Firedancer v0.1 Audit

Security Assessment Report

July 8, 2024 (version 1.0)

Project Team:

Technical Testing	Bryan C. Geraghty, Jordan Whitehead, Loren Browman
Technical Editing	Brandon Perry
Project Management	Holly Leitru

Atredis Partners ***www.atredis.com***

Table of Contents

Engagement Overview

Assessment Components and Objectives

The Firedancer team recently engaged Atredis Partners ("Atredis") to perform an assessment of v0.1 of the Firedancer Solana Validator. Objectives included validation that Firedancer was developed with security best practices in mind, and to obtain third party validation that any significant vulnerabilities present in Firedancer were identified for remediation.

Testing was performed from April 15 through May 31, 2024, by Bryan C. Geraghty, Jordan Whitehead, and Loren Browman of the Atredis Partners team with Holly Leitru providing project management and delivery oversight. For Atredis Partners' assessment methodology, please see Appendix I of this document and for team biographies, please see Appendix II. Specific testing components and testing tasks are included below.

COMPONENT	ENGAGEMENT TASKS
Firedancer v0.1 Audit	
Assessment Targets	• Firedancer v0.1 • Reimplements various algorithms (Ballet) • Ed25519 • AES • Base58 • Blake3 • Bmtree • Chacha20 • Hex • Hmac • Http • Json • Reed Solomon encoding • Sha256 • Sha512 • Shred • Txn • Wsample • X509 • Components sandboxed in "Tiles" • A tile is a unique process • Each tile is sandboxed, and communicates over one-directional "links", custom IPC mechanisms • The sandboxing and communication mechanisms are in scope • In scope tiles: • Net Tile • Reads and writes packets from/to AF_XDP sockets • Forwards the data to their appropriate handlers

COMPONENT	ENGAGEMENT TASKS
	• Can send ARP probes to retrieve mac address for next hop
	• Netmux Tile
	• QUIC Tile
	• Implements QUIC, an encrypted, multiplexing transport layer network protocol
	• Implements TPU protocol on QUIC for transactions to block producers
	• Verify Tile
	• Dedup Tile
	• Pack Tile
	• Shred Tile
	• Sign (Keyguard)
	• Only component with access to private key material
	• Must never leak key material
	• Metric Tile
	• Bank
	• Executes transactions and updates accounting state
	• PoH
	• “Proof of History” tile
	• Store
	• IPC messaging and Networking layer (Tango)
	• High performance IPC, cache, and control primitives
	• Used for IPC links between tiles
	• Includes shared memory-based communication primitives
	• Includes eBPF programs for XDP redirection
	• Included implementation of TLS subset
	• Flamenco
	• Most of the Flamenco module is out of scope
	• Flamenco/leaders is in scope
	• Provides APIs for the Solana leader schedule
	• Rust FFI
	• Application Module
	• Util
	• The “Funk” module is out of scope
	• Firedancer Solana Fork
	• Repository at a given tagged commit
	• Contains modifications to Solana to support Firedancer
Assessment Tasks	• Source-Assisted Penetration Testing of the Firedancer Client
	• Assessment of Firedancer’s processing of untrusted data from the network
	• Assess XDP layer communication and routing
	• Assess QUIC implementation

Component	Engagement Tasks
	• • Assess sending of ARP and priming of kernel's ARP table • Assessment of Firedancer's Re-implementation of various algorithms • Dynamic and Static testing of in scope Firedancer components • Audit of changes made in the Firedancer Solana Fork • Review of changes made from the original Solana implementation • Evaluation of Firedancer Fuzzing Harnesses and Tests • Code coverage assessment for existing fuzzing harnesses, identifying under-covered areas • Possible modification or creation of harnesses as needed • Review of existing unit tests, noting significant untested behavior • Assessment of Firedancer Exploit Mitigations • Component Isolation and Sandboxing Evaluation • Assessment of key material isolation in Keyguard • Assessment of generated seccomp profiles • Evaluation of mitigation configurations in the build
Reporting and Analysis	
Analysis and Deliverables	• Status Reporting and Realtime Communication • Comprehensive Engagement Deliverable • Engagement Outbrief and Remediation Review

The ultimate goal of the assessment was to provide a clear picture of risks, vulnerabilities, and exposures as they relate to accepted security best practices, such as those created by the National Institute of Standards and Technology (NIST), Open Web Application Security Project (OWASP), or the Center for Internet Security (CIS). Augmenting these, Atredis Partners also draws on its extensive experience in secure development and in testing high-criticality applications and advanced exploitation.

Engagement Tasks

Atredis Partners performed the following tasks, at a high level, for in-scope targets during the engagement.

Application Penetration Testing

For relevant web applications, APIs, and web services, Atredis performed automated and manual application penetration testing of these components, applying generally accepted testing best practices as derived from OWASP and the Web Application Security Consortium (WASC).

Testing was performed from the perspective of an anonymous intruder, identifying scenarios from the perspective of an opportunistic, Internet-based threat actor with no knowledge of the environment, as well as, from the perspective a user working to laterally move through the environment to bypass security restrictions and user access levels.

Where relevant, Atredis Partners utilized both automated fuzzing and fault injection frameworks as well as purpose-built, task-specific testing tools tailored to the application and platforms under review.

Configuration and Architecture Review

Atredis Partners performed a high-level review of available documentation and configuration data with an eye toward the overall functional design and soundness of the implementation. A key aspect of this component was identifying gaps in the architecture and design regarding aspects of design that reduce overall defensibility, aimed at pointing out fundamental issues in the application architecture that should be addressed early in the development cycle as opposed to later when the platform is closer to a full production state.

While specific vulnerabilities may have been identified during the architecture and configuration review, the intent was less on finding individual defects and more on how the design of a given target affects its overall defensibility. Outcomes of the architecture review help inform testing objectives throughout the rest of the engagement while also helping the client define a long-term platform maturity and security design roadmap.

Source Code Analysis

Atredis reviewed the in-scope application source code, with an eye for security-relevant software defects. To aid in vulnerability discovery, application components were mapped out and modeled until a thorough understanding of execution flow, code paths, and application design and architecture was obtained. To aid in this process, the assessment team engaged key stakeholders and members of the development team where possible to provide structured walkthroughs and interviews, helping the team rapidly gain an understanding of the application’s design and development lifecycle.

Status Reporting and Realtime Communication

Atredis Partners scheduled regular status meetings with client representatives during the project and reported findings in realtime as soon as they were confirmed, via secure communication channels.

High Priority Issue Reporting

For Critical and High severity issues discovered during the engagement, Atredis delivered the details to the client once the vulnerability was confirmed. Atredis worked with the client's team to ensure understanding of the issue, the impact, and the proposed mitigation strategy.

Status Reports and Meetings

Atredis delivered, at a minimum, weekly status reports to the client and scheduled status meetings during the engagement. Standard Atredis status reports included project overview and tracking information such as percentage complete on each phase of the testing process. Reporting and meetings delivered relevant issues discovered that week as well as provided remediation guidance and additional information to the client team.

Executive Summary

This assessment was scoped as an "in-development" assessment of features that were ready for review at the time when the engagement was scoped. To facilitate this, Atredis was provided with a public GitHub repository[1] and a specific commit hash to review. The working commit hash was reviewed during each weekly call and adjusted as changes were made to the codebase. The commit hashes that were reviewed during the engagement are listed below.

- 89da411981808437f094de2682e70ee8d7e44d52 – March 13, 2024
- cc52c2c464f4e2e67813d86895247705c38907dd – April 19, 2024
- 30fb51e2634d8ca80de34e497169bf8a0f6183a7 – May 13, 2024
- 1d60058a9493cb890f07a4650ddf26e6e02cd226 – May 14, 2024
- ded1c05a40dfffa8882b388f6c7879a86edb4e0f – May 17, 2024

Since a large portion of the assessment focused on implementation of specific algorithms, most of the testing was performed through code review, unit testing, and fuzzing specific functions. To test API interactions and full transaction processing, Atredis also added debug outputs to the Firedancer node and performed dynamic testing with HTTP clients and a modified version of the Solana client.

