Agave. This will provide clarity on the necessity of each check and make it possible to verify consistency between Agave and Firedancer. It will also make confident maintenance to the codebase possible.

5 Sandbox Evaluation

To mitigate the risks associated with Remote Code Execution (RCE) vulnerabilities, Firedancer v0.1 employs a robust sandboxing strategy. Its design is quite remarkable, locking down the processes to the furthest extent possible and hence limiting the potential damage from exploits.

Firedancer v0.1 has two kinds of processes, normal tiles and the Agave sandbox, which differ significantly.

Tile Sandboxes

In production mode, all tiles, except for the tiles interacting directly with Agave, run in their own process with their own sandbox, setup with `fd_sandbox` in `fd_topo_run_tile`. For development purposes, there is also a mode that uses threads instead of processes along with significantly less sandboxing. Due to relevance, however, this mode was not investigated in more detail.

The initialization sequence for each tile process is complex, as it has to ensure the greatest possible separation and sandboxing, while at the same time still connecting the tiles. By spawning new processes for each tile, Firedancer reaps the full benefits of address-space-layout-randomization (ASLR). For performance reasons, Firedancer uses huge pages as stacks, so it cannot simply `execve()` into a new process. One main process thus spawns several child processes via `execve()`, and then `clone()`s them to a process using a huge-page-backed stack. This new process then first memory-maps all required shared memory with other tiles, runs a `priviledged_init` function specific to the tile, then drops all privileges and sandboxes itself.

The exact sandbox process is well-documented in `fd_sandbox.h`, so we refrain from repeating all details here. Notably, the sandbox includes:

- A whitelist of open file descriptors
- Dropping privileges to a specified UID/GID
- Unsharing almost all namespaces, including the mount-namespace
- Prohibiting core-dumps
- Clearing the capability bounding set
- Removing environment variables

Post-initialization, the process becomes highly restricted, limited to operations on shared memory and predefined system calls.

System Call Filtering with Seccomp

A seccomp filter further narrows the system call surface to a minimal set, tailored to the needs of each tile. The implementation details are defined in custom domain-specific language (DSL) files named `.seccomppolicy`. Comments in these files justify the capabilities granted or restricted.

The granted permissions are mostly just writing to the logfile-file-descriptor. In addition, the network tile has very limited send and recv syscalls allowed, and quic can call `getrandom`.

The tile with the broadest permissions is the metrics tile, which hosts an HTTP server and thus requires permissions to read, write, close, and poll various file descriptors, dictated by the needs of an HTTP connection. However, this does not constitute a security issue: the only way to get new file descriptors is via the whitelisted `accept` syscall passing new incoming connections, and there is a sanity-check that there are no unwanted open file descriptors when the sandbox is initialized.

The quic tile is allowed to call `getrandom`, which in theory would allow it to exhaust the system randomness entropy, potentially blocking other consumers of randomness on the system.

Overall, this is a *very* strong sandbox design, that restrics almost everything down to syscall arguments, and everything required is individually whitelisted.

Agave Sandbox

There is one significant exception to the sandboxing approach: the Agave validator process. Agave wasn't originally designed with sandboxing in mind and is challenging to restrict effectively.

Three of Firedancer v0.1's tiles need direct access to Agave functionality. Specifically, the `store`, `bank`, `poh` tiles. To facilitate this access, they are run as threads within the Agave process and address space. Each of these tiles has full read/write capabilities into any other tile or link, effectively functioning as a single security boundary.

A partial sandbox is still implemented, although it is much more limited compared to Firedancer v0.1's internal sandboxing. Each tile in Agave's address space is spawned as a thread with minimal sandboxing measures, involving only changes to UID/GID.

Outside of this unsandboxed area, the most vulnerable security aspect of the sandbox is the shared-memory links between tiles, which will be examined in more detail in Section [Shared Memory Between Tiles][#shared-memory-between-tiles].

Information Leak Requirements

As shown later on, the security of the inter-tile communication boundary is currently fairly weak. This means that, in theory, an attacker gaining RCE in an upstream tile can propagate this through multiple sandboxes, always exploiting the following one via the shared-memory link.

However, all tiles run in different processes with a different randomized address-space layout. Even using a powerful 'write-what-where' primitive, an exploit needs to know what to write, and where. This usually requires a leak of information, such as a pointer onto the heap, or a function-pointer.

The current design of Firedancer v0.1 follows a linear structure where earlier tiles can send data to later tiles but cannot receive data back. The shared-memory areas are usually writable only by the publisher, which would be the attacker, not the consumer. So there cannot be any leak of data on this link. Hence, despite the relatively weak inter-tile communication security, ASLR significantly mitigates the risk of exploitation.

One notable area of concern was the hugetables explicitly mapped for the stack, and the location of other shared-memory areas. However, Firedancer effectively randomizes their locations, complicating any attempts at information leaks.

There are leakless exploitation techniques, however applicable scenarios for these are generally hard to find and need the correct primitives. They often rely on partial-overwrites of values, and relative-addressing gadgets. There is a lot of prior work with regards to leakless heap-exploitation in glibc, but those techniques also typically rely on brute force to bypass randomization, with incorrect guesses resulting in crashes.

One other way to get leaks are speculative hardware attacks. There have been instanaces of cross-process leaks used to exploit chromium:

- Escaping the Chrome Sandbox with RIDL
- Mitigating RIDL Side-Channel Attack in Microsoft Edge on Windows

This exploit in particular relies on the fact that processes share the same core, which might be hard or impossible to provoke in Firedancer, depending on how exactly the tile threads are pinned to cores. But there might be other such bugs that even work cross-core.

6 | Exploit Investigation

Possible Impacts

To evaluate the potential risks associated with Remote Code Execution (RCE) in each tile of Firedancer v0.1, we will analyze the functional requirements without considering sandbox escapes. This gives insight into what each sandbox protects, and where an attacker would want to get RCE. It also provides some view into what kind of issues accidental bugs in the tiles might have on the system, even without an active attacker being present. We assume that less than 2/3rd of the stake is running Firedancer v0.1, ensuring Agave retains a superminority capable of halting consensus in case of issues.

Each tile is crucial for the proper functioning of Firedancer v0.1. The failure of any single tile, such as through a null-pointer dereference, will shut down the entire validator. The impact of different types of bugs varies, with RCE offering significant flexibility for non-functional impacts.

Below, we define categories for consequences of bugs we expect are most likely:

1. '**REPAIR**': Agave has to repair all blocks. Votes continue to be sent and processed, but block production ceases.
2. '**NOBLOCK**': Block production ceases.
3. '**VOTE_ONLY**': Firedancer v0.1 can only produce vote-only blocks, as votes are ingressed via gossip.
4. '**PACK**': The attacker gains significant control over the content of blocks this node produces when it acts as leader.
5. '**ILLEGAL**': The attacker can cause Firedancer v0.1 to produce illegal, slashable blocks.
6. '**LEAK**': The attacker can leak cryptographic key material, notably the validator identity or vote key.
7. '**AGAVE**': The attacker can take over the Agave runtime, consensus, and pretty much everything else. If more than 1/3rd of the stake runs Firedancer v0.1, this could halt consensus, necessitating a hardfork to resolve. Additionally, the laxer sandbox allows easier interaction with the outer world, or compromise of the server as a whole.

Using these impact categorizations, we consider the impact of bugs causing either a crash, unresponsiveness or RCE individually for each tile. The table below summarizes the consequence of each. Note that RCE always also implies an impact of unresponsiveness and crash, since those are trivial to cause with RCE.

Tile	Crash	Unresponsive	RCE
net	Full DoS	NOBLOCK/REPAIR	PACK
quic	Full DoS	VOTE_ONLY	PACK
verify	Full DoS	VOTE_ONLY	PACK/ILLEGAL
dedup	Full DoS	VOTE_ONLY	PACK/ILLEGAL
pack	Full DoS	NOBLOCK	PACK/ILLEGAL
bank	Full DoS	NOBLOCK	PACK/ILLEGAL/AGAVE/LEAK
poh	Full DoS	NOBLOCK	PACK/ILLEGAL/AGAVE/LEAK
shred	Full DoS	NOBLOCK/REPAIR	ILLEGAL
store	Full DoS	NOBLOCK/REPAIR	PACK/ILLEGAL/AGAVE/LEAK
sign	Full DoS	NOBLOCK	LEAK
metrics	Full DoS	N/A	N/A

Notably, since all of replay still runs in Agave, a simple node unresponsiveness without crash likely only leads to halting of block production. Additionally, there is little option for Firedancer to permanently fork off. Only Firedancer v0.1 nodes with compromised Agave/Replay might participate in a split-brain. An attacker likely can't benefit from producing illegal blocks, since neither Agave nor other Firedancers v0.1 will accept them, and the compromised Firedancer v0.1 instance will quickly switch to a valid fork.

Note that most of the consequences above arise inherently from the functionality of the tiles. For example, you have to trust verify to accurately verify signatures, as that is the explicit task.

Sandbox Secrets

Another interesting attack vector is secret exfiltration. There aren't many secrets in play with Firedancer, most notably the vote and identity key. So the only Firedancer tile which actively holds secrets worth stealing is the sign tile, which is well-protected and has a minimal surface area.

However, the Agave runtime also needs access to the identity and vote keys. In turn, all tiles in the Agave address space can leak the keys. In addition, the Agave tile isn't really sandboxed, making it an interesting target for comprehensive system exploits, such as stealing SSH credentials or traversing the operator's internal infrastructure to access other sensitive systems.

In theory, a competent operator might prevent this simply by the UID/GID sandbox, but more mitigations could be put in place regardless. Implementing light sandboxing techniques such as Linux Landlock, as already used in newer Firedancer sandboxes, could help alleviate some of these issues. While not as robust as the primary Firedancer sandboxes, this approach provides some level of mitigation.

7 Exploit Paths

As seen above, RCE in different tiles/sandboxes has different impacts. Therefore, we must consider the robustness of the interfaces between the tiles to assess if an attacker can traverse from one sandbox to another. For this, we will examine the shared-memory interfaces between the tiles in more detail.

Shared Memory Between Tiles

Firedancer's message-passing architecture relies on shared-memory mappings for each producer/-consumer link. These mappings, called workspaces, are joined before the sandboxing of a process is applied. Multiple data structures are placed in these regions, with the producer having read-write access while consumers typically have read-only access to the links.

Fragments are sent over these links, each containing an 8-byte signature with link-defined meaning, a sequence number, some other metadata, and optionally payload data.

Given Firedancer's inherently multi-process architecture, there is potential for time-of-check to time-of-use (TOCTOU) vulnerabilities if consumers are not careful. An attacker could send a message and replace its content while the consumer is parsing it. Reading an entire message atomically is rarely feasible.

Ideally, each tile would copy all data received from a link into memory exclusively controlled by that tile before parsing or making decisions. However, this approach has performance implications, as each tile would need to constantly copy all data.

To optimize performance, Firedancer does not always blindly memcpy all data before making decisions. It often partially examines data for early rejections. This is acceptable if only rejections are performed, but it increases the risk of introducing TOCTOU bugs.

These issues can occur maliciously or due to unreliable links that might be overrun. If a link's producer generates too many items for the consumer to keep up, the producer might overwrite an item the consumer is handling. Firedancer addresses this by using a stream-multiplexer for each tile, which provides three main packet parsing functions:

- **before_frag**: Called with the publisher ID, sequence number, and signature for each packet for early filtering.
- **during_frag**: Called with payload data. It should copy all data to avoid overrun issues since data might be overwritten at any time. This stage should not have problematic side effects.
- **after_frag**: Called when the fragment was not overrun during `during_frag` and should commit everything.

Additionally, there are periodic housekeeping functions. Full documentation is available in fd_mux.h.

Apart from overrun/malicious TOCTOU bugs, another type of bug can occur: overly trusting the data. While some trust is necessary, such as trusting the verify tile to check signatures correctly, it is unwise to assume the data provided is entirely safe. Lengths, pointers, or offsets transferred over links could chain a bug into the consumer, potentially enabling an RCE-chain: exploiting QUIC, then Verify, and so on.

We investigated how feasible TOCTOU like bugs are, and what data is trusted and checked over each shared memory link. We detail our findings on this in the next sections.

TOCTOU Investigation

We manually inspected the data accessed by each tile in before/during_frag to find time-of-check - time-of-use (TOCTOU) bugs, which could be abused by an attacker racing the check.

Tile	TOCTOU Vulnerable
net	no
quic	no
verify	no
dedup	no
pack	no
bank	no
poh	Microblock trailer fully unchecked. In the microblocks received from bank, only transactions are copied, not the trailer, which includes the bank index. This means the bank can change header after any checks. This can be used with a later `fseq_update` to have an arbitrary write-what-where primitive. In practice, both bank+poh tiles run in the same address space anyways, so not a huge issue.(Finding: PoH trusts the microblock_trailer it got from pack too much)
shred	Always copies full entry, but not `fd_entry_batch_meta_t` for fragments arrinving from PoH. The decision to finalize a batch is made by reading `entry_meta->block_complete`, before copying it. This means you could create a batch that doesn't actually have the `block_complete` flag correct. Not a practical issue, as blocks are automatically completed when the `PENDING_BATCH_WMARK` is reached, so following code has to handle this case correctly anyways.
store	no
sign	no

During our investigation, we focused mainly on the source code and expected compiler behavior. However, TOCTOU-like bugs can also be introduced by the compiler. If variables are not defined as **`volatile`**, the compiler might assume the memory contents do not change and emit a double-fetch. GCC has been known to do this in certain switch-case statements.

While we found no instances of this in Firedancer, our check was not exhaustive. Behavior may change with different compiler versions and compile flags. An example of a large project affected by such a bug is the The XEN Hypervisor. More information on this can be found in a Report by NCCGroup.

To address these potential issues, we recommend a more structured approach: Write the code in a way so it's obvious where you do the copy, and then copy everything at once. Going through all these places and checking behavior from code alone is feasible, but has to be done on each source modification and pull-request again. Additionally, without very cautious measures like marking everything as volatile, it is hard to always predict the compiler's behaviour. This measure would likely have adverse performance impact. Keeping early-abort checks is fine, as long as you re-check after the copy.

While this restructuring might have performance implications, it would be a defense-in-depth approach to enhance security.

Inter-Tile Trust-Based Exploit Chains

In addition to potential TOCTOU vulnerabilities, there are risks associated with tiles overly trusting parsed data from other tiles, especially when this data includes pointers, lengths, or offsets. Each of these elements should be verified to be in-bounds before use.

We examined all links to identify where such data is transferred and trusted and found several instances where this trust is violated.

Example: The pack-bank interface.

- The pack tile provides a raw pointer to the agave-bank to the bank tile.
- This pointer is fully unchecked and used as a 'self' pointer in Rust.
- This unchecked pointer creates a flexible exploit primitive, likely allowing an exploit to chain from the Pack tile into the Agave address space.

Findings:

- PoH trusts the microblock_trailer it got from pack too much
- Bank pointers are unprotected
- Pack trusts verify tile for parsing transactions

Recommendations

There are some defense-in-depth measures Firedancer can take to improve security with inter-tile transfers.

1. **Never Trust Offsets, Lengths, or Pointers Shared Between Tiles:**

- While tiles must trust the sender regarding the data's accuracy, they should not trust it in ways that can compromise memory safety.
- Each tile should validate any pointers, lengths, or offsets received from another tile.

2. **Avoid Simultaneous Transfer of Raw Bytes and Parsed Representations:**

- Ensure that Firedancer does not transfer both raw bytes and their parsed representation simultaneously unless all consumers re-run the parsing logic to verify the data's correctness.
- Currently, this issue exists with transactions where both raw and parsed versions are transferred, which do not necessarily match.

8 Patches to Agave

Firedancer v0.1 integrates Firedancer and Agave code through a custom foreign-function interface (FFI). This interface uses 32 custom functions for bi-directional communication. For example, Agave defines functions that Firedancer v0.1 calls, such as `fd_ext_blockstore_insert_shreds`, used by the shred tile to insert shreds into the Agave blockstore. Conversely, Firedancer defines functions that Agave calls, such as `fd_ext_poh_begin_leader`, invoked by Agave's replay stage to execute transactions in a new bank.

The patch-set is surprisingly elegant, and small enough so that Firedancer v0.1 currently can keep track with Agave development, by applying it on-top of the current Agave master. Each patch well structured and single-purpose.

Interface Robustness

The interface's robustness is crucial due to the different programming languages on both ends (C for Firedancer and Rust for Agave). We found no major issues related to language constraints. However, crash-resistance issues were identified, particularly concerning the use of `unwrap` or `expect` in Rust, which can cause the entire validator to shut down if triggered. There are two places where is this an issue:

One significant issue arises in the communication of stake weights. Firedancer imposes strict limits on queue sizes, with a maximum of 40200 entries to fit into a single fragment on the link. However, it is possible to have more than 40200 staked nodes, which would cause the unwrap on the Agave side to panic and shut down Firedancer v0.1. While Agave might also struggle with such a high number of staked nodes, it would not crash under the same conditions. (Finding: Stake weight sending has easy to reach DoS by creating more than 40200 validators)

Another problem was found with gossip connection information. There is a discrepancy where Agave sends a maximum of 40200 entries, while Firedancer panics if it receives more than 41999 entries. This off-by-one error needs to be addressed to ensure consistent behavior between the two systems. (Finding: Potential panic in `fd_stake_ci_dest_add_fini`)

There are other unwraps, but those are not reasonably triggerable. For example, the shred-insert might fail, but that only happens when there are errors returned form the RocksDB, in which case there is no clean recovery anyways.

Interactions During Leader Slots

We specifically considered interactions in between Agave and Firedancer in the case when the Firedancer v0.1 node acts as leader and produces blocks. Below is a high-level overview of this process.

In the background, Agave continuously replays all blocks. When the node becomes a leader, the Agave replay stage constructs a bank and sends it via the custom `Firedancer_PoH_Recorder` to the Firedancer PoH Tile. This PoH Tile then notifies the Pack Tile that the node is now the leader. Consequently, the Pack Tile begins ingesting transactions from its queues, which are kept up-to-date with new transactions at all times. The Pack Tile creates microblocks and sends them back to the Agave bank through the Firedancer Bank Tiles.

Multiple Bank Tiles can be involved in this process, with the Pack Tile responsible for locking accounts. After execution, the Bank Tiles forward the execution results to the PoH Tile, where a new tick is created. During this time, the PoH Tile ensures that ticks are progressing smoothly.

Once a slot is completed, the Pack Tile notifies the PoH Tile that packing is finished. The PoH Tile then waits until it has received all the microblocks currently in transit before finalizing the slot. As the unlock of accounts only happens in PoH, the insertion order of execution results into PoH is guaranteed to be consistent with the order they were applied to the bank.

AGAVE Replay
Agave MISC
AGAVE Banks
Firedancer Pack
Firedancer Banks
Firedancer PoH
Firedancer Shred

replay transactions in a loop

maybe_start_leader()

becomes leader.
Create new leader bank

Firedancer_PoH_Recorder::set_bank()

leader bank arc ptr*

lock PoH

POH_PKT_TYPE_BECAME_LEADER
(incl: leader-bank-ptr)

PoH::fd_ext_poh_begin_leader

unlock PoH

pack microblock

check if new bank is idle & schedule microblock

microblock + bank_ptr

mark bank as busy

Round robin on ID that pack chose

convert microblock into Agave memory format

get pre_balance_info

load and execute TXs

DONT LOCK ACCOUNTS! (fake it)
create new batch
bank.load_and_execute_transactions()
Gather results

execution result

commit transaction + results to bank arc ptr

commit

minor commit housekeeping

Hash Executed Transactions

mix TXs + hash tick
into PoH

mark bank as idle

microblock / empty-tick

create shred, FEC

register_tick

publish to net

keep room to always accept max number of microblocks

book-keeping: Timed-out leader-slot

POH_PKT_TYPE_DONE_PACKING

updates expected number of uBlocks

keep hashing, until we receive the exact number of microblocks promised

9 Tile Evaluation

In this section, we investigate the more interesting functionality of various tiles within Firedancer v0.1. While all tiles were audited, this section highlights where we found noteworthy observations.

Additionally, we evaluated potential discrepancies in behavior between Agave and Firedancer, particularly in networking and consensus-related aspects. (Also see the previous Section on Recommendation for Agave Compatibility). Our findings for each tile are detailed below.

Sign Tile and Cryptography Evaluation

Sign Tile

There are various parts in Firedancer v0.1 that need access to sensitive key material to sign messages. Firedancer wraps this into a single Sign Tile. All other components that require signing capabilities interact with this tile via a memory mapped queue. In the Solana protocol, there is an identity key. This signing key is employed for a multitude of message formats. These formats include such things as shreds, quic and gossip. A signing domain encompasses at least one message format. Since the Sign Tile receives messages to be signed via a queue, ideally, these domains would be strictly distinguishable. Unfortunately, they are not. However, Firedancer provides a CMBC proof which guarantees that domains can be distinguished by parsing messages within the Sign Tile and checking for certain constraints.

Even though the Sign Tile uses this, right now only two domains are implemented:

- `FD_KEYGUARD_ROLE_LEADER`
- `FD_KEYGUARD_ROLE_TLS`

These domains are used for signing shreds and quic/TLS handshake hashes respectively. Those two are trivial to distinguish, based on message size alone.

Cryptographic Primitives

Three components, namely `keccak256`, `Sha512` and `Ed25519` were investigated in more detail.

Hash Functions

The Keccak256 and Sha512 implementations were checked for completeness and correctness. Especially the round constants, state transistions and absence of XKCP-like buffer overflows were checked.

Ed25519

For Ed25519 in particular, it should under no circumstances be possible to leak the key from the Sign Tile. Assuming an RCE attacker that gains code execution within another tile, timing attacks become a serious threat.

As this is such a relevant factor, the implementation was checked manually and machine guided for constant timeness. The `TIMECOP` tool, a patchset for `Valgrind` was used to identify private information that is used in branching or lookups. No such information was found. TimeCop found a variable-time path in the signature verification, which is unproblematic, as verification only uses publicly known values. The code was also checked for code paths that could result in the compiler including variable-time code, such as divisions.

The behaviour of various special input values, including the identity, invalid/low order points was investigated. Due to checks in the code, but also the robustness of the underlying primitive, no poor behaviour or DOS-vectors were identified. To identify possible signature verification bypasses, the code was checked for early returns, otherwise faulty code paths or incorrect arithmetic. None were identified. As certain checks are omitted, signatures verified in Firedancer are malleable. However, as non-malleability is not required by the employed protocols, this does not pose a threat to security as of this moment. Source code comments also hint to the fact that Firedancer's developers are aware of this property.

Firedancer's ed25519 implementation is tested against the `Wycheproof` set of test vectors, which already checks for many common attacks. This helps to foster confidence into the correctness of the implementation. As the ed25519 sign function takes the pubkey and private key separately, without an inherent check of public key correspondence, the private key could be leaked if it were combined with a faulty public key. However, the Sign Tile has an explicit check that this is not the case when loading a key from disk.

In Firedancer v0.1, the key is not only stored in the Sign Tile, but also in the Agave validator. Agave requires it to, for example, sign gossip messages. As much of the current functionality is still implemented in Agave and the in-address-space running Firedancer tiles, many components have access to the key without communicating with the sign tile. However, the design of the sign tile itself is sound, and once the signing capabilities of Agave are no longer required, the key will be well protected.

Quic Tile

As the Quic Tile is one of the primary externally rechable entry points into the Firedancer v0.1 validator, it was investigated in-depth. QUIC is a general-purpose network protocol for, initially designed by Google. It allows for easy connection multiplexing, and is built on UDP. Its most common application is in HTTP/3.

Solana leverages QUIC for its core transaction ingress, allowing the use of existing DoS mitigation solutions to protect transaction ingress before traffic reaches the validator node. However, QUIC is a complex protocol and offers guarantees that are not essential for Solana transactions. For instance, all QUIC connections must be TLS1.3 encrypted, making connection establishment relatively expensive due to the cryptographic handshake involved. This handshake includes a signature from the identity key, which is used to authenticate staked nodes when communicating within Agave. This mechanism enables Stake-Weighted Quality of Service (SWQOS), ensuring that high-stake nodes are guaranteed connections, while clients without stake may not receive any bandwidth allocation, preventing them from submitting transactions during high-load conditions.

Firedancer's QUIC implementation demonstrates significant care and polish. Our review identified few instances where QUIC did not meet specifications, and no memory corruption or overflow issues were found.

Dynamic testing, particularly of the costly connection initiation, revealed a bug in handling invalid handshakes, causing a validator to respond with an excessive number of acknowledgments in a single connection. (Finding: QUIC tile DoS with INITIAL and CONNECTION CLOSE frames)

There are concerns about slow-loris attacks, where an attacker opens many connections to block others, keeping them alive via periodic pings. Firedancer v0.1 currently lacks protections against such DoS attacks, with no connection limiting based on IP and no SWQOS.

In our test setup, a single QUIC tile handled approximately 270Mbit/s of traffic consisting mostly of costly crypto handshakes, while using 100% of a CPU core.