Key Conclusions

The Firedancer implementation was well designed and implemented. It was also very well covered by unit tests and fuzzing harnesses: around 90 percent. The implementation effectively addressed security concerns related to general network applications, applications written in C, Linux operating environments, Web3 architecture, and cryptography.

Ultimately, Atredis identified one vulnerability that was remotely exploitable that allowed network connections to crash the Firedancer node. Other findings included one crash due to command-line argument parsing and a couple of informational findings related to potential memory leaks.

[1] https://github.com/firedancer-io/firedancer

Platform Overview

Tiles

In Firedancer, a tile is a subcomponent of the system that can be sandboxed as a separate process and can communicate with other tiles over a shared memory channel. Each tile process is bound to a dedicated processor core by default and multiple instances of each tile can be executed where desired. For each tile, a sandbox is set up with shared memory links to other tiles, file descriptor and seccomp filters, and a mux wrapper is set up with callbacks to the tile. The mux wrapper's main loop handles tick and credit tracking, signals, fragment processing, and tile callbacks. Each tile reads and writes data from its shared memory links when the callbacks are called. The Firedancer startup output below shows the shared memory links and running tiles with their link mappings.

```
LINKS
   0 ( 32 MiB):     net_quic  kind_id=0   wksp_id=0   depth=16384  mtu=2048       burst=1
   1 ( 32 MiB):    net_shred  kind_id=0   wksp_id=1   depth=16384  mtu=2048       burst=1
   2 ( 32 MiB):     quic_net  kind_id=0   wksp_id=0   depth=16384  mtu=2048       burst=1
   3 ( 32 MiB):    shred_net  kind_id=0   wksp_id=1   depth=16384  mtu=2048       burst=1
   4 ( 67 MiB):  quic_verify  kind_id=0   wksp_id=0   depth=16384  mtu=0          burst=16384
   5 ( 34 MiB): verify_dedup  kind_id=0   wksp_id=3   depth=16384  mtu=2086       burst=1
   6 ( 34 MiB):   dedup_pack  kind_id=0   wksp_id=4   depth=16384  mtu=2086       burst=1
   7 ( 34 MiB):  gossip_pack  kind_id=0   wksp_id=4   depth=16384  mtu=2086       burst=1
   8 (199 MiB):    stake_out  kind_id=0   wksp_id=10  depth=128    mtu=1608032    burst=1
   9 (  8 MiB):    pack_bank  kind_id=0   wksp_id=5   depth=128    mtu=65535      burst=1
  10 (  8 MiB):     bank_poh  kind_id=0   wksp_id=6   depth=128    mtu=65535      burst=1
  11 (  0 MiB):     poh_pack  kind_id=0   wksp_id=6   depth=128    mtu=40         burst=1
  12 (  1 GiB):    poh_shred  kind_id=0   wksp_id=8   depth=16384  mtu=65535      burst=1
  13 (189 MiB):   crds_shred  kind_id=0   wksp_id=8   depth=128    mtu=1527608    burst=1
  14 ( 12 GiB):  shred_store  kind_id=0   wksp_id=9   depth=65536  mtu=167168     burst=16388
  15 (  0 MiB):    quic_sign  kind_id=0   wksp_id=12  depth=128    mtu=130        burst=1
  16 (  0 MiB):    sign_quic  kind_id=0   wksp_id=13  depth=128    mtu=64         burst=1
  17 (  0 MiB):   shred_sign  kind_id=0   wksp_id=14  depth=128    mtu=32         burst=1
  18 (  0 MiB):   sign_shred  kind_id=0   wksp_id=15  depth=128    mtu=64         burst=1

TILES
   0 (  3 GiB):          net  kind_id=0   wksp_id=16  cpu_idx=1   out_link=-1  in=[-2, -3]  out=[ 0,  1]
   1 ( 17 GiB):         quic  kind_id=0   wksp_id=17  cpu_idx=2   out_link=4   in=[ 0, -16]  out=[ 2, 15]
   2 (  2 GiB):       verify  kind_id=0   wksp_id=18  cpu_idx=3   out_link=5   in=[-4]  out=[]
   3 (  3 GiB):        dedup  kind_id=0   wksp_id=19  cpu_idx=4   out_link=6   in=[ 5]  out=[]
   4 (  4 GiB):         pack  kind_id=0   wksp_id=20  cpu_idx=5   out_link=9   in=[ 6,  7, -11]  out=[]
   5 (  2 GiB):         bank  kind_id=0   wksp_id=21  cpu_idx=6   out_link=10  in=[ 9]  out=[]
   6 (  7 GiB):          poh  kind_id=0   wksp_id=22  cpu_idx=7   out_link=12  in=[10,  8,  9]  out=[11,  7,  8, 13]
   7 ( 19 GiB):        shred  kind_id=0   wksp_id=23  cpu_idx=8   out_link=14  in=[-1, 12,  8, 13, -18]  out=[ 3, 17]
   8 ( 13 GiB):        store  kind_id=0   wksp_id=24  cpu_idx=9   out_link=-1  in=[14]  out=[]
   9 ( 24 MiB):         sign  kind_id=0   wksp_id=25  cpu_idx=10  out_link=-1  in=[-15, -17]  out=[16, 18]
  10 (  3 GiB):       metric  kind_id=0   wksp_id=26  cpu_idx=11  out_link=-1  in=[]  out=[]
```

Tile topology output when Firedancer is started

While testing the tile communications, Atredis added debugging outputs to closely monitor fragment processing to fully understand the data flow and challenge/confirm assumptions.

```
DEBUG   2024-06-28 13:42:21.017187644 GMT+00 264286:264298 atredis:asym-research-solana:1
fd1:[group]:net:0 src/app/fdctl/run/tiles/fd_net.c(122)[net_rx_aio_send]: Atredis
net_rx_aio_send called
DEBUG   2024-06-28 13:42:21.017206742 GMT+00 264287:264302 atredis:asym-research-solana:2
fd1:[group]:quic:0 src/app/fdctl/run/tiles/fd_quic.c(302)[after_frag]: Atredis after_frag
called
DEBUG   2024-06-28 13:42:21.017207116 GMT+00 264286:264298 atredis:asym-research-solana:1
fd1:[group]:net:0 src/app/fdctl/run/tiles/fd_net.c(163)[net_rx_aio_send]: Atredis net tile
out to quic
DEBUG   2024-06-28 13:42:21.017252203 GMT+00 264287:264302 atredis:asym-research-solana:2
fd1:[group]:quic:0 src/app/fdctl/run/tiles/fd_quic.c(399)[quic_stream_receive]: Atredis
quic_stream_receive called
DEBUG   2024-06-28 13:42:21.017263479 GMT+00 264287:264302 atredis:asym-research-solana:2
fd1:[group]:quic:0 src/app/fdctl/run/tiles/fd_quic.c(440)[quic_stream_notify]: Atredis
quic_stream_notify called
DEBUG   2024-06-28 13:42:21.017274963 GMT+00 264287:264302 atredis:asym-research-solana:2
fd1:[group]:quic:0 src/app/fdctl/run/tiles/fd_quic.c(302)[after_frag]: Atredis after_frag
called
DEBUG   2024-06-28 13:42:21.017273982 GMT+00 264288:264295 atredis:asym-research-solana:3
fd1:[group]:verify:0 src/app/fdctl/run/tiles/fd_verify.c(73)[during_frag]: Atredis calling
fd_chunk_to_laddr to dedup link
DEBUG   2024-06-28 13:42:21.017291735 GMT+00 264288:264295 atredis:asym-research-solana:3
fd1:[group]:verify:0 src/app/fdctl/run/tiles/fd_verify.c(121)[after_frag]: Atredis
verifying transaction
DEBUG   2024-06-28 13:42:21.017296336 GMT+00 264287:264302 atredis:asym-research-solana:2
fd1:[group]:quic:0 src/app/fdctl/run/tiles/fd_quic.c(302)[after_frag]: Atredis after_frag
called
DEBUG   2024-06-28 13:42:21.017296981 GMT+00 264288:264295 atredis:asym-research-solana:3
fd1:[group]:verify:0 src/app/fdctl/run/tiles/fd_verify.h(79)[fd_txn_verify]: Atredis
verifying msg
010001033b6a27bcceb6a42d62a3a8d02a6f0d73653215771de243a63ac048a18b59da298a88e3dd7409f195fd5
2db2d3cba5d72ca6709bf1d94121bf3748801b40f6f5c0000000000000000000000000000000000000000000000
0000000000000000002e0fc94c02119dd533bd061d9f20a79a73ef6ffb0ee42b39035b23c9724822cd010202000
10c0200000000e8764817000000
DEBUG   2024-06-28 13:42:21.017308577 GMT+00 264288:264295 atredis:asym-research-solana:3
fd1:[group]:verify:0 src/app/fdctl/run/tiles/fd_verify.h(83)[fd_txn_verify]: Atredis
verifying signature
251vSVDm4nVNFHx7mviKPcpbLUe657Jm66jDzrbkytPWR8SPwLVJowjhWETEp8e51pQCfPEs26D7v9MMgWVA9RCS
DEBUG   2024-06-28 13:42:21.017395657 GMT+00 264288:264295 atredis:asym-research-solana:3
fd1:[group]:verify:0 src/app/fdctl/run/tiles/fd_verify.c(130)[after_frag]: Atredis calling
fd_dcache_compact_next
DEBUG   2024-06-28 13:42:21.017407249 GMT+00 264289:264300 atredis:asym-research-solana:4
fd1:[group]:dedup:0 src/app/fdctl/run/tiles/fd_dedup.c(121)[after_frag]: Atredis dedup
after_frag called
DEBUG   2024-06-28 13:42:21.017431474 GMT+00 264290:264296 atredis:asym-research-solana:5
fd1:[group]:pack:0 src/ballet/pack/fd_pack_cost.h(126)[fd_pack_compute_cost]: Atredis
computing cost for txn with signature
251vSVDm4nVNFHx7mviKPcpbLUe657Jm66jDzrbkytPWR8SPwLVJowjhWETEp8e51pQCfPEs26D7v9MMgWVA9RCS
DEBUG   2024-06-28 13:42:21.017454941 GMT+00 264290:264296 atredis:asym-research-solana:5
fd1:[group]:pack:0 src/ballet/pack/fd_pack_cost.h(177)[fd_pack_compute_cost]: Atredis pack
fee: 0, compute: 200000
DEBUG   2024-06-28 13:42:21.017460284 GMT+00 264290:264296 atredis:asym-research-solana:5
fd1:[group]:pack:0 src/ballet/pack/fd_pack.c(606)[fd_pack_estimate_rewards_and_compute]:
Atredis pack cost: 1473, rewards: 5000
DEBUG   2024-06-28 13:42:21.017622411 GMT+00 264285:264294 atredis:asym-research-solana:6
fd1:[group]:bank:0 src/app/fdctl/run/tiles/fd_bank.c(148)[hash_transactions]: Atredis
hashing txn with signature
251vSVDm4nVNFHx7mviKPcpbLUe657Jm66jDzrbkytPWR8SPwLVJowjhWETEp8e51pQCfPEs26D7v9MMgWVA9RCS
```

```
DEBUG   2024-06-28 13:42:21.017916455 GMT+00 264287:264302 atredis:asym-research-solana:2
fd1:[group]:quic:0 src/app/fdctl/run/tiles/fd_quic.c(483)[quic_tx_aio_send]: Atredis
quic_tx_aio_send called
DEBUG   2024-06-28 13:42:21.017928306 GMT+00 264287:264302 atredis:asym-research-solana:2
fd1:[group]:quic:0 src/app/fdctl/run/tiles/fd_quic.c(518)[quic_tx_aio_send]: Atredis
writing to mcache
DEBUG   2024-06-28 13:42:21.017938977 GMT+00 264286:264298 atredis:asym-research-solana:1
fd1:[group]:net:0 src/app/fdctl/run/tiles/fd_net.c(242)[route_loopback]: Atredis
route_loopback called
DEBUG   2024-06-28 13:42:21.017938913 GMT+00 264287:264302 atredis:asym-research-solana:2
fd1:[group]:quic:0 src/app/fdctl/run/tiles/fd_quic.c(483)[quic_tx_aio_send]: Atredis
quic_tx_aio_send called
DEBUG   2024-06-28 13:42:21.017955714 GMT+00 264286:264298 atredis:asym-research-solana:1
fd1:[group]:net:0 src/app/fdctl/run/tiles/fd_net.c(242)[route_loopback]: Atredis
route_loopback called
--- SNIP ---
DEBUG   2024-06-28 13:42:21.034639945 GMT+00 264286:264298 atredis:asym-research-solana:1
fd1:[group]:net:0 src/app/fdctl/run/tiles/fd_net.c(242)[route_loopback]: Atredis
route_loopback called
DEBUG   2024-06-28 13:42:21.137055632 GMT+00 264290:264296 atredis:asym-research-solana:5
fd1:[group]:pack:0 src/ballet/pack/fd_pack_cost.h(126)[fd_pack_compute_cost]: Atredis
computing cost for txn with signature
2Kccc8TpwaEsMEn7XXSSb7dRRq8LmCTWhSitXZmFmf9gdZv3PNb9xUAxiEj5uofop4FkjigA5VZPP3SC9HBdgHAX
DEBUG   2024-06-28 13:42:21.137085334 GMT+00 264290:264296 atredis:asym-research-solana:5
fd1:[group]:pack:0 src/ballet/pack/fd_pack_cost.h(177)[fd_pack_compute_cost]: Atredis pack
fee: 0, compute: 200000
DEBUG   2024-06-28 13:42:21.137092622 GMT+00 264290:264296 atredis:asym-research-solana:5
fd1:[group]:pack:0 src/ballet/pack/fd_pack.c(606)[fd_pack_estimate_rewards_and_compute]:
Atredis pack cost: 3443, rewards: 5000
INFO    2024-06-28 13:42:21.323086485 GMT+00 264285:264294 atredis:asym-research-solana:7
fd1:[group]:poh:0 src/app/fdctl/run/tiles/fd_poh.c(1309)[during_frag]:
done_packing(slot=21,seen_microblocks=4,microblocks_in_slot=4)
INFO    2024-06-28 13:42:21.323247479 GMT+00 264285:264294 atredis:asym-research-solana:7
fd1:[group]:poh:0 src/app/fdctl/run/tiles/fd_poh.c(847)[no_longer_leader]:
no_longer_leader(next_leader_slot=22)
DEBUG   2024-06-28 13:42:21.323312113 GMT+00 264291:264299 atredis:asym-research-solana:8
fd1:[group]:shred:0 src/disco/keyguard/fd_keyguard_client.c(25)[fd_keyguard_client_sign]:
Atredis keyguard sign
DEBUG   2024-06-28 13:42:21.323346177 GMT+00 264292:264301 atredis:asym-research-solana:10
fd1:[group]:sign:0 src/app/fdctl/run/tiles/fd_sign.c(120)[after_frag]: Atredis signing as
FD_KEYGUARD_ROLE_LEADER
INFO    2024-06-28 13:42:21.324682604 GMT+00 264285:535    atredis:asym-research-solana:f22
fd1:[group]:1    src/app/fdctl/run/tiles/fd_poh.c(847)[no_longer_leader]:
no_longer_leader(next_leader_slot=22)
INFO    2024-06-28 13:42:21.324712581 GMT+00 264285:535    atredis:asym-research-solana:f22
fd1:[group]:1    src/app/fdctl/run/tiles/fd_poh.c(914)[fd_ext_poh_reset]:
fd_ext_poh_reset(slot=22,next_leader_slot=22)
```

Sample Atredis Debug Outputs

Sandboxing

The Firedancer system uses sandboxing mechanisms provided by the Linux kernel to effectively isolate components, hoping to mitigate the impact of vulnerabilities in the codebase. Many components are run in "tiles", which can be contained in their own sandboxed processes. The tiles may also be run as threads in a developer build. These tiles are performant and use shared memory channels to communicate. In addition to the isolation provided by executing in separate processes, Firedancer further isolates the tiles by dropping privileges, enforcing seccomp (secure computing) rules, and utilizing namespaceing.