As examined above, QUIC unresponsiveness likely only lead to production of vote-only blocks. This could be a practical issue if numerous bots attempt to spam transactions from multiple IPs. Although old connections remain open until they timeout, validators are likely to receive forwarded transactions from other validators that connect early.

This mechanism is vulnerable to gaming, where bots aware of Firedancer's behavior could implement strategies to ensure their transaction inclusion, highlighting the need for robust protections against slow-loris attacks.

Pack Tile

During their leader slots, a Solana validator is resposible for building new blocks. The leader incentivized to select those transactions that maximize his block rewards while staying under the block size and cost limits. Firedancer convieniently summarizes these limits in the `fd_pack_limits` struct:

```
struct fd_pack_limits {
  ulong max_cost_per_block;          /* in [0, ULONG_MAX) */
  ulong max_vote_cost_per_block;     /* in [0, max_cost_per_block] */
  ulong max_write_cost_per_acct;     /* in [0, max_cost_per_block] */
  ulong max_data_bytes_per_block;    /* in [0, ULONG_MAX - 183] */
  ulong max_txn_per_microblock;      /* in [0, 16777216] */
  ulong max_microblocks_per_block;   /* in [0, 1e12) */
};
```

- `max_cost_per_block`: limits the cost of the blocks transactions
- `max_vote_cost_per_block`: limits the cost of vote transactions
- `max_write_cost_per_acct`: limits the cost of transactions write to a single account
- `max_data_bytes_per_block`: limit the size of the block
- `max_txn_per_microblock`: limits the size of each microblock
- `max_microblocks_per_block`: limits the number of microblocks per block

It is crucial that these limits are at least as strict as the limits Agave imposes on incoming blocks, as otherwise the block generated by Firedancer will get rejected by the network. The pack tile ingests transactions from the dedup and gossip tile and is connected to the PoH tile to get notifed when the leader slot starts. Finished blocks get published to one or more banking tiles where they will get executed. For more about this flow, check out the section on Interactions When Leader.

Transaction scheduling

The pack tile schedules transactions one microblock at a time. A microblock is a set of transactions with no write conflicts. Each Microblock has a CU limit 1.5M, 75% percent of which are allocated for vote transactions unless the vote cost limit of the entire block has been reached.

Firedancer always tries to schedule the transaction with the highest reward to cost ratio first. The rewards are proportional the fee per signature and the priority fee while costs are calculated based on Agave cost model.

Microblock scheduling is divided into three phases:

Phase 1: Primary transaction scheduling

Schedule transactions until the space allocated for vote transactions is reached.

Phase 2: Vote transaction scheduling

Schedule vote transactions until the space allocated for vote transaction is filled.

Phase 3: Secondary transaction scheduling

Schedule transactions until the microblock is full.

Transaction spam

To reduce latency, transactions are scheduled and executed as soon as possible. This reduces the effectiveness of the prioritisation mechanism as transactions arriving later can't get reordered before transactions that are already scheduled, even if they have a higher priority. Therefore spamming transactions to get them executing as soon as possible is incentivised with the current scheduler.

Invalid and Expired Transactions

In addition, the current design has a significant issue with transactions have a valid signature, but are not executable.

Each transaction includes a block-hash responsible for its timeliness. This block-hash must reference one of the past 150 slots (approximately one minute); after that, transactions expire and become invalid. There are also durable-nonce transactions, that include a 'fake' blockhash stored in an account on the blockchain. When executing, this blockhash is checked against the durable-nonce-account state, and the transaction is rejected if it is invalid.

Another common reason for transaction rejection is an insufficient fee-payer balance. Each transaction on Solana requires a minimum fee of 5000 Lamports. If the fee-payer's balance is insufficient, the transaction is rejected by the runtime.

To summarize, the three likely vectors to reject transactions are:

- Insufficient fee-payer balance.
- Invalid block-hash:
  - For normal transactions: the blockhash is not in the last 150 blocks.
  - For durable-nonce transactions: the blockhash does not match the one in the durable-nonce-account.

Currently, Firedancer v0.1 has no mitigations against these vectors. An attacker can spam transactions with a high reward-to-cost ratio, filling up block space, for free. By specifying victim accounts as writable in these transactions, they can also effectively create a denial-of-service on single accounts.

There is an outlined architecture for fee-payer balance checks, but it always returns true: fee payer balance checks.

Addressing Invalid Transactions

The current situation is understandable, as it is quite tricky to solve. To mitigate transaction spam, it is essential to know past block-hashes, manage fee-payer balances, and handle durable-nonce transactions correctly.

Agave replay could notify on all blockhashes, with Pack storing the past 150 to ensure validity, maybe switching on fork changes. However, handling invalid or empty fee-payer accounts and durable-nonce transactions requires recent bank access, which is challenging due to the separation of Pack and the bank in different processes.

Currently, Pack has no way to communicate with the bank, which runs in a separate process and address space. One solution could be promoting Pack into the Agave address space, though this would weaken the sandboxing. Another solution is to introduce a 'check' tile in the Agave space that verifies transaction validity against the current working bank before they reach Pack. A third solution, theorized by Firedancer, would be to implement a bloom-filter on valid accounts, and pass that on to pack.

Discussions with Firedancer developers revealed they are working on a holistic approach that considers the frequency of invalid transactions from senders and limits their access to block space. One idea could be to reserve block space for 'trusted' senders while penalizing untrusted ones. The exact design of this is still pending, and will have to be evaluated against robustness while it's done. This general approach to make transaction rejection cheap while penalizing those who submit invalid transactions is good.

Comparison to Agave

Agave's scheduler, while also fairly naive, has direct access to the bank, allowing it to avoid unnecessarily locking accounts for extended periods. This is largely done in `check_age`, a function on the bank itself, that even loads nonce-accounts without lock if required.

Unlike Agave, Firedancer's Pack implementation cannot currently include Address-Lookup-Transactions (ALT). This limitation mainly results in the censorship of ALT transactions from all Firedancer v0.1 blocks. However, the replay stage, fully provided by Agave, can correctly handle ALT transactions packed into blocks by other leaders.

Shred Tile

In Solana, blocks are transferred between validators using a custom protocol. Each block is divided into many fragments, called shreds. Multiple shreds are grouped into Forward Error Correction (FEC) sets, which also contain 'coding' shreds. These coding shreds use Reed-Solomon erasure coding to ensure resistance to packet loss without needing retransmission. This is critical as each slot is only 400ms long, and validators must replay the previous slot to build a new one, leaving little time for retransmission.

Each shred is self-contained, with its own header and signature. The Shred Tile has two main responsibilities:

1. **Creating FEC Sets:** The Shred Tile receives transaction batches from the PoH Tile and creates FEC sets, including generating the coding shreds. These shreds are then sent out via the Net Tile.
2. **Reassembly of Incoming Shreds:** The Shred Tile handles the reassembly of incoming shreds from the network. It buffers incoming shreds until enough are received and then uses Reed-Solomon erasure recovery to reconstruct any missing ones. All received shreds are inserted into the Agave blockstore, from which Agave replay picks them up.

There are different types of shreds as the Solana network has evolved its shred protocol over time:

- **Legacy Shreds**: These are no longer accepted by Agave.
- **Merkle Shreds**: Currently in use.
- **Chained-Merkle Shreds**: A new development not yet in use by the network.

Issues Identified

During our investigation, we found several issues with how Firedancer accepts and rejects shreds:

1. **FEC Set Size Limitation:** Firedancer limits the number of shreds in a single FEC set to 67 data shreds and 67 coding shreds. While this matches the maximum size an unmodified Agave client will send, Agave itself can accept larger FEC sets. This limitation means that Firedancer nodes might not accept slots that the rest of the network builds consensus on. However, this issue is mitigated by Agave replay, which can handle any FEC set that Agave can handle, as the replay ingress path remains the original Agave code. (Finding: Firedancer v0.1's limit for number of shreds in a FEC-set is lower than Agave)
2. **Acceptance of Legacy Shreds:** Firedancer's acceptance criteria for legacy shreds are too lax. Since Agave no longer accepts legacy shreds, accepting them could temporarily put Firedancer v0.1 on the wrong side of consensus. (Finding: Firedancer does not handle legacy shreds correctly)

3. **Acceptance of mismatching Shreds:** Firedancer doesn't sufficiently check that the parameters of incoming shreds are correct and match in-between all shreds. This could cause Firedancer to accept shreds that Agave rejects (Finding: Firedancer does not check shred version)

In general, it is crucial for Firedancer to make the same acceptance and rejection decisions as Agave. As the code responsible for this is distributed over different components of the Agave codebase, this isn't trivial to achieve. Implementing a fuzzer could help identify discrepancies and ensure consistent behavior between Firedancer and Agave. This consistency will become even more critical with increasing stake and the planned removal of the Agave repair path.

10 | Datastructures

Firedancer comes with its own library of data structures, most of which are heavily optimized for performance. These optimizations often rely on guarantees that the caller must provide, which is quite different from the typical design of non-high-performance data structures.

We started a systematic evaluation of all datastructures, but quickly found that to be inefficient, largely due to the lack of internal checks. Instead, we focused on concrete users and tiles.

Caller Guarantees

To illustrate how security of the data structures depend on their usage, consider Firedancer's hashmap implementation. In it the null key is used to indicate that no entry is present. This means that the constructor of the hashmap must ensure it does not store entries with an all-zero hash, as they might not be retrievable. Similar expectations apply to other data structures, where the caller is responsible for not inputting 'weird' values, avoiding the deletion of non-existent elements, and so forth. This design places the burden of security on the callers, making it challenging to evaluate or fuzz the data structures independently.

Hashing and Performance

A significant concern with Firedancer's sets and maps is the lack of randomization in the hashing functions. Typically, hashmaps handle collisions by falling back to a linear list within each hash bucket. If an attacker can predict the hash bucket into which a value will be sorted, they can severely impact the performance of the hashset, as linked-list operations degrade from `O(1)` to potentially `O(n)`. (Finding: Hashmaps are vulnerable to HashDoS attacks)

Recommendation

We recommend to make the security-constraints of all data structures very explicit. Add comments what invariants have to be conserved by the caller, specifically to maintain security. That way, developers can quickly double-check that what they are implementing follows the requirements.

11 | Findings

This section outlines all of our findings. They are classified into one of five severity levels – critical, high, medium, low, informational – detailed in Appendix D.

All findings are listed in Table 3 and further described in the following sections.

Table 3: Findings

Identifier	Name	Severity
ND-FD1-MD-01	Hashmaps are vulnerable to HashDoS attacks	Medium
ND-FD1-HI-01	QUIC tile DoS with INITIAL and CONNECTION CLOSE frames	High
ND-FD1-MD-02	QUIC vulnerable to a low bandwith DoS	Medium
ND-FD1-MD-03	Quic implementation vulnerable to slowloris attacks	Medium
ND-FD1-LO-01	Risk of Sign-Tile Exhaustion	Low
ND-FD1-HI-02	Stake weight sending has easy to reach DoS by creating more than 40200 validators	High
ND-FD1-MD-04	Potential panic in `fd_stake_ci_dest_add_fini`	Medium
ND-FD1-MD-05	Firedancer v0.1's limit for number of shreds in a FEC-set is lower than Agave	Medium
ND-FD1-LO-02	Firedancer does not check shred version	Low
ND-FD1-LO-03	Firedancer does not check for consistency between code and data shreds	Low
ND-FD1-MD-06	Firedancer does not handle legacy shreds correctly	Medium
ND-FD1-NI-01	QUIC Nitpicks	Nitpick
ND-FD1-LO-04	PoH trusts the microblock_trailer it got from pack too much	Low
ND-FD1-LO-05	Bank pointers are unprotected	Low
ND-FD1-LO-06	Pack trusts verify tile for parsing transactions	Low
ND-FD1-IN-01	Agave joins `shred_store` workspace as RW, could be RO	Informational

[ND-FD1-MD-01] Hashmaps are vulnerable to HashDoS attacks

Severity	Impact	Affected Component	Status
Medium	Attacker can cause tiles to become unresponsive, especially Dedup	Datastructures	Acknowledged

Firedancer does not use randomized hash functions (or none at all for the dedup stage) in its hashmaps. Attackers can abuse this to degrade the performance of the hashmaps from O(1) to O(n) and potentially cause a denial of service.

Details

The hashmap implementation used in fd_tcache.h is based on linear probing. The tag used by fd_dedup.h is just the first 64 bits of a transactions signature [1] and therefore is fully controlled by an attacker who can send many transactions with colliding tags and create a long sequence of filled slots. Trying to access an element which hashes to the first slot in the sequence will be very slow as it has to iterate through the entire sequence to find an empty slot.

Suggested Fix

Use a randomized hash function.