Any issues in the sandboxing code could mean insufficient constraint on the sandbox components, which could ease exploitation of vulnerabilities in the Firedancer system. The sandbox components were reviewed to identify any potential gaps or improvements.

The Firedancer tiles will drop privileges by downgrading to a user specified in the configuration (presumably a low-privileged user). The tiles also use the `PR_SET_NO_NEW_PRIVS` setting to prevent the tiles from being able to elevate their privilege in the future via a `setuid` binary or similar mechanism. The environment is also cleared, to not leak any sensitive environment information to a potentially compromised tile.

Each tile type has a special seccomp profile created for it, specifying some logic that will evaluate what system calls that tile will be able to access. Most tiles use a very reduced set of system calls. The logic in the seccomp filtering also somewhat validates the arguments passed to the system call, for example verifying that the `write` system call is only invoked with the logging file descriptor. Without access to system calls like `dup2` or `close` the tile will be unable to use `write` with any file object except the allowed logging file object. The seccomp profiles contain useful comments for understanding the tile's system call needs and are compiled into a logic to provide a seccomp BPF program for that tile's sandboxing. The profiles and compiled BPF programs were evaluated to ensure no dangerous system calls were available to any tile.

In addition to the seccomp filtering, a sandboxed tile will use the `proc` filesystem to verify its allowed file descriptors when setting up the sandbox, to ensure that only the expected file descriptors are present in the sandboxed process.

The sandboxing puzzle is completed with use of Linux namespaces. New namespaces are created for the tile including network, user, and cgroup namespaces. The filesystem root is pivoted to a unique directory and access to the previous filesystem is revoked by unmounting the previous filesystem root.

Overall, the sandboxing is effective, with the acknowledged risk that a compromised tile will be able to communicate with other tiles through shared memory channels. Any exploitation that would cross from one tile to another component would require vulnerabilities reachable via the communication channels used between components of the Firedancer system, or vulnerabilities reachable in the host machine with the limited number of system calls available.

Network Data Processing

The Firedancer system communicates over the network with other validators and untrusted parties. Firedancer seeks to make this communication path rapid and secure, utilizing its own packet processing and implementation of network protocols. The protocols implemented are generally optimized for Firedancer's workload, reducing attack surface and unnecessary processing time for unneeded features.

Firedancer bypasses much of the Linux Kernel's network processing logic by using XDP (eXpress Data Path), which allows Firedancer to use a small bit of logic in the form of an eBPF (extended Berkley Packet Filter) program. This program is hooked into the Linux Kernel's network processing and will direct Firedancer packets to a user-space portion of the Firedancer system.

The source file `src/waltz/xdp/fd_xdp_redirect_prog.c` contains the logic for the eBPF that will redirect packets into Firedancer. The function `fd_xdp_redirect` performs some basic checks and will forward any incoming IPv4 UDP packets that have a known destination port.

The eBPF engine used by the Linux Kernel seeks to verify any inserted eBPF, to ensuring the code cannot deadlock or access invalid memory. Issues in the Firedancer eBPF could result in packets either forwarded when they should not have been or not forwarded when they should have been. Incorrect forwarding could result in degraded communications for applications on the system, and the logic was reviewed to ensure proper routing logic.

Net Tile

The user-space portion that next handles the incoming packets is an instance of the "net" tile. The instances of the "net" tile have a registered XDP socket that uses a ring buffer system to pull out the forwarded packets. The net tile will verify some pieces of the packet before copying it into a shared memory cache "link" used to communicate between tiles. Either the link to the "shred" tile or the "QUIC" tile are used, depending on the destination port.

Ring buffer logic can often be prone to race conditions and memory corruption issues, so the XDP socket usage was checked for possible issues. The lifetime of the frames used by the XDP sockets were validated to ensure they were not improperly reused.

Because the "net" tile uses XDP for network communication, it must be able to generate the network headers for any packets sent back out to the network. This includes the Ethernet header, which must include the MAC address of the next hop for the packet. If the host system already is aware of the MAC address for the next destination, the Firedancer system can retrieve it from the host's ARP (Address Resolution Protocol) table. If the destination MAC address is not known, an entry must be made in the host's ARP table, and an ARP probe must be broadcast to the network. When a response is received by the host, it will populate the entry in the host's table, which can later be retrieved by the "net" tile.

Issues in Firedancer's ARP system could lead to degradation or denial of service if it cannot resolve needed addresses, or if the network could be flooded with unnecessary ARP probes. Firedancer's communications with the host and creation of ARP requests was reviewed for any such issues. See Net Tile: ARP Table Size Error Ignored.

QUIC Tile

The QUIC tile handles both the legacy path for transaction data, and the QUIC path. The legacy path uses the data provided in the UDP packets and plaintext transaction data, which is passed to the transaction re-assembler, given some initial validation, and passed to the "verify" tile. The QUIC path processes incoming UDP packets according to the QUIC and TLS protocols before passing the data to the transaction re-assembler for initial validation and forwarding to the "verify" tile.

In either path, failure to properly validate and parse the transaction data could lead to memory corruption or denial of service. The transaction parsing was reviewed for such issues. The Firedancer repository contains a fuzz harness for the initial transaction validation. The fuzz harness and coverage information were reviewed to try and identify any bottlenecks to fully testing this sensitive code.

The QUIC path also contains sensitive code, as issues in the implementation of QUIC, TLS, and other dependent protocols could lead to issues such as memory corruption, information disclosure, or denial of service. The QUIC and TLS implementations both do not implement the complete specification of their respective protocols, with features irrelevant to Firedancer's use case being unimplemented. Any deviations from the specifications of the RFC were reviewed to ensure lack of those features would not introduce vulnerabilities. For the TLS implementation certificate chains were not validated, and steps normally required by a client such as hostname validation were missing from the TLS logic. This is not a security issue in Firedancer's case, as the underlying transaction data contains all the information needed to authenticate incoming data.

Both the QUIC and TLS parsing make extensive use of C language macros to define secure parsing helpers. The systems created were effective at ensuring that parsed structures were validated, and parsing could not continue past the end of the input buffer.

Multiple fuzzing harnesses exist for the QUIC, TLS, and x509 certificate parsing logic. As with other sections, the fuzzing coverage was reviewed for possible bottlenecks, and the fuzzing harnesses were reviewed for any obviously missing functionality. The fuzzing logic provided for these components are especially good examples of effective harnesses.

Verify Tile

The verify tile verifies the Ed25519 signatures of transactions, performs basic duplicate transaction filtering, and passes the fragment off to the Dedup tile.

```
    /* The first signature is the transaction id, i.e. a unique identifier.
       So use this to do a quick dedup of ha traffic. */

    /* TODO: use more than 64 bits to dedup. */
    ulong ha_dedup_tag = *((ulong *)signatures);
    int ha_dup;
    FD_FN_UNUSED ulong tcache_map_idx = 0; /* ignored */
    FD_TCACHE_QUERY( ha_dup, tcache_map_idx, ctx->tcache_map, ctx->tcache_map_cnt,
ha_dedup_tag );
    if( FD_UNLIKELY( ha_dup ) ) {
      return FD_TXN_VERIFY_DEDUP;
    }

    /* Verify signatures */
    int res = fd_ed25519_verify_batch_single_msg( msg, msg_sz, signatures, pubkeys, ctx-
>sha, signature_cnt );
    if( FD_UNLIKELY( res != FD_ED25519_SUCCESS ) ) {
      return FD_TXN_VERIFY_FAILED;
    }

    /* Insert into the tcache to dedup ha traffic.
       The dedup check is repeated to guard against duped txs verifying signatures at the
same time */
    FD_TCACHE_INSERT( ha_dup, *ctx->tcache_sync, ctx->tcache_ring, ctx->tcache_depth, ctx-
>tcache_map, ctx->tcache_map_cnt, ha_dedup_
    if( FD_UNLIKELY( ha_dup ) ) {
      return FD_TXN_VERIFY_DEDUP;
    }
```

Transaction verification

For signature verification, Firedancer uses a custom Ed25519 implementation that has been built to match the Dalek[2] Rust implementation that is used in Solana. The Firedancer implementation is well-covered by an extensive unit testing and fuzz testing suite. Aside from typical targeted unit tests and fuzz harnesses, this suite includes a differential fuzzing harness that compares the outputs of the dynamically loaded Dalek implementation to the Firedancer implementation. Atredis manually reviewed the Firedancer ED25519 implementation for elliptic curve cryptography vulnerabilities as well as verifying that it matches the Dalek implementation.

For interactive testing, Atredis used a modified Solana client to perform signature bit fuzzing and verified that only one signature value is accepted per signed message.

```
--- snip ---

Updated signature bytes: [22, B4, CE, FC, 5D, BA, 21, 73, AF, 5E, AF, C1, E1, A5, 8A, AD,
FB, 64, 33, 8E, 9F, 93, 3C, B9, 39, 4F, 08, F3, 19, E6, 3F, AA, B8, E9, 19, C5, EB, 80, 47,
C2, 53, 6E, 4F, 0B, 0C, BD, 86, 6A, 92, FC, 41, CD, 0C, 30, A7, BC, 62, D7, AD, FC, 22, F4,
0B, 07]
Updated signature:
hFFDmo1iB1VZuRgya5jsVRAxXZ47zkxiJFJ3WqL4A2RTWCP8Ahb4rTP1qRKgQgr4oCN5DXaKpYP2AV6FzqxawjL
Fuzz offset: 62, value 12
Expected error: Error { request: Some(SendTransaction), kind: RpcError(RpcResponseError {
code: -32003, message: "Transaction signature verification failure", data: Empty }) }
Updated signature bytes: [22, B4, CE, FC, 5D, BA, 21, 73, AF, 5E, AF, C1, E1, A5, 8A, AD,
FB, 64, 33, 8E, 9F, 93, 3C, B9, 39, 4F, 08, F3, 19, E6, 3F, AA, B8, E9, 19, C5, EB, 80, 47,
C2, 53, 6E, 4F, 0B, 0C, BD, 86, 6A, 92, FC, 41, CD, 0C, 30, A7, BC, 62, D7, AD, FC, 22, F4,
0C, 07]
Updated signature:
hFFDmo1iB1VZuRgya5jsVRAxXZ47zkxiJFJ3WqL4A2RTWCP8Ahb4rTP1qRKgQgr4oCN5DXaKpYP2AV6Fzqxawok
Fuzz offset: 62, value 13
Expected error: Error { request: Some(SendTransaction), kind: RpcError(RpcResponseError {
code: -32003, message: "Transaction signature verification failure", data: Empty }) }

--- snip ---

Updated signature bytes: [22, B4, CE, FC, 5D, BA, 21, 73, AF, 5E, AF, C1, E1, A5, 8A, AD,
FB, 64, 33, 8E, 9F, 93, 3C, B9, 39, 4F, 08, F3, 19, E6, 3F, AA, B8, E9, 19, C5, EB, 80, 47,
C2, 53, 6E, 4F, 0B, 0C, BD, 86, 6A, 92, FC, 41, CD, 0C, 30, A7, BC, 62, D7, AD, FC, 22, F4,
6F, 0D]
Updated signature:
hFFDmo1iB1VZuRgya5jsVRAxXZ47zkxiJFJ3WqL4A2RTWCP8Ahb4rTP1qRKgQgr4oCN5DXaKpYP2AV6Fzqxb5Lp
Fuzz offset: 63, value 14
Expected error: Error { request: Some(SendTransaction), kind: RpcError(RpcResponseError {
code: -32003, message: "Transaction signature verification failure", data: Empty }) }
Updated signature bytes: [22, B4, CE, FC, 5D, BA, 21, 73, AF, 5E, AF, C1, E1, A5, 8A, AD,
FB, 64, 33, 8E, 9F, 93, 3C, B9, 39, 4F, 08, F3, 19, E6, 3F, AA, B8, E9, 19, C5, EB, 80, 47,
C2, 53, 6E, 4F, 0B, 0C, BD, 86, 6A, 92, FC, 41, CD, 0C, 30, A7, BC, 62, D7, AD, FC, 22, F4,
6F, 0E]
```

[2] https://github.com/dalek-cryptography/ed25519-dalek

```
Updated signature:
hFFDmo1iB1VZuRgya5jsVRAxXZ47zkxiJFJ3WqL4A2RTWCP8Ahb4rTP1qRKgQgr4oCN5DXaKpYP2AV6Fzqxb5Lq
Fuzz offset: 63, value 15
Expected error: Error { request: Some(SendTransaction), kind: RpcError(RpcResponseError {
code: -32003, message: "Transaction signature verification failure", data: Empty }) }
Updated signature bytes: [22, B4, CE, FC, 5D, BA, 21, 73, AF, 5E, AF, C1, E1, A5, 8A, AD,
FB, 64, 33, 8E, 9F, 93, 3C, B9, 39, 4F, 08, F3, 19, E6, 3F, AA, B8, E9, 19, C5, EB, 80, 47,
C2, 53, 6E, 4F, 0B, 0C, BD, 86, 6A, 92, FC, 41, CD, 0C, 30, A7, BC, 62, D7, AD, FC, 22, F4,
6F, 0F]
Updated signature:
hFFDmo1iB1VZuRgya5jsVRAxXZ47zkxiJFJ3WqL4A2RTWCP8Ahb4rTP1qRKgQgr4oCN5DXaKpYP2AV6Fzqxb5Lr
```

Sample of Dynamic Signature Fuzzing Output

Dedup Tile

The Dedup tile is a simple component that performs basic duplicate transaction filtering by caching and checking transaction signatures that have recently been processed. Testing for this component was primarily done through code review.

```
int is_dup;
FD_TCACHE_INSERT( is_dup, *ctx->tcache_sync, ctx->tcache_ring, ctx->tcache_depth, ctx-
>tcache_map, ctx->tcache_map_cnt, *opt_sig );
*opt_filter = is_dup;
if( FD_LIKELY( !*opt_filter ) ) {
  *opt_chunk     = ctx->out_chunk;
  *opt_sig       = 0; /* indicate this txn is coming from dedup, and has already been
parsed */
  ctx->out_chunk = fd_dcache_compact_next( ctx->out_chunk, *opt_sz, ctx->out_chunk0, ctx-
>out_wmark );
}
```