Notes

Pretty much everything used to be vulnerable to this so you can find a lot of information on similar attacks (with linear probing its much easier than with chaining hashmaps since you don't need exact collisions):

- Paper: Denial of Service via Algorithmic Complexity Attacks
- Talk: Efficient Denial of Service Attacks on Web Application Platforms

There is another small issue in fd_dedup.h if the first 64 bits of the signature are 0. (which is very unlikely since the signature has to be valid) This will cause the transaction to either always be filtered if the first slot is empty (since it will already contain 0) or never be filtered.

Firedancer also uses other randomized datastructure, for example treaps, which may also be susceptible to similar attacks if they are not properly randomized and exposed to an remote attacker.

Another issue with linear probing is that even with randomization high load factors can cause performance to drop very quicky. fd_tcache.h deals with this by removing old entries after a threshold.

Resolution

Firedancer is aware of this issue and plans to mitigate it by per-validator random-seeding of hashmaps. This is a good solution.

[ND-FD1-HI-01] QUIC tile DoS with INITIAL and CONNECTION CLOSE frames

Severity	Impact	Affected Component	Status
High	Validator doesn't accept transactions, potential for amplification attack	QUIC	Resolved

When using a `CONNECTION CLOSE` frame in a `INITIAL` packet, the server answers with ~2500 ACK packets. This can be abused to create a DoS with ~500 client packets/s, or to create an amplification attack when `retry = false`.

Commit: `30fb51e2634d8ca80de34e497169bf8a0f6183a7` (`quic - fixed RETRY`)

Details

QUIC packets can contain multiple FRAMES with data (like CRYPTO, PING, ACK, STREAM, …) encapsulated into these frames. In a normal connection establish behaviour, the `INITIAL` packet consists of the `CRYPTO` frame and a `PADDING` frame.

When modifing the `INITIAL` packet to consist of a `CRYPTO` frame, as well as a `CONNECTION CLOSE` frame, we can observe very high load on the `quic` tile. Around 500 packets/s (~ 5 MBit/s) of these packets are enough to fully exhaust the `quic` tile.

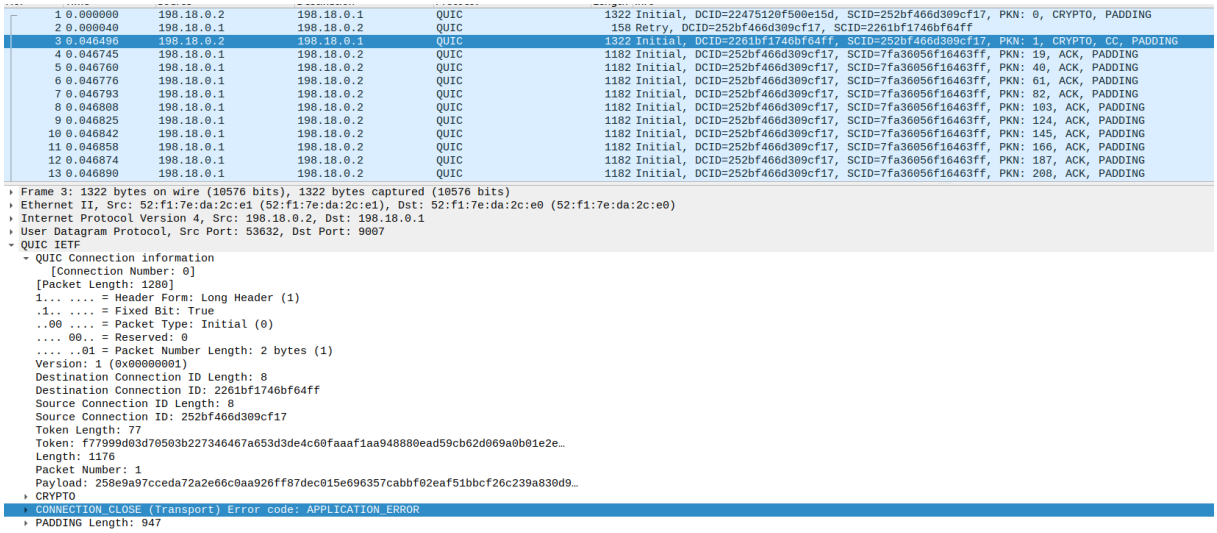


Figure 2: Wireshark

```
snapshot for 2024-05-17 11:26:20.684398592 GMT+02
    tile |     pid |      stale | heart |        sig | in backp |          backp cnt |  % hkeep |  % backp |   % wait |  % ovrnp |  % ovrnr |  % filt1 |  % filt2 | % finish
---------+---------+------------+-------+------------+----------+--------------------+----------+----------+----------+----------+----------+----------+----------+----------
     net |   94685 |          - |     - | run( run ) |    -( -) |        0(     +0) |   32.812 |    0.000 |   60.718 |    0.000 |    0.000 |    0.000 |    0.000 |    0.000
    quic |   94687 |          - |     - | run( run ) |    -( -) |        0(     +0) |    0.008 |    0.000 |    0.000 |    0.000 |    0.000 |    0.000 |    0.000 |   99.992
  verify |   94686 |          - |     - | run( run ) |    -( -) |        0(     +0) |    0.429 |    0.000 |   99.571 |    0.000 |    0.000 |    0.000 |    0.000 |    0.000
  verify |   94688 |          - |     - | run( run ) |    -( -) |        0(     +0) |    0.423 |    0.000 |   99.577 |    0.000 |    0.000 |    0.000 |    0.000 |    0.000
  verify |   94689 |          - |     - | run( run ) |    -( -) |        0(     +0) |    0.424 |    0.000 |   99.576 |    0.000 |    0.000 |    0.000 |    0.000 |    0.000
  verify |   94692 |          - |     - | run( run ) |    -( -) |        0(     +0) |    0.435 |    0.000 |   99.565 |    0.000 |    0.000 |    0.000 |    0.000 |    0.000
  verify |   94693 |          - |     - | run( run ) |    -( -) |        0(     +0) |    0.424 |    0.000 |   99.576 |    0.000 |    0.000 |    0.000 |    0.000 |    0.000
  verify |   94695 |          - |     - | run( run ) |    -( -) |        0(     +0) |    0.436 |    0.000 |   99.564 |    0.000 |    0.000 |    0.000 |    0.000 |    0.000
  verify |   94694 |          - |     - | run( run ) |    -( -) |        0(     +0) |    0.432 |    0.000 |   99.568 |    0.000 |    0.000 |    0.000 |    0.000 |    0.000
  verify |   94699 |          - |     - | run( run ) |    -( -) |        0(     +0) |    0.434 |    0.000 |   99.566 |    0.000 |    0.000 |    0.000 |    0.000 |    0.000
  verify |   94698 |          - |     - | run( run ) |    -( -) |        0(     +0) |    0.434 |    0.000 |   99.566 |    0.000 |    0.000 |    0.000 |    0.000 |    0.000
  verify |   94696 |          - |     - | run( run ) |    -( -) |        0(     +0) |    0.423 |    0.000 |   99.577 |    0.000 |    0.000 |    0.000 |    0.000 |    0.000
   dedup |   94697 |          - |     - | run( run ) |    -( -) |        0(     +0) |    1.074 |    0.000 |   98.926 |    0.000 |    0.000 |    0.000 |    0.000 |    0.000
    pack |   94691 |          - |     - | run( run ) |    -( -) |        0(     +0) |    0.100 |    0.000 |   99.900 |    0.000 |    0.000 |    0.000 |    0.000 |    0.000
    bank |   94683 |          - |     - | run( run ) |    -( -) |        0(     +0) |    0.466 |    0.000 |   99.534 |    0.000 |    0.000 |    0.000 |    0.000 |    0.000
    bank |   94683 |          - |     - | run( run ) |    -( -) |        0(     +0) |    0.491 |    0.000 |   99.509 |    0.000 |    0.000 |    0.000 |    0.000 |    0.000
     poh |   94683 |          - |     - | run( run ) |    -( -) |        0(     +0) |    1.098 |    0.000 |   98.902 |    0.000 |    0.000 |    0.000 |    0.000 |    0.000
   shred |   94701 |          - |     - | run( run ) |    -( -) |        0(     +0) |    0.801 |    0.000 |   99.199 |    0.000 |    0.000 |    0.000 |    0.000 |    0.000
   store |   94683 |          - |     - | run( run ) |    -( -) |        0(     +0) |    0.265 |    0.000 |   99.735 |    0.000 |    0.000 |    0.000 |    0.000 |    0.000
    sign |   94700 |          - |     - | run( run ) |    -( -) |        0(     +0) |    0.474 |    0.000 |   97.714 |    0.000 |    0.000 |    0.000 |    0.000 |    1.812
  metric |   94702 |          - |     - | run( run ) |    -( -) |        0(     +0) |  100.000 |    0.000 |    0.000 |    0.000 |    0.000 |    0.000 |    0.000 |    0.000

            link |  tot TPS |  tot bps | uniq TPS | uniq bps |  ha tr% | uniq bw% | filt tr% | filt bw% |          ovrnp cnt |          ovrnr cnt |          slow cnt |              tx seq
-----------------+----------+----------+----------+----------+---------+----------+----------+----------+---------------------+---------------------+-------------------+--------------------
    quic->net    |    60.1K |   568.4M |    60.1K |   568.4M |  100.000 |  100.000 |  100.000 |  100.000 |        0(     +0) |        0(     +0) |        0(     +0) | 1747634(   +952)
  shred->net     |      0.0 |      0.0 |      0.0 |      0.0 |    0.000 |    0.000 |    0.000 |    0.000 |        0(     +0) |        0(     +0) |        0(     +0) |          0(     +0)
    net->quic    |    505.1 |   5341.7K |    505.1 |   5341.7K |  100.000 |  100.000 |    0.000 |    0.000 |        8(     +8) |        0(     +0) |        0(     +0) |          0(     +0)
    sign->quic   |      0.0 |      0.0 |      0.0 |      0.0 |    0.000 |    0.000 |    0.000 |    0.000 |        0(     +0) |        0(     +0) |        0(     +0) |          0(     +0)
```

Figure 3: Tile Utilization

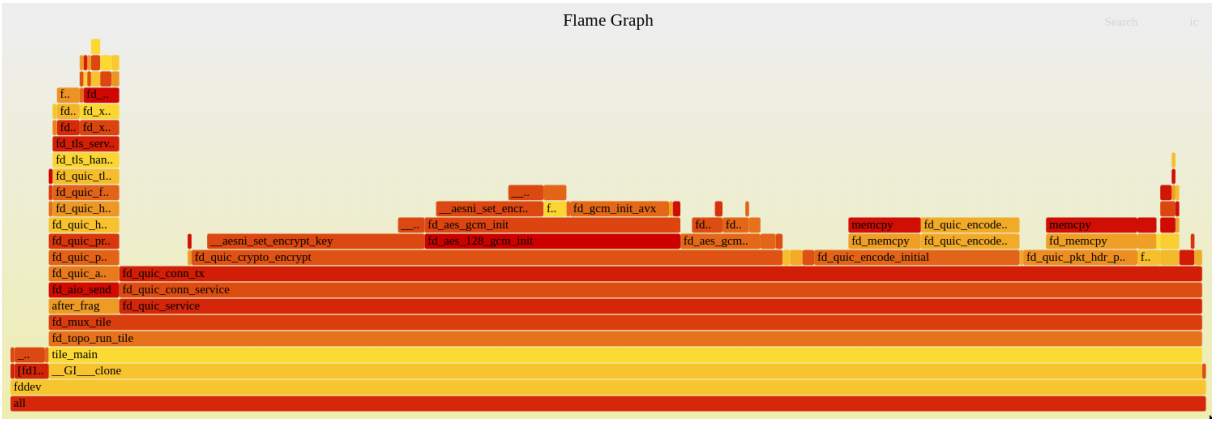


Figure 4: Flamegraph

The reason for the resource exhaustion stems from a **lot** (~2500) of QUIC `ACK` packets that are responsed to the initial sender. 500 `INITIAL` packets/s (~ 5MBit/s) generate around 570 MBit/s of traffic to the sender. In case of `retry = false`, this attack can be used as amplification attack against other IP addresses with an amplification factor of ~50x.

The root cause for this behaviour is currently unknown. Something in the connection-handling state machine in `fd_quic_service` breaks.

Suggested fixes

Investigate why the combination of `INITIAL` and `CONNECTION CLOSE` generates that many ACK packets.

Reproduction

For brevity, the full reproduction scripts aren't included in this report.

Resolution

Firedancer fixes this with PR#2013. Neodyme has confirmed the issues doen't occur on the latest code anymore.