Simple duplication checking logic

Pack Tile

The Pack tile is a moderately complex sub-component that arranges verified transactions into optimized microblocks. As part of this functionality, it performs careful calculation of block sizes, transaction counts, time delays, and calculates fees and rewards. The results are published to the Bank tile(s).

```
 /* There's overhead associated with each microblock the bank tile tries
     to execute it, so the optimal strategy is not to produce a microblock
     with a single transaction as soon as we receive it.  Basically, if we
     have less than 31 transactions, we want to wait a little to see if we
     receive additional transactions before we schedule a microblock.  We
     can model the optimum amount of time to wait, but the equation is
     complicated enough that we want to compute it before compile time.
     wait_duration[i] for i in [0, 31] gives the time in nanoseconds pack
     should wait after receiving its most recent transaction before
     scheduling if it has i transactions available.  Unsurprisingly,
     wait_duration[31] is 0.  wait_duration[0] is ULONG_MAX, so we'll
     always wait if we have 0 transactions. */
FD_IMPORT( wait_duration, "src/ballet/pack/pack_delay.bin", ulong, 6, "" );

--- snip ---

ctx->wait_duration_ticks[ 0 ] = ULONG_MAX;
for( ulong i=1UL; i<MAX_TXN_PER_MICROBLOCK+1UL; i++ ) {
  ctx->wait_duration_ticks[ i ]=(ulong)(fd_tempo_tick_per_ns( NULL )*(double)wait_duration[
i ] + 0.5);
}
```

Tick timing logic

```
#define VOTE_PROG_COST 2100UL

#define MAP_PERFECT_0  ( STAKE_PROG_ID           ), .cost_per_instr=        750UL
#define MAP_PERFECT_1  ( CONFIG_PROG_ID          ), .cost_per_instr=        450UL
#define MAP_PERFECT_2  ( VOTE_PROG_ID            ), .cost_per_instr=VOTE_PROG_COST
#define MAP_PERFECT_3  ( SYS_PROG_ID             ), .cost_per_instr=        150UL
#define MAP_PERFECT_4  ( COMPUTE_BUDGET_PROG_ID  ), .cost_per_instr=        150UL
#define MAP_PERFECT_5  ( ADDR_LUT_PROG_ID        ), .cost_per_instr=        750UL
#define MAP_PERFECT_6  ( BPF_UPGRADEABLE_PROG_ID ), .cost_per_instr=       2370UL
#define MAP_PERFECT_7  ( BPF_LOADER_1_PROG_ID    ), .cost_per_instr=       1140UL
#define MAP_PERFECT_8  ( BPF_LOADER_2_PROG_ID    ), .cost_per_instr=        570UL
#define MAP_PERFECT_9  ( LOADER_V4_PROG_ID       ), .cost_per_instr=       2000UL
#define MAP_PERFECT_10 ( KECCAK_SECP_PROG_ID     ), .cost_per_instr=        720UL
#define MAP_PERFECT_11 ( ED25519_SV_PROG_ID      ), .cost_per_instr=        720UL

--- snip ---

for( ulong i=0UL; i<txn->instr_cnt; i++ ) {
  instr_data_sz += txn->instr[i].data_sz;

  ulong prog_id_idx = (ulong)txn->instr[i].program_id;
  fd_acct_addr_t const * prog_id = addr_base + prog_id_idx;

  /* Lookup prog_id in hash table */

  fd_pack_builtin_prog_cost_t null_row[1] = {{{ 0 }, 0UL }};
  fd_pack_builtin_prog_cost_t const * in_tbl = fd_pack_builtin_query( prog_id, null_row );
  builtin_cost     +=  in_tbl->cost_per_instr;
  non_builtin_cnt += !in_tbl->cost_per_instr; /* The only one with no cost is the null one
*/

  if( FD_UNLIKELY( in_tbl==compute_budget_row ) )
    if( FD_UNLIKELY( 0==fd_compute_budget_program_parse( txnp->payload+txn-
>instr[i].data_off, txn->instr[i].data_sz, cbp ) ) )
      return 0UL;

  vote_instr_cnt += (ulong)(in_tbl==vote_row);

}
```

Instruction costs

Bank Tile

The Bank tile reads transactions from the Pack tile and initializes a Solana `solana-tx-message-v1` message for each transaction using the Blake3 hash function to hash the message contents. The message initialization function contains logic to support building both `legacy` and `0` Solana transaction versions[3] based on the input.

[3] https://solana.com/docs/advanced/versions

```
fd_blake3_init( blake3 );
fd_blake3_append( blake3, "solana-tx-message-v1", 20UL );
fd_blake3_append( blake3, payload + txn->message_off, payload_sz - txn->message_off );
fd_blake3_fini( blake3, out_txn->message_hash );
```

Solana message initialization

Once the transaction messages are built, it delegates `fd_ext_bank_pre_balance_info`, `fd_ext_bank_load_and_execute_txns`, and `fd_ext_bank_commit_txns` to the Solana validator over the Solana and the Rust FFI Interface.

```
for( ulong i=0UL; i<txn_cnt; i++ ) {
  fd_txn_p_t * txn = (fd_txn_p_t *)( dst + (i*sizeof(fd_txn_p_t)) );

  void * abi_txn = ctx->txn_abi_mem + (sanitized_txn_cnt*FD_BANK_ABI_TXN_FOOTPRINT);
  void * abi_txn_sidecar = ctx->txn_sidecar_mem + sidecar_footprint_bytes;

  int result = fd_bank_abi_txn_init( abi_txn, abi_txn_sidecar, ctx->_bank, ctx->blake3,
txn->payload, txn->payload_sz, TXN(txn),
  ctx->metrics.txn_load_address_lookup_tables[ result ]++;
  if( FD_UNLIKELY( result!=FD_BANK_ABI_TXN_INIT_SUCCESS ) ) continue;

  txn->flags |= FD_TXN_P_FLAGS_SANITIZE_SUCCESS;

  fd_txn_t * txn1 = TXN(txn);
  sidecar_footprint_bytes += FD_BANK_ABI_TXN_FOOTPRINT_SIDECAR( txn1->acct_addr_cnt, txn1-
>addr_table_adtl_cnt, txn1->instr_cnt,
  sanitized_txn_cnt++;
}

/* Just because a transaction was executed doesn't mean it succeeded,
   but all executed transactions get committed. */
int load_results[ MAX_TXN_PER_MICROBLOCK ] = {0};
int executing_results[ MAX_TXN_PER_MICROBLOCK ] = {0};
int executed_results[ MAX_TXN_PER_MICROBLOCK ] = {0};

void * pre_balance_info = fd_ext_bank_pre_balance_info( ctx->_bank, ctx->txn_abi_mem,
sanitized_txn_cnt );

void * load_and_execute_output = fd_ext_bank_load_and_execute_txns( ctx->_bank,
                                                                    ctx->txn_abi_mem,
                                                                    sanitized_txn_cnt,
                                                                    load_results,
                                                                    executing_results,
                                                                    executed_results );
```

Initializing and executing transactions over the Rust FFI

After transactions have been executed, the Bank tile creates a Merkle tree of the transaction signatures, using SHA-256 to create the leaves, and passes it on the PoH (Proof of History) tile.

PoH Tile

The PoH (Proof of History[4]) tile performs the core Solana proof work of managing leader duties and producing microblocks and ticks. Data is loaded from the Stake, Pack, or Bank workspaces and Ticks and Microblocks are published to the Shred tile. Atredis reviewed the source code of the PoH tile to understand data flow and to look for memory corruption or processing logic issues, but runtime testing was minimal.

```
uchar data[ 64 ];
fd_memcpy( data, ctx->hash, 32UL );
fd_memcpy( data+32UL, ctx->_microblock_trailer->hash, 32UL );
fd_sha256_hash( data, 64UL, ctx->hash );

ctx->hashcnt++;
ulong hashcnt_delta = ctx->hashcnt - ctx->last_hashcnt;
ctx->last_hashcnt = ctx->hashcnt;

/* The hashing loop above will never leave us exactly one away from
    crossing a tick boundary, so this increment will never cause the
    current tick (or the slot) to change, except in low power mode
    for development, in which case we do need to register the tick
    with the leader bank.  We don't need to publish the tick since
    sending the microblock below is the publishing action. */
if( FD_UNLIKELY( !(ctx->hashcnt%ctx->hashcnt_per_tick) ) ) {
  fd_ext_poh_register_tick( ctx->current_leader_bank, ctx->hash );
  if( FD_UNLIKELY( ctx->hashcnt>=(ctx->next_leader_slot_hashcnt+ctx->hashcnt_per_slot) ) )
{
    /* We ticked while leader and are no longer leader... transition
        the state machine. */
    no_longer_leader( ctx );
  }
}

publish_microblock( ctx, mux, *opt_sig, target_slot, hashcnt_delta, txn_cnt );
```

Tick and microblock publishing

[4] https://solana.com/news/proof-of-history

Shred Tile

The Shred tile is a complex sub-component that creates Solana Shreds[5], prioritizes destination validators based on stake, adds them to the block store through the Store tile, and propagates them to the network via the Net tile. Atredis reviewed the source code of the Shred tile to understand data flow and to look for memory corruption or processing logic issues. However, some of the functions are very complex and did not receive full runtime coverage. For example, the `fd_fec_resolver_add_shred` function is 300+ lines and `fd_shred_dest_compute_children` is 150 lines of code.

Store Tile

The Store tile is a thin wrapper that delegates block storage functionality to the Solana node through `fd_ext_blockstore_insert_shreds` calls over the Solana and the Rust FFI Interface.

```
static inline void
after_frag( void *              _ctx,
            ulong               in_idx,
            ulong               seq,
            ulong *             opt_sig,
            ulong *             opt_chunk,
            ulong *             opt_sz,
            ulong *             opt_tsorig,
            int *               opt_filter,
            fd_mux_context_t * mux ) {
  (void)in_idx;
  (void)seq;
  (void)opt_sig;
  (void)opt_chunk;
  (void)opt_tsorig;
  (void)opt_filter;
  (void)mux;

  fd_store_ctx_t * ctx = (fd_store_ctx_t *)_ctx;

  fd_shred34_t * shred34 = (fd_shred34_t *)ctx->mem;

  FD_TEST( shred34->shred_sz<=shred34->stride );
  if( FD_LIKELY( shred34->shred_cnt ) ) {
    FD_TEST( shred34->offset<*opt_sz  );
    FD_TEST( shred34->shred_cnt<=34UL );
    FD_TEST( shred34->stride==sizeof(shred34->pkts[0]) );
  }

  /* No error code because this cannot fail. */
  fd_ext_blockstore_insert_shreds( fd_ext_blockstore, shred34->shred_cnt, ctx->mem+shred34-
>offset, shred34->shred_sz, shred34->stride );

  FD_MCNT_INC( STORE_TILE, TRANSACTIONS_INSERTED, shred34->est_txn_cnt );
```

[5] https://github.com/solana-foundation/specs/blob/main/p2p/shred.md

```
}
```

Store tile functionality

Sign Tile

The Sign tile is responsible for all Firedancer transaction signing including the Leader, TLS, and Gossip roles. It configures the Ed25519 signing context, uses Keyguard to load the private signing key into memory and then calls the `fd_ed25519_sign` function to sign payloads. Signature testing is described in the Verify Tile section.

```
switch( ctx->in_role[ in_idx ] ) {
  case FD_KEYGUARD_ROLE_LEADER:
    fd_memcpy( ctx->_data, ctx->in_data[ in_idx ], 32UL );
    break;
  case FD_KEYGUARD_ROLE_TLS:
    fd_memcpy( ctx->_data, ctx->in_data[ in_idx ], 130UL );
    break;
  case FD_KEYGUARD_ROLE_GOSSIP:
    fd_memcpy( ctx->_data, ctx->in_data[ in_idx ], sz );
    break;
    case FD_KEYGUARD_ROLE_REPAIR:
    fd_memcpy( ctx->_data, ctx->in_data[ in_idx ], sz );
    break;
  default:
    FD_LOG_CRIT(( "unexpected link role %lu", ctx->in_role[ in_idx ] ));
}
```

Setting the ED25519 Context

```
uchar const * identity_key = fd_keyload_load( tile->sign.identity_key_path, /* pubkey only:
*/ 0 );
ctx->private_key = identity_key;
```

Loading the private key with Keyguard

```
switch( ctx->in_role[ in_idx ] ) {
  case FD_KEYGUARD_ROLE_LEADER: {
    if( FD_UNLIKELY( !fd_keyguard_payload_authorize( ctx->_data, 32UL,
FD_KEYGUARD_ROLE_LEADER ) ) ) {
      FD_LOG_EMERG(( "fd_keyguard_payload_authorize failed" ));
    }
    fd_ed25519_sign( ctx->out[ in_idx ].data, ctx->_data, 32UL, ctx->public_key, ctx-
>private_key, ctx->sha512 );
    break;
  }
  case FD_KEYGUARD_ROLE_TLS: {
    if( FD_UNLIKELY( !fd_keyguard_payload_authorize( ctx->_data, 130UL,
FD_KEYGUARD_ROLE_TLS ) ) ) {
      FD_LOG_EMERG(( "fd_keyguard_payload_authorize failed" ));
    }
    fd_ed25519_sign( ctx->out[ in_idx ].data, ctx->_data, 130UL, ctx->public_key, ctx-
>private_key, ctx->sha512 );
    break;
  }
  case FD_KEYGUARD_ROLE_GOSSIP: {
    if( FD_UNLIKELY( !fd_keyguard_payload_authorize( ctx->_data, *opt_sz,
FD_KEYGUARD_ROLE_GOSSIP ) ) ) {
      FD_LOG_EMERG(( "fd_keyguard_payload_authorize failed" ));
      }
    if ( fd_keyguard_payload_matches_ping_msg( ctx->_data, *opt_sz ) ) {
      /* Gossip tile sends the sh256 pre-image for ping/pong msgs. */
      uchar hash[32];
      fd_sha256_hash( ctx->_data, *opt_sz, hash );

      fd_ed25519_sign( ctx->out[ in_idx ].data, hash, 32UL, ctx->public_key, ctx-
>private_key, ctx->sha512 );
    } else {
      fd_ed25519_sign( ctx->out[ in_idx ].data, ctx->_data, *opt_sz, ctx->public_key, ctx-
>private_key, ctx->sha512 );
    }
  }
}
```

Transaction signing

Metrics tile

The Metrics tile is responsible for consolidating metrics data reported by tiles and making them available for consumption via an embedded HTTP server and a single `/metrics` endpoint. Metrics data is reported in the Prometheus plain text key-value format and are not behind any form of authentication making them public accessible to all users that can reach the HTTP server of default port 7999 bound to the public interface. A sample of the data and format used is shown below.

```
# HELP tile_pid The process ID of the tile.
# TYPE tile_pid gauge
tile_pid{kind="net",kind_id="0"} 76882
tile_pid{kind="quic",kind_id="0"} 76880
tile_pid{kind="verify",kind_id="0"} 76891
tile_pid{kind="verify",kind_id="1"} 76887
tile_pid{kind="verify",kind_id="2"} 76885
tile_pid{kind="verify",kind_id="3"} 76879
tile_pid{kind="verify",kind_id="4"} 76892
tile_pid{kind="verify",kind_id="5"} 76881
tile_pid{kind="verify",kind_id="6"} 76886
tile_pid{kind="verify",kind_id="7"} 76884
tile_pid{kind="verify",kind_id="8"} 76900
tile_pid{kind="verify",kind_id="9"} 76907
tile_pid{kind="dedup",kind_id="0"} 76902
tile_pid{kind="pack",kind_id="0"} 76893
tile_pid{kind="bank",kind_id="0"} 76883
tile_pid{kind="bank",kind_id="1"} 76883
tile_pid{kind="poh",kind_id="0"} 76883
tile_pid{kind="shred",kind_id="0"} 76895
tile_pid{kind="store",kind_id="0"} 76883
tile_pid{kind="sign",kind_id="0"} 76890
tile_pid{kind="metric",kind_id="0"} 76906

# HELP quic_sent_packets Number of IP packets sent.
# TYPE quic_sent_packets counter
quic_sent_packets{kind="quic",kind_id="0"} 3
```

Metrics Data in Prometheus format

Atredis did not identify the publishing of any sensitive material that would directly aid an attacker. However, there are no settings or controls provided to node administrators that allow for the filtering of metrics data beyond specifying the listening HTTP port.