[ND-FD1-MD-02] QUIC vulnerable to a low bandwith DoS

Severity	Impact	Affected Component	Status
Medium	Validator doesn't accept transactions	QUIC	Acknowledged

Firedancer's Quick interface is vulnerable to a low-bandwidth denial of service attack.

Commit: `30fb51e2634d8ca80de34e497169bf8a0f6183a7` (`quic - fixed RETRY`)

Details

To further investigate the bahaviour of the FD validator and its robustness to QUIC spam, we reduced the `max_idle_timeout` to `100ms`. Otherwise we would've run always in the "slowloris"/connection exhaustion finding above.

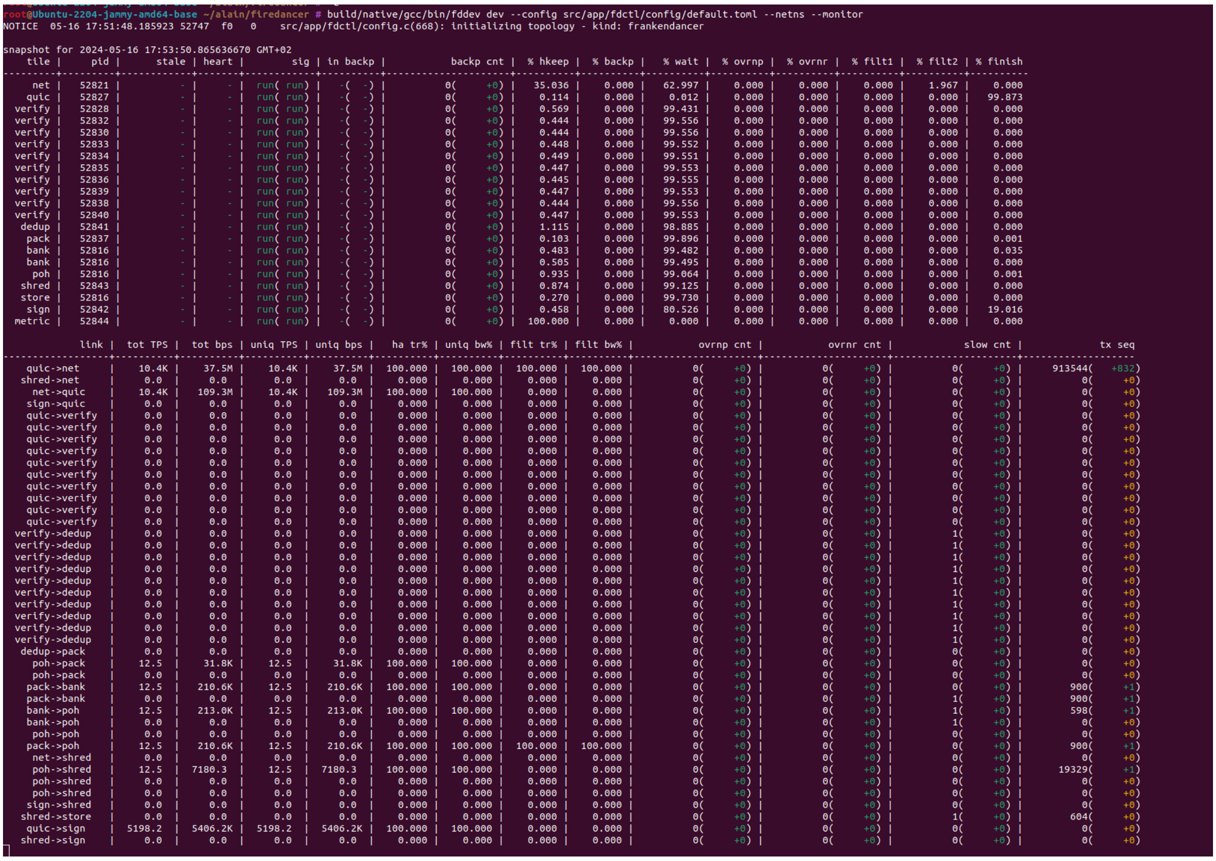
With those settings, we could estimate how resource expensive serveral operations of the FD QUIC stack are.

Case: Retry = true (Amplification protection)

In the `retry = true` case, a full handshake looks like:

```
Client: INITIAL
Server: RETRY (provides Token)
Client: INITIAL (with Token)
Server: TLS Flow => x25519 => expensive
```

The TLS flow is quite expensive computation wise. A "low bandwith" (~10k pakets/s, 100 MBits/s) DoS with the above timeout settings is enough to bring down a single `quic` tile. Note that we did not send any close packets, and relied purely on the timeout to close old connections.



```
root@Ubuntu-2204-jammy-amd64-base ~/alain/firedancer # build/native/gcc/bin/fddev dev --config src/app/fdctl/config/default.toml --netns --monitor
NOTICE  05-16 17:51:48.185923 52747  f0   0    src/app/fdctl/config.c(668): initializing topology - kind: frankendancer

snapshot for 2024-05-16 17:53:50.865636670 GMT+02
    tile |    pid |      stale | heart |        sig | in backp |          backp cnt |  % hkeep |  % backp |   % wait |  % ovrnp |  % ovrnr |  % filt1 |  % filt2 | % finish
---------+--------+------------+-------+------------+----------+--------------------+----------+----------+----------+----------+----------+----------+----------+----------
     net |  52821 |          - |     - | run( run)  |  -(  -)  |          0(     +0) |   35.036 |    0.000 |   62.997 |    0.000 |    0.000 |    0.000 |    1.967 |    0.000
    quic |  52827 |          - |     - | run( run)  |  -(  -)  |          0(     +0) |    0.114 |    0.000 |    0.012 |    0.000 |    0.000 |    0.000 |    0.000 |   99.873
  verify |  52828 |          - |     - | run( run)  |  -(  -)  |          0(     +0) |    0.569 |    0.000 |   99.431 |    0.000 |    0.000 |    0.000 |    0.000 |    0.000
  verify |  52832 |          - |     - | run( run)  |  -(  -)  |          0(     +0) |    0.444 |    0.000 |   99.556 |    0.000 |    0.000 |    0.000 |    0.000 |    0.000
  verify |  52830 |          - |     - | run( run)  |  -(  -)  |          0(     +0) |    0.444 |    0.000 |   99.556 |    0.000 |    0.000 |    0.000 |    0.000 |    0.000
  verify |  52833 |          - |     - | run( run)  |  -(  -)  |          0(     +0) |    0.448 |    0.000 |   99.552 |    0.000 |    0.000 |    0.000 |    0.000 |    0.000
  verify |  52834 |          - |     - | run( run)  |  -(  -)  |          0(     +0) |    0.449 |    0.000 |   99.551 |    0.000 |    0.000 |    0.000 |    0.000 |    0.000
  verify |  52835 |          - |     - | run( run)  |  -(  -)  |          0(     +0) |    0.447 |    0.000 |   99.553 |    0.000 |    0.000 |    0.000 |    0.000 |    0.000
  verify |  52836 |          - |     - | run( run)  |  -(  -)  |          0(     +0) |    0.445 |    0.000 |   99.555 |    0.000 |    0.000 |    0.000 |    0.000 |    0.000
  verify |  52839 |          - |     - | run( run)  |  -(  -)  |          0(     +0) |    0.447 |    0.000 |   99.553 |    0.000 |    0.000 |    0.000 |    0.000 |    0.000
  verify |  52838 |          - |     - | run( run)  |  -(  -)  |          0(     +0) |    0.444 |    0.000 |   99.556 |    0.000 |    0.000 |    0.000 |    0.000 |    0.000
  verify |  52840 |          - |     - | run( run)  |  -(  -)  |          0(     +0) |    0.447 |    0.000 |   99.553 |    0.000 |    0.000 |    0.000 |    0.000 |    0.000
   dedup |  52841 |          - |     - | run( run)  |  -(  -)  |          0(     +0) |    1.115 |    0.000 |   98.885 |    0.000 |    0.000 |    0.000 |    0.000 |    0.000
    pack |  52837 |          - |     - | run( run)  |  -(  -)  |          0(     +0) |    0.103 |    0.000 |   99.896 |    0.000 |    0.000 |    0.000 |    0.000 |    0.001
    bank |  52816 |          - |     - | run( run)  |  -(  -)  |          0(     +0) |    0.483 |    0.000 |   99.482 |    0.000 |    0.000 |    0.000 |    0.000 |    0.035
    bank |  52816 |          - |     - | run( run)  |  -(  -)  |          0(     +0) |    0.505 |    0.000 |   99.495 |    0.000 |    0.000 |    0.000 |    0.000 |    0.000
     poh |  52816 |          - |     - | run( run)  |  -(  -)  |          0(     +0) |    0.935 |    0.000 |   99.064 |    0.000 |    0.000 |    0.000 |    0.000 |    0.001
   shred |  52843 |          - |     - | run( run)  |  -(  -)  |          0(     +0) |    0.874 |    0.000 |   99.125 |    0.000 |    0.000 |    0.000 |    0.000 |    0.000
   store |  52816 |          - |     - | run( run)  |  -(  -)  |          0(     +0) |    0.270 |    0.000 |   99.730 |    0.000 |    0.000 |    0.000 |    0.000 |    0.000
    sign |  52842 |          - |     - | run( run)  |  -(  -)  |          0(     +0) |    0.458 |    0.000 |   80.526 |    0.000 |    0.000 |    0.000 |    0.000 |   19.016
  metric |  52844 |          - |     - | run( run)  |  -(  -)  |          0(     +0) |  100.000 |    0.000 |    0.000 |    0.000 |    0.000 |    0.000 |    0.000 |    0.000

            link |  tot TPS |  tot bps | uniq TPS | uniq bps |   ha tr% | uniq bw% | filt tr% | filt bw% |          ovrnp cnt |          ovrnr cnt |           slow cnt |             tx seq
-----------------+----------+----------+----------+----------+----------+----------+----------+----------+--------------------+--------------------+--------------------+-------------------
    quic->net    |    10.4K |    37.5M |    10.4K |    37.5M |  100.000 |  100.000 |  100.000 |  100.000 |          0(     +0) |          0(     +0) |          0(     +0) |    913544(  +832)
   shred->net    |      0.0 |      0.0 |      0.0 |      0.0 |    0.000 |    0.000 |    0.000 |    0.000 |          0(     +0) |          0(     +0) |          0(     +0) |         0(    +0)
     net->quic   |    10.4K |   109.3M |    10.4K |   109.3M |  100.000 |  100.000 |    0.000 |    0.000 |          0(     +0) |          0(     +0) |          0(     +0) |         0(    +0)
    sign->quic   |      0.0 |      0.0 |      0.0 |      0.0 |    0.000 |    0.000 |    0.000 |    0.000 |          0(     +0) |          0(     +0) |          0(     +0) |         0(    +0)
    quic->verify |      0.0 |      0.0 |      0.0 |      0.0 |    0.000 |    0.000 |    0.000 |    0.000 |          0(     +0) |          0(     +0) |          0(     +0) |         0(    +0)
    quic->verify |      0.0 |      0.0 |      0.0 |      0.0 |    0.000 |    0.000 |    0.000 |    0.000 |          0(     +0) |          0(     +0) |          0(     +0) |         0(    +0)
    quic->verify |      0.0 |      0.0 |      0.0 |      0.0 |    0.000 |    0.000 |    0.000 |    0.000 |          0(     +0) |          0(     +0) |          0(     +0) |         0(    +0)
    quic->verify |      0.0 |      0.0 |      0.0 |      0.0 |    0.000 |    0.000 |    0.000 |    0.000 |          0(     +0) |          0(     +0) |          0(     +0) |         0(    +0)
    quic->verify |      0.0 |      0.0 |      0.0 |      0.0 |    0.000 |    0.000 |    0.000 |    0.000 |          0(     +0) |          0(     +0) |          0(     +0) |         0(    +0)
    quic->verify |      0.0 |      0.0 |      0.0 |      0.0 |    0.000 |    0.000 |    0.000 |    0.000 |          0(     +0) |          0(     +0) |          0(     +0) |         0(    +0)
    quic->verify |      0.0 |      0.0 |      0.0 |      0.0 |    0.000 |    0.000 |    0.000 |    0.000 |          0(     +0) |          0(     +0) |          0(     +0) |         0(    +0)
    quic->verify |      0.0 |      0.0 |      0.0 |      0.0 |    0.000 |    0.000 |    0.000 |    0.000 |          0(     +0) |          0(     +0) |          0(     +0) |         0(    +0)
    quic->verify |      0.0 |      0.0 |      0.0 |      0.0 |    0.000 |    0.000 |    0.000 |    0.000 |          0(     +0) |          0(     +0) |          0(     +0) |         0(    +0)
    quic->verify |      0.0 |      0.0 |      0.0 |      0.0 |    0.000 |    0.000 |    0.000 |    0.000 |          0(     +0) |          0(     +0) |          0(     +0) |         0(    +0)
  verify->dedup  |      0.0 |      0.0 |      0.0 |      0.0 |    0.000 |    0.000 |    0.000 |    0.000 |          0(     +0) |          0(     +0) |          1(     +0) |         0(    +0)
  verify->dedup  |      0.0 |      0.0 |      0.0 |      0.0 |    0.000 |    0.000 |    0.000 |    0.000 |          0(     +0) |          0(     +0) |          1(     +0) |         0(    +0)
  verify->dedup  |      0.0 |      0.0 |      0.0 |      0.0 |    0.000 |    0.000 |    0.000 |    0.000 |          0(     +0) |          0(     +0) |          1(     +0) |         0(    +0)
  verify->dedup  |      0.0 |      0.0 |      0.0 |      0.0 |    0.000 |    0.000 |    0.000 |    0.000 |          0(     +0) |          0(     +0) |          1(     +0) |         0(    +0)
  verify->dedup  |      0.0 |      0.0 |      0.0 |      0.0 |    0.000 |    0.000 |    0.000 |    0.000 |          0(     +0) |          0(     +0) |          1(     +0) |         0(    +0)
  verify->dedup  |      0.0 |      0.0 |      0.0 |      0.0 |    0.000 |    0.000 |    0.000 |    0.000 |          0(     +0) |          0(     +0) |          1(     +0) |         0(    +0)
  verify->dedup  |      0.0 |      0.0 |      0.0 |      0.0 |    0.000 |    0.000 |    0.000 |    0.000 |          0(     +0) |          0(     +0) |          1(     +0) |         0(    +0)
  verify->dedup  |      0.0 |      0.0 |      0.0 |      0.0 |    0.000 |    0.000 |    0.000 |    0.000 |          0(     +0) |          0(     +0) |          1(     +0) |         0(    +0)
  verify->dedup  |      0.0 |      0.0 |      0.0 |      0.0 |    0.000 |    0.000 |    0.000 |    0.000 |          0(     +0) |          0(     +0) |          1(     +0) |         0(    +0)
  verify->dedup  |      0.0 |      0.0 |      0.0 |      0.0 |    0.000 |    0.000 |    0.000 |    0.000 |          0(     +0) |          0(     +0) |          1(     +0) |         0(    +0)
   dedup->pack   |      0.0 |      0.0 |      0.0 |      0.0 |    0.000 |    0.000 |    0.000 |    0.000 |          0(     +0) |          0(     +0) |          0(     +0) |         0(    +0)
     poh->pack   |     12.5 |    31.8K |     12.5 |    31.8K |  100.000 |  100.000 |    0.000 |    0.000 |          0(     +0) |          0(     +0) |          0(     +0) |         0(    +0)
     poh->pack   |      0.0 |      0.0 |      0.0 |      0.0 |    0.000 |    0.000 |    0.000 |    0.000 |          0(     +0) |          0(     +0) |          0(     +0) |         0(    +0)
    pack->bank   |     12.5 |   210.6K |     12.5 |   210.6K |  100.000 |  100.000 |    0.000 |    0.000 |          0(     +0) |          0(     +0) |          0(     +0) |       900(    +1)
    pack->bank   |      0.0 |      0.0 |      0.0 |      0.0 |    0.000 |    0.000 |    0.000 |    0.000 |          0(     +0) |          0(     +0) |          1(     +0) |       900(    +1)
    bank->poh    |     12.5 |   213.0K |     12.5 |   213.0K |  100.000 |  100.000 |    0.000 |    0.000 |          0(     +0) |          0(     +0) |          1(     +0) |       598(    +1)
    bank->poh    |      0.0 |      0.0 |      0.0 |      0.0 |    0.000 |    0.000 |    0.000 |    0.000 |          0(     +0) |          0(     +0) |          1(     +0) |         0(    +0)
     poh->poh    |      0.0 |      0.0 |      0.0 |      0.0 |    0.000 |    0.000 |    0.000 |    0.000 |          0(     +0) |          0(     +0) |          0(     +0) |         0(    +0)
    pack->poh    |     12.5 |   210.6K |     12.5 |   210.6K |  100.000 |  100.000 |  100.000 |  100.000 |          0(     +0) |          0(     +0) |          0(     +0) |       900(    +1)
     net->shred  |      0.0 |      0.0 |      0.0 |      0.0 |    0.000 |    0.000 |    0.000 |    0.000 |          0(     +0) |          0(     +0) |          0(     +0) |         0(    +0)
     poh->shred  |     12.5 |   7180.3 |     12.5 |   7180.3 |  100.000 |  100.000 |    0.000 |    0.000 |          0(     +0) |          0(     +0) |          0(     +0) |     19329(    +1)
     poh->shred  |      0.0 |      0.0 |      0.0 |      0.0 |    0.000 |    0.000 |    0.000 |    0.000 |          0(     +0) |          0(     +0) |          0(     +0) |         0(    +0)
     poh->shred  |      0.0 |      0.0 |      0.0 |      0.0 |    0.000 |    0.000 |    0.000 |    0.000 |          0(     +0) |          0(     +0) |          0(     +0) |         0(    +0)
    sign->shred  |      0.0 |      0.0 |      0.0 |      0.0 |    0.000 |    0.000 |    0.000 |    0.000 |          0(     +0) |          0(     +0) |          0(     +0) |         0(    +0)
   shred->store  |      0.0 |      0.0 |      0.0 |      0.0 |    0.000 |    0.000 |    0.000 |    0.000 |          0(     +0) |          0(     +0) |          1(     +0) |       604(    +0)
    quic->sign   |   5198.2 |  5406.2K |   5198.2 |  5406.2K |  100.000 |  100.000 |    0.000 |    0.000 |          0(     +0) |          0(     +0) |          0(     +0) |         0(    +0)
   shred->sign   |      0.0 |      0.0 |      0.0 |      0.0 |    0.000 |    0.000 |    0.000 |    0.000 |          0(     +0) |          0(     +0) |          0(     +0) |         0(    +0)
```

Figure 5: Tile Utilization

This DoS Also causes the sign tile to take around 20% of load, because it signs the TLS certificate for each connection.

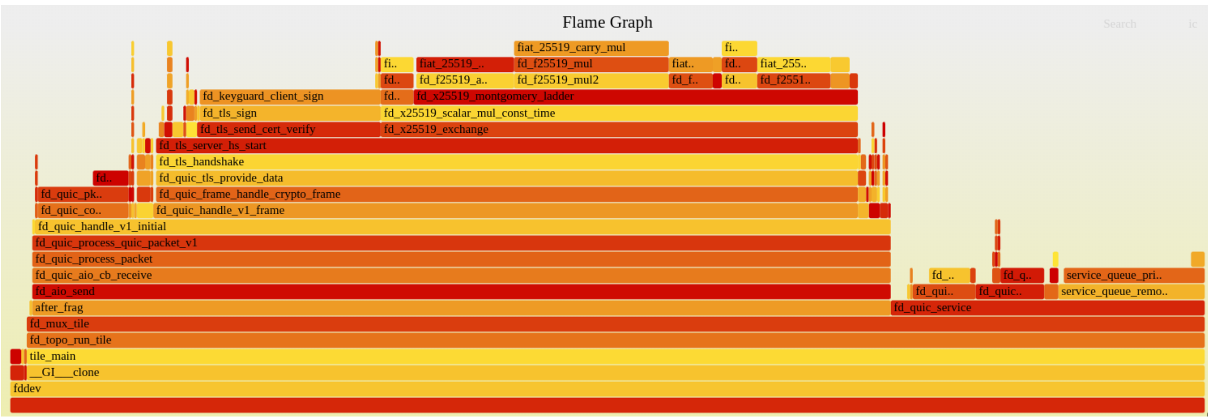


Figure 6: Flamegraph

Case: Retry = false

In the `retry = false` case,the “handshake” is reduced to:

```
Client: INITIAL
Server: TLS Flow => x25519 => expensive
```

Each `INITIAL` packet causes a sign interaction. around 50MBit/s (~5k pakets/s) bring down the quic tile

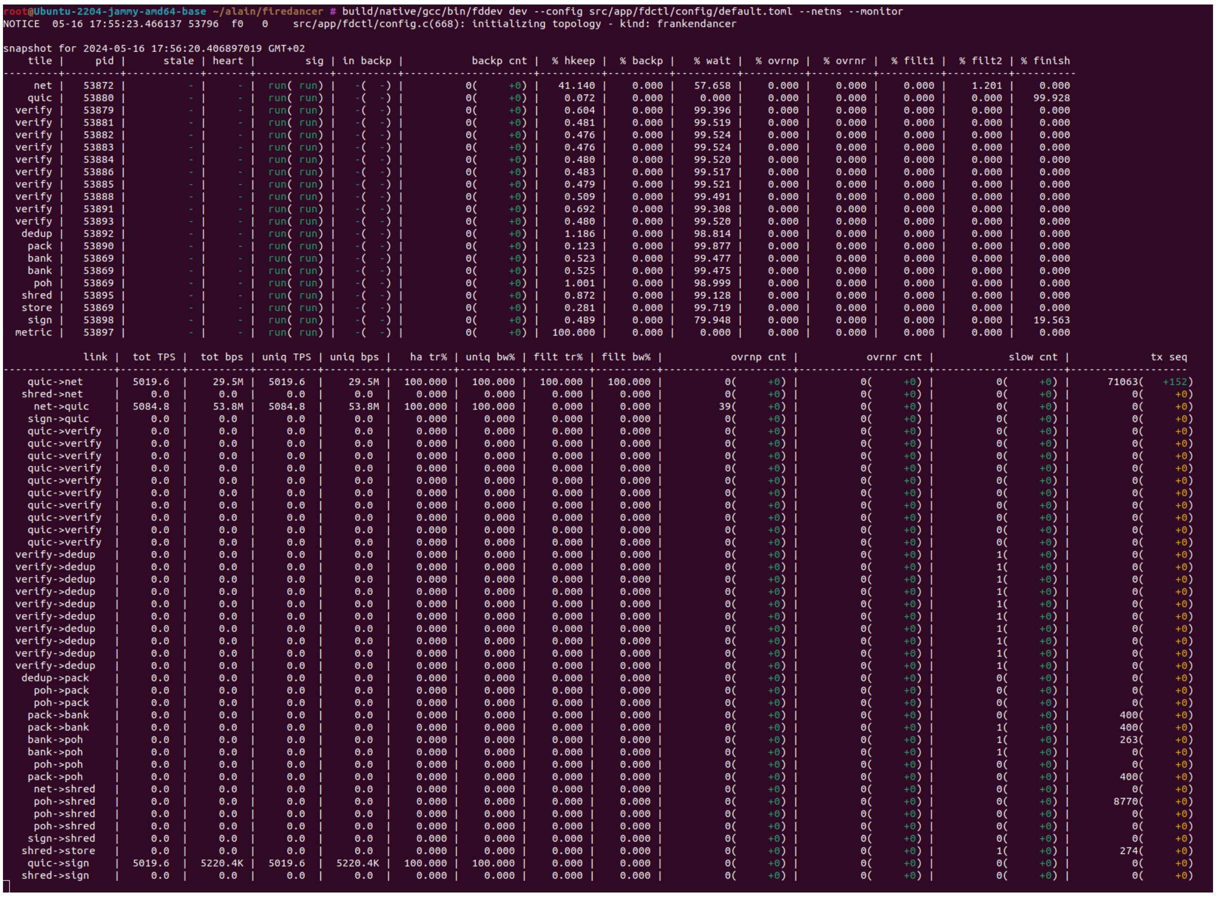


Figure 7: Tile Utilization

This DoS also causes the `sign` tile to take around 20% of load.

Suggested Fix

As an ‘easy’ fix we recommend using the ‘retry’ ip-validation and limiting the maximum number of concurrent connections per IP. In some cases it may be sufficient to allow only one connection per

IP at a time. We also recommend restricting the number of connections that can be established per second per IP.

Remediation

Firedancer is aware of this issue, and is already tracking it as Issue#1376: Ensure QUIC can survive DoS (QoS/credit management)

With the newest changes, the QUIC tile can handle around 270Mbit/s of crafted packets.

[ND-FD1-MD-03] Quic implementation vulnerable to slowloris attacks

Severity	Impact	Affected Component	Status
Medium	Validator doesn't accept transactions	QUIC	Acknowledged

Firedancers quic implementation does not deploy any mitigations to slowloris attacks. As the maximum number of concurrent connections is limited, attackers could claim all available connections and keep them open to prevent other clients from connecting.

Commit: `30fb51e2634d8ca80de34e497169bf8a0f6183a7` (`quic - fixed RETRY`)

Details

By default, firedancer uses the following configuration regarding quic connections:

```
idle_timeout_millis = 10000
max_concurrent_connections = 2048
```

Given these values, an attacker can keep a connection open by sending just one packet every 10 seconds. Therefore, an attacker could keep 2048 connections open by sending less than 300 packets per second.

This, however, does not affect connections that have already been established. It might be possible to cause already established connections to timeout short-volumetric DoS attacks, but we haven't investigated that closer.

Potential Fix

As stated in section 21.6 of RFC9000, QUIC itself does not define any mitigations for Slowloris attacks but proviedes some potential mitigations. However, the provided mitigations might not be applicable in this particular case.

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.

[ND-FD1-LO-01] Risk of Sign-Tile Exhaustion

Severity	Impact	Affected Component	Status
Low	Firedancer v0.1 can't produce blocks	Sign	Invalid

NOTE: This finding was found to be invalid, due to how the link-mux round-robins all connections.

Using a single `sign` tile for all crypto operations increases the risk of a validator DoS, e.g. if the `sign` tile is exhausted by a QUIC DoS.

Commit: `30fb51e2634d8ca80de34e497169bf8a0f6183a7` (`quic - fixed RETRY`)

Details

By design, there is only one sign tile per server. This single sing tile handles QUIC certificate signing, as well as all other keyguard operations.

In the worst case an resource exhaustion attack on the `quic` tile can bring down the `signer` tile and with it all other operations needed for the validator to run.

Depending on the exact commit, and if retry is enabled, we've seen different amount of possible sign-tile utilization, the newest version tested had ~20% sign utilization with a single quic tile doing 100% new connection handling. But the exact percentages could be 'optimized' and depend on the hardware.

We currently estimate that it would be possible to exhaust the sign tile with around 3-4 QUIC tiles, but haven't tested that yet.

Originally suggested fixes

To increase resiliency, we recommend using a dedicated `quic-sign` tile only responsible for the quic certificate signing operations. It is fairly easy to reach externally, and has the possiblity to stall later tiles on their signing operations.

Another aproach might be to introduce a priority queue to the sign tile, so 'validator-triggered' sign operations are always handled first.

Observation

The mux used to ingress signing-requests into the QUIC tile evenly consumes from all producers. No single producer can thus block other tiles by having a long queue. Workload is still shared, but this is essentially an implementation of the separate queue mechanism, so we don’t expect practical issues to occur here, except for slightly increased singning latency.

[ND-FD1-HI-02] Stake weight sending has easy to reach DoS by creating more than 40200 validators

Severity	Impact	Affected Component	Status
High	Firedancer v0.1 Crash	Agave-FFI	Fixed

Due to memory constraints, Firedancer currently limits the maximum number of stake/vote account weights to send to the stake link via `fd_ext_poh_publish_leader_schedule` to 40200 (`FIREDANCER_STAKE_WEIGHT_CNT`):

```
1     if stakes.len() > FIREDANCER_STAKE_WEIGHT_CNT as usize {
2         panic!("unable to communicate leader schedule to Firedancer,
              too many stake weights");
3     }
```

Source

This condition can be triggered fairly easily from any network participant, making all firedancer nodes panic on epoch-boundary.

Details

The sent stake amounts are taken from `bank.epoch_staked_nodes(epoch)`:

```
1     /// staked nodes on epoch boundaries, saved off when a bank.slot()
          is at
2     ///   a leader schedule calculation boundary
3     epoch_stakes: HashMap<Epoch, EpochStakes>,
```

This in turn is usually computed via `update_epoch_stakes`: Source

For the leader-schedule computation, a weighted random draw over *all* staked nodes is taken, there is no minimum delegation amount. The epoch_stakes thus necessarily include *all* staked nodes.

An attacker could thus simply create tons of new vote-accounts, create stake accounts to delegate some funds to each, and wait for an epoch-boundary. The updated stake weights will be sent to firedancer, panicing the validator.

This attack is fairly cheap and quick: Each vote-account costs 0.0270744 SOL in rent, each stake account 0.00228288 SOL plus the staked amount. Solana *mainnet* does NOT have the minimum stake feature activated yet (which would enforce a 1 SOL minimum delgation).

You could thus stake the minimum amount of 1 lamport, and have a cost of less than 0.03 SOL/staked node, or an equivalent of <5$.

Inactive vote accounts (vote accounts that haven't voted since at least a full epoch), can be fully closed, refunding all rent. Same for stake accounts. (eg vote_state/mod.rs).

This means fully exhausting 40200 staked nodes *temporarily* locks around 200k$ in rent. You can create at least one such pair per transaction, likely many more. Landing 40k (non-conflicting!) TXs can be done in well under an hour, depending on network congestion.

Agave does not impose any limit on the number of staked nodes. It likely won't be too happy with so many staked nodes either, especially around epoch-boundaries, but should be able to handle it. (If not, that'd be an easy DoS vector for agave as well).

Since the leader-schedule is computed from a weighted sample of all nodes, you can't simply ignore low-staked nodes in your transfer to firedancer, but have to build another solution that allows FD to handle arbitrarily many staked notes, at least for the leader-schedule computation. Or you do that on the Agave side, and send the completed leader-schedule together with nodes with significant stake-weight for turbine.

Cluster Connection Infos

A similar theoretical problem exists in the transferral of cluster connection information from agave to FD, but there it is NOT a practical issue. Cluster connection info is transmitted every 5s from agave to firedancer.

This happens via `fd_ext_poh_publish_cluster_info`, via `firedancer_send_cluster_nodes`. That also hard-caps the number of items to 40200, printing a warning when it goes over, but not panicing. Instead, it takes the 'first' 40200 nodes: `peers.iter().enumerate().take(len)`:

```
        let len = usize::min(Self::FIREDANCER_CLUSTER_NODE_CNT as usize
            , peers.len());
        //...
        for (i, node) in peers.iter().enumerate().take(len) {
```

cluster_info.rs

Ultimately, this iterates over all cluster info's known to gossip: crds.rs.

This in turn uses the `nodes` index set, which perserves insertion order. So even if very many new cluster info packets were to be spammed, those new ones would be dropped in the firedancer transfer. There are some intricacies on removing nodes from the list, which uses `swap_remove`, but this isn't an attack vector either, because Agave does periodic cleanup to limit the number of possible infos to around 8192 (`CRDS_UNIQUE_PUBKEY_CAPACITY`). Infos are pruned based on stake-weight, so spammy infos

will get removed first. The limit is burstable to higher than 8192, but we find it highly unlikely that this could cause issues. cluster_info.rs

Fix

Firedancer impemented a fix for this issue in PR#2307. This works around the issue, by sending only the most-staked 40k nodes, and how many stake the remaining nodes have. By lucky happenstance, the hashing involved in computing the leader-schedule and trubine-fanout is largely compatible this this, and a leader schedule can still be computed even with missing information. Some turbine fanouts might get missed, but this isn't a big issue, as it is not expected for the network to grow this large anytime soon, and this isn't a valid attack vector for malicious actors anymore.

[ND-FD1-MD-04] Potential panic in `fd_stake_ci_dest_add_fini`

Severity	Impact	Affected Component	Status
Medium	Firedancer v0.1 Crash	Shred	Fixed

When parsing gossip contact information, Firedancer always adds itself first.

1. Agave gathers up to 40200 contact infos from gossip, and sends them to FD
2. FD checks if there are less than 40200 entries, and then adds itself
3. FD updates epoch stake destinations

The concrete issue is a panic if Agave sends exactly 40200 infos, leading to a panic in this check:

```
1  fd_stake_ci_dest_add_fini( fd_stake_ci_t * info,
2                             ulong           cnt ) {
3    /* The Rust side uses tvu_peers which excludes the local validator.
4       Add the local validator back. */
5    FD_TEST( cnt<MAX_SHRED_DESTS );
6      ...
```

Source

To fix this, let Agave send a maximum of `40200-1` infos in `firedancer_send_cluster_nodes`

Actually exploiting this in practice is likely very annoying, since Agave periodically prunes the contact map to ~8k connections.

Fix

This is fixed in PR#2307, by simple sending at maximum one node less from Agave.

[ND-FD1-MD-05] Firedancer v0.1's limit for number of shreds in a FEC-set is lower than Agave

Severity	Impact	Affected Component	Status
Medium	Firedancer v0.1 has to use repair for some blocks	Shred	Acknowledged

The shreds sent over the network are forward-error-corrected. Agave usually produces 32:32 data:coding shred sets to send to other validators. This set might be larger in certain conditions, especially at the end of slots. Firedancer correctly handles this, and can accept FEC-sets of up to 67:67. Agave should not usually produce sets larger than this.

However, Agave *accepts* sets larger than this! As far as we can tell, the only limitation here is in the sanitize functions of the respective shred parsers. There is a check for a max of 256 coding shreds in shred_code.rs::sanitize() Where as data-shreds are limited to 32768 `MAX_DATA_SHREDS_PER_SLOT`

In addition, during erasure-decoding, the reed-solomon-erasure coder Agave uses has a check for

```
        if data_shards + parity_shards > F::ORDER {
            return Err(Error::TooManyShards);
        }
```

Where F is `GF2**8`, so there is an implicit limit that `#data + #coding` must be at most 256. However, this is only used when actually erasure recovering shred data. If all shreds are received (or if a only-data-shred FEC is sent), this is never used. One block could consist of a single FEC-set, only limited by the number of shreds a block can have (32768:0).

Firedancer will simply reject all shreds larger than 67:67 in `fd_fec_resolver_add_shred`.

In practice, while replay and repair are running in Agave, this isn't a big issue. If the network builds consensus on this block, enough nodes will have the shred to repair from. Agave repair is a totally different ingress that uses only Agave code, so there is no limit to FEC set sizes there. In the past, there were validators running exclusively on repair, so Firedancer v0.1 won't fork, just be a bit behind while recovery catches up.

This will have to be fixed once Firedancer no longer uses Agave for repair.

We have not verified the exact limits with Agave in practice, there might be checks in unexpected places.

[ND-FD1-LO-02] Firedancer does not check shred version

Severity	Impact	Affected Component	Status
Low	Not investigated	Shred	Fixed

Firedancer gets shred version from agave (`fd_shred_version`), but never actually checks that in the shred-ingress. Blockstore insert does NOT catch that. It just checks shreds are consistent between each other in a FEC set. Agave has `should_discard_shred` in the *fetch* stage.

We haven't investigated impact here, but the check should be simple enough to add.

Fix

Firedancer fixes this in PR#2274.

[ND-FD1-LO-03] Firedancer does not check for consistency between code and data shreds

Severity	Impact	Affected Component	Status
Low	Not investigated	Shred	Invalid

NOTE: This finding was found to not be accurate. Still including it, since the discussion might be interesting.

Firedancer only checks for consistency between each data and code shred separately, not between them. Shreds might, for example, have a different shred version or slot. Source.

We haven't investigated whether this is actually reachable or filtered out by some previous checks, but we recommend adding a check in any case to ensure there are no edge cases.

Agave re-runs the check on blockstore inserts, but this bug might be usable to cause slots that have to rely on repair. The fix is straight-forward: Check that all fields shared across the first data and coding shred are identical.

Discussion

This finding is invalid. Here is an annotated outline of the fields of each shred, and how they are checked:

```
1  struct __attribute__((packed)) fd_shred {
2  fd_ed25519_sig_t signature;
3
4    uchar  variant;     // taken from base_data_shred for both coding+
       data
5    ulong  slot;        // taken from base_data_shred for both coding+
       data
6    uint   idx;         // "Index of this shred within the slot". used
       for computing `in_type_idx`.
7    ushort version;     // taken from base_data_shred for both coding+
       data
8    uint   fec_set_idx; // taken from base_data_shred for both coding+
       data
9
10   union {
11     struct __attribute__((packed)) {
12       ushort parent_off; // taken from base_data_shred
```

```
13        uchar flags;       // can be arbitrary. 0b10xx_xxxx is invalid
              value, but that is later caught by agave sanitize() on
              blockstore insert. reference tick is only for some time-
              estimation in agave, not critical.
14        ushort size;       // implicitly checked during parsing in
              fd_shred_parse
15      } data;
16
17      struct __attribute__((packed)) {
18        ushort data_cnt;  // taken from base_parity_shred
19        ushort code_cnt;  // taken from base_parity_shred
20        ushort idx;       // must be equal to parity-shred-index
21      } code;
22    };
23  };
```

One thing of note, is that Firedancer is relying on some checks in the Agave sanitize functions to ensure 'shred-acceptance-parity'

This checks that no shred can be last-in-slot, while NOT also being a data-complete shred (`0b10xx_xxxx`), as well as limits on the maximum number of shreds. Firedancer doesn't currently run those checks explicitly, though they still happen on Agave blockstore-insert, and has to be kept in mind when replacing Blockstore with its own tile.

[ND-FD1-MD-06] Firedancer does not handle legacy shreds correctly

Severity	Impact	Affected Component	Status
Medium	Not investigated	Shred	Fixed

As far as we can tell, there are no checks on what shreds are allowed. Firedancer parses all of them, and they have to be one of the 4 Legacy/Merkle Data/Code types. But later, Firedancer simply assumes to only be handling Merkle shreds: Source:

```
1   int is_data_shred = shred_type==FD_SHRED_TYPE_MERKLE_DATA;
2
3   if( !is_data_shred ) { /* Roughly 50/50 branch */
```

This only checks if a shred is NOT a merkle data shred, and then uses the 'code' field only present on code shreds. But Legacy Data shreds are also passed-through to the fec-resolver and then treated as Merkle-Code shreds.

Also, there is no support for the newly introduced chained merkle shreds, which are behind an inactive feature right now, though, on both test- and mainnet: PR#34916. So it might be prudent to whitelist merkle-shreds explicitly.

Agave has also gone and done the same: PR#34328 The corresponding feature gate was activated in Epoch 600 on Apr 9, 2024.

We haven't investigated impact here in detail, but the fix should be easy enough. It seems likely that an attacker can at least cause Firedancer to accept shreds that Agave would not.

Fix

Firedancer fixes this in PR#2274

[ND-FD1-NI-01] QUIC Nitpicks

Severity	Impact	Affected Component	Status
Nitpick	N/A	QUIC	Acknowledged

A couple of quic findings batched, all nit/info severity.

QUIC Key-Logging discards 32 bytes of client/server secret

QUIC secrets are defined as

```
struct fd_quic_tls_secret {
  uint  suite_id;
  uint  enc_level;
  uchar read_secret [ 64 ];
  uchar write_secret[ 64 ];
  uchar secret_len;
};
```

The logging in `waltz/quic/fd_quic.c:2902` only prints the first 32 bytes (hex encoded 64 bytes)

Internally quic also only uses 32 byte secret length everywhere, so only the size is wrong, and should be read from the struct as well (`secret->secret_len`):

```
/* 96+ 1 chars */ s = fd_cstr_append_char( s, ' ' );
/* 97+64 chars */ s = fd_hex_encode( s, client_secret, 32UL );
/*   161 chars */     fd_cstr_fini( s );
```

The same pattern applies a number of other places that copy the secret around.

Wrong (to small) len size in FD_QUIC_MBR_TYPE_{TOKEN,PREFERRED_ADDRESS}

`FD_QUIC_MBR_TYPE_{TOKEN,PREFERRED_ADDRESS}` are defined as

```
#define FD_QUIC_MBR_TYPE_TOKEN(NAME,TYPE)             \
  uint   NAME##_len;                                  \
  uchar  NAME[1024];                                  \
  uchar  NAME##_present;
#define FD_QUIC_MBR_TYPE_PREFERRED_ADDRESS(NAME,TYPE) \
  uint   NAME##_len;                                  \
  uchar  NAME[1024];                                  \
  uchar  NAME##_present;
```

During parsing with Macros, they are truncated to a `uchar` sz, which is limited to 255. While this doesn't have immediate impact, we recommend to use a larger field here, or explicitly reduce the maximum name size.

```
#define FD_QUIC_PARSE_TP_TOKEN(NAME)                          \
  do {                                                        \
    if( FD_UNLIKELY( sz>sizeof(params->NAME) ) ) return -1; \
    fd_memcpy( params->NAME, buf, sz );                       \
    params->NAME##_len = (uchar)sz;                           \
    params->NAME##_present = 1;                               \
  } while(0)

#define FD_QUIC_PARSE_TP_PREFERRED_ADDRESS(NAME)              \
  do {                                                        \
    if( FD_UNLIKELY( sz>sizeof(params->NAME) ) ) return -1; \
    fd_memcpy( params->NAME, buf, sz );                       \
    params->NAME##_len = (uchar)sz;                           \
    params->NAME##_present = 1;                               \
  } while(0)
```

FD_QUIC_PARSE_FAIL should be FD_QUIC_ENCODE_FAIL

Currently, when encoding max data frames, the return value is checked:

```
 /* attempt to write into buffer */
            frame_sz = fd_quic_encode_max_data_frame( payload_ptr,
                                                      (ulong)(
                                                         payload_end -
                                                         payload_ptr ),
                                                      &frame.max_data )
                                                         ;
            if( FD_LIKELY( frame_sz != FD_QUIC_PARSE_FAIL ) ) {
```

The code should check for `FD_QUIC_ENCODE_FAIL` instead of `FD_QUIC_PARSE_FAIL`. Both defines are set to be `~0`, but this might change in the future:

```
#define FD_QUIC_PARSE_FAIL (~(ulong)0)
#define FD_QUIC_ENCODE_FAIL ( ~(ulong)0)
```

This pattern also occures multiple times in the remaining function

Inconsistency in NEW_TOKEN Frame

A lot of parsing code relies on the fact that varint's in quic have a maximum value of `2^62 - 1`, making overflow-checks unnecessary for ulong's.

Every `Frame` definition uses `ulong` field sizes to parse a `quic varint`, e.g. as seen in the `CRYPTO` frame:

```
FD_TEMPL_DEF_STRUCT_BEGIN(crypto_frame)
  FD_TEMPL_MBR_FRAME_TYPE( type, 0x06,0x06 )
  FD_TEMPL_MBR_ELEM_VARINT ( offset,      ulong           )
  FD_TEMPL_MBR_ELEM_VARINT ( length,      ulong           )
  FD_TEMPL_MBR_ELEM_VAR_RAW( crypto_data, 0,12000, length )
FD_TEMPL_DEF_STRUCT_END(crypto_frame)
```

`varint` guarantee that the result is `< 2^62 - 1`. Thus, integer overflow checks can be omitted when the FD quic components uses the parsed data structures of `ulong` (!) values. (`0x3fffffffffffffff`)

In case of the `NEW_TOKEN` frame, and only in this frame, a `uint` (4 byte) is used instead:

```
FD_TEMPL_DEF_STRUCT_BEGIN(new_token_frame)
  FD_TEMPL_MBR_FRAME_TYPE  ( type, 0x07,0x07 )
  FD_TEMPL_MBR_ELEM_VARINT ( token_len, uint             )
  FD_TEMPL_MBR_ELEM_VAR_RAW( token,     0,8192, token_len )
FD_TEMPL_DEF_STRUCT_END(new_token_frame)
```

In the parsing code above, a `varint` value with the max size of `2^62 - 1` is cast to `token_len = (uint)varint`, allowing to have a max value of `0xffffffff`. Parsing with such a high value currently fails in the next step when `0xffffffff > MAX_BITS` for the actual `token` field (= 8192 BITS).

Still, e.g. if the parsing code is changed someday, large values of `uint` might cause overflow problems, as they are assumed by the devs to never overflow.

We thus recommend to use `ulong` consistently, as some assumptions of the current codebase relay on the fact, that sizes don't have the uppermost two bits set (= no overflow when adding to sizes etc.)

[ND-FD1-LO-04] PoH trusts the microblock_trailer it got from pack too much

Severity	Impact	Affected Component	Status
Low	Enables exploit from one Sandbox to another	PoH	Fixed

In the pack_poh link, microblocks are sent together with a `fd_microblock_trailer_t`:

```
struct fd_microblock_trailer {
  /* The hash of the transactions in the microblock, ready to be
     mixed into PoH. */
  uchar hash[ 32UL ];

  /* Bank index to return the bank busy seq on to indicate that
     we are done processing these accounts. */
  ulong bank_idx;

  /* Sequence number to return on the bank_busy fseq to indicate
     that the accounts have been fully processed and can be
     released to pack for reuse. */
  ulong bank_busy_seq;
};
typedef struct fd_microblock_trailer fd_microblock_trailer_t;
```

This has 1.5 bugs. Firstly, the microblock trailer is not copied to PoH space, but simply referenced by pointer. That would allow pack to race any checks that PoH does.

Second and more important: There are no checks on the `bank_idx`. It is used to index into the bank_busy array, but can overflow to point pretty much anywhere.

This gives an attacker a powerful write-what-where primitive to escape from pack into the PoH/Agave sandbox via `fd_fseq_update`, which essentially is just a memory write, to which you provide a pointer and a value, both are taken from the microblock trailer.

```
   fd_fseq_update( ctx->bank_busy[ ctx->_microblock_trailer->bank_idx ],
       ctx->_microblock_trailer->bank_busy_seq );
```

Source: fd_poh.c

Since `bank_busy` is hard-limited to 64 entries, Firedancer should check incoming trailers against this.

Practical exploitation of this issue is likely quite hard, and wasn't further investigated.

Fix

Firedancer fixes this in PR#2184. The microblock trailer is now copied, and it is verified that the bank index has a sensible value.

[ND-FD1-LO-05] Bank pointers are unprotected

Severity	Impact	Affected Component	Status
Low	Potential for exploit-chaining across sandboxes	Agave	Acknowledged

Firedancer v0.1 currently passes around raw bank pointers to tiles running outside of the PoH/Labs address space.

Specifically, when becoming leader, Agave replay constructs a new bank, and passes the pointer to PoH, which in turn forwards it to Pack. Pack then forwards this pointer together with a packed microblock to the bank tile, which then calls into Agave, where the Bank pointer is used.

Pack is a fully sandboxed standalone process and, ideally, shouldn't be able to tamper with a pointer from within the Agave process. Since Pack can provide both the bank pointer and a set of 'arbitrary' microblock contents, it seems reasonable to assume that an exploit chain from Pack to Agave is possible but far from trivial.

We have yet to investigate how hard this is exactly; it likely needs an info leak.

A possible fix for this would be to only pass around a "bank-index", and have a set of possible banks in shared-memory within the Agave address space. That way, replay can let PoH know that a new bank is available in slot X, and this slot number can be propagated through all chains, ultimately getting verified if it's still valid before use.

[ND-FD1-LO-06] Pack trusts verify tile for parsing transactions

Severity	Impact	Affected Component	Status
Low	Potential for exploit-chaining across sandboxes	Pack	Acknowledged

The Pack tile trusts sizes and counts from the transactions parsed in the verify tile. This makes it possible, if quite hard, to chain exploits from the verify tile to the pack tile.

Some discussion on this already exists over at PR#1393 Move txn parsing into verify from QUIC.

We recommend to make this as robust as possible, and prevent this chain. The verify tile is only responsible for verifying incoming transactions. A compromise here, even a compromise for all Firedancer validators, would only lead to illegal blocks being produced, which other validators will not accept, as the path to verify transactions in replay is different and does not go through the same verify tile. Compromising just the Pack tile isn't all that much more interesting, but it is a lot closer to replay.

Conveniently, Pack has control over the bank pointer used for replay. This is another interesting exploit primitive that makes it, at the very least, feasible to break out of Pack into an Agave Bank. This, in turn, is catastrophic, as all of Agave+PoH shares the same address space. (see Issue: Bank pointers are unprotected)

Compromising replay on all nodes in the Network gives an attacker the keys to the castle, providing the ability to do pretty much any otherwise illegal state changes, like minting SOL out of thin air. Different tiles have different exploit severity levels, so we recommend making it hard to chain an exploit between the different sandboxes.

A possible fix would be to move parsing from verify down to pack, where it ultimately has to happen. Note, however, that a lack of parsing in verify could create issues with crafted signatures influencing the dedup map.

[ND-FD1-IN-01] Agave joins `shred_store` workspace as RW, could be RO

Severity	Impact	Affected Component	Status
Info	Weaker Sandbox	Agave Sandbox	Fixed

In `clone_labs_memory_space_tiles`, Firedancer sets up which workspaces the PoH/Agave memory space as access to.

The `shred_store` workspace is joined with write permissions, which isn't required as the store tile in the labs-address-space only consumes from the link, and the shred tile which publishes is it's own sandboxed process.

Something similar might apply to `dedup_pack`, though that's technically required, as both the `dedup_pack` and `gossip_pack` link operate on that workspace, and Agave has to publish to `gossip_pack`. We are not sure what the performance/memory impact would be, but it might be feasible to split up those two links into separate workspaces to prevent the Agave workspace from meddling with the `dedup_pack` link. In practice, Agave is a lot more sensitive than dedup, so a split here doesn't provide much additional security.

Remediation

Firedancer fixed this in PR#2185 agave: reduce workspace permissions.

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.

Neodyme AG

Dirnismaning 55
Halle 13
85748 Garching
E-Mail: contact@neodyme.io

https://neodyme.io