The attack surface exposed by this additional interface was enumerated to determine if the request handling was safe from malformed requests and resource exhaustion attack vectors. Some investigated attack scenarios include header parsing errors, excessively large headers, streaming large HTTP bodies, and creating many simultaneous connections in efforts to use all available file descriptors. The `/metrics` endpoint does not accept any additional arguments, reducing the overall attack surface.

Keyguard

Keyguard is meant to be a single point for handling identity key material. But in practice, the Sign, Shred, PoH, Repair, Gossip and QUIC tiles call the `fd_keyload_load` function, then store the key material in their own memory. The function provides a `public_key_only` parameter that causes only the public key to be returned, which is how the function is called in most cases and only the Sign and Repair[6] tiles request the private key. However, Atredis noted that when the `public_key_only` parameter is specified, the private key is still loaded from the key file into memory, then zeroed out. This does not seem to currently represent a problem, but it does seem like an unnecessary exposure of the private key.

```
uchar const * FD_FN_SENSITIVE
fd_keyload_load( char const * key_path,
                 int          public_key_only ) {
  /* Load the signing key. Since this is key material, we load it into
     its own page that's non-dumpable, readonly, and protected by guard
     pages. */
  uchar * key_page = fd_sandbox_alloc_protected_pages( 1UL, 2UL );

  read_key( key_path, key_page );

  if( public_key_only ) explicit_bzero( key_page, 32UL );

  /* For good measure, make the key page read-only */
  if( FD_UNLIKELY( mprotect( key_page, 4096UL, PROT_READ ) ) )
    FD_LOG_ERR(( "mprotect failed (%i-%s)", errno, fd_io_strerror( errno ) ));

  if( public_key_only ) return key_page+32UL;
  else                  return key_page;
}
```

Keyguard loading the private and public key from file

Keyguard also provides some functionality for performing authorization checks, as shown below.

[6] The Repair tile was not in scope for this assessment

```
FD_FN_PURE int
fd_keyguard_payload_authorize( uchar const * data,
                                ulong         sz,
                                int           role ) {
  switch( role ) {
    case FD_KEYGUARD_ROLE_VOTER: return fd_keyguard_payload_matches_txn_msg( data, sz );
    case FD_KEYGUARD_ROLE_GOSSIP: return fd_keyguard_payload_matches_gossip_msg( data, sz );
    case FD_KEYGUARD_ROLE_LEADER: return fd_keyguard_payload_matches_shred( data, sz );
    case FD_KEYGUARD_ROLE_TLS: return fd_keyguard_payload_matches_tls_cv( data, sz );
    case FD_KEYGUARD_ROLE_X509_CA: return fd_keyguard_payload_matches_x509_csr( data, sz );
    default: return 0;
}
```

Keyguard authorization checks

Solana and the Rust FFI Interface

Firedancer builds and links against a modified version of the Solana Labs Rust validator. This is intended to act as a stopgap to retain functionality not yet implemented in Firedancer such as transaction execution, gossip, and RPC. The Solana Labs repository was forked at commit `2c921e44` and now exports 14 functions which the Firedancer C code base can access through the Rust Foreign Function Interface (FFI).

Atredis reviewed the changes made to Solana through static analysis to ensure no security issues were introduced since forking Solana. This was also done to make recommendations for improving code quality and avoiding common pitfalls by following official documentation and best practices.

The Rust FFI allows for easy interfacing between Firedancer's C code and existing Solana Rust code. An example of an exported Rust function in Solana is shown below.

```
#[no_mangle]
pub extern "C" fn fd_ext_bank_pre_balance_info( bank: *const std::ffi::c_void, txns: *const
std::ffi::c_void, txn_count: u64 ) -> *mut std::ffi::c_void {
    use solana_sdk::transaction::SanitizedTransaction;
    use std::borrow::Cow;
    use std::sync::atomic::Ordering;
```

Exported Rust Function Callable from C Code

Rust can also call into C code in Firebase. For example, sending gossiped votes.

```
extern "C" {
    fn fd_ext_poh_publish_gossip_vote(data: *const u8, len: usize);
}
fd_ext_poh_publish_gossip_vote(data.as_ptr(), data.len());
```

Rust Code Calling C Code Using FFI

In doing so, several of Rust's safety guarantees are bypassed which can introduce potential security risks and undefined behavior. The Firedancer development team is acutely aware of the FFI pitfalls as demonstrated in code. Object ownership and lifetime management has been considered and memory is correctly allocated and deallocated across the FFI boundary. Type safety for exported functions also appears to have been implemented correctly. The use of recommended `c_void` raw pointers is used consistently to pass objects across the FFI boundary. Atredis did note that most of these cases would benefit from an additional null check before dereferencing in Solana: Unvalidated Raw Pointers at FFI Boundary.

Apart from the FFI boundary, other areas were considered, such as the introduction of unsafe functions like `std::mem::forget`, but these were ultimately found not to expose any additional attack surface. The current Solana Labs implementation was also reviewed for commits that may have addressed security related issues in code that may not have received a patch in Firedancer. Imported crates were also analyzed for known security issues but none were identified that would adversely affect Firedancer.

Reimplemented Algorithms

In the tested version of Firedancer, the following algorithms have been reimplemented to support fast transactions:

- Ed25519
- Base58
- Base64
- ChaCha20
- Hex
- HMAC
- Reed Solomon
- SHA256
- SHA512
- QUIC

For each of these algorithms, Atredis reviewed the existing test and fuzz harness coverage, reviewed the code, and added unit or fuzz test cases where coverage was missing. More detailed testing summaries have been provided for Ed25519 in the Verify Tile section and QUIC in the QUIC Tile section.

Parsers

Picohttpparser

The `picohttpparser` has been implemented to parse user input and determine if requests are valid. Fixed limits have been hardcoded into the Metrics tile including connection buffer sizes, maximum number of connections, and the maximum number of headers allowable in a single request. While headers are parsed, they are essentially ignored along with the HTTP version further limiting the attack surface. The implementation also contains mitigations such as terminating incomplete HTTP requests early to prevent against Slowloris style of Denial of Service (DoS) attacks. A fuzzing harness for `picohttpparser` already exists with a single valid request included in the fuzzing corpus. This is likely enough given the current features of the web server; the corpus may be expanded upon if the server is updated to support new HTTP verbs, endpoints, or arguments in the query string or HTTP body.

Dynamic testing was used to verify protections around DoS attacks. These tests were carried out on a test node with access to a HTTP server from a local machine. Atredis was able to cause the test server to become unresponsive in Metrics: Large Messages Cause HTTP Endpoint to Become Unresponsive and kill Firedancer completely as documented in Metrics: Excessive Connections Leads to Denial of Service.

X.509 Parser

The X.509 certificate parser has been implemented from scratch to meet RFC 5280[7] (Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile) and RFC 8410[8] (Algorithm Identifiers for Ed25519, Ed448, X25519, and X448 for Use in the Internet X.509 Public Key Infrastructure). During the assessment, Atredis reviewed the source code for memory corruption and logic issues, reviewed and executed the existing unit test and fuzz harnesses, and added unit test cases and fuzz corpora where coverage was missing. No issues were identified in the X.509 parser.

[7] https://datatracker.ietf.org/doc/html/rfc5280

[8] https://datatracker.ietf.org/doc/html/rfc8410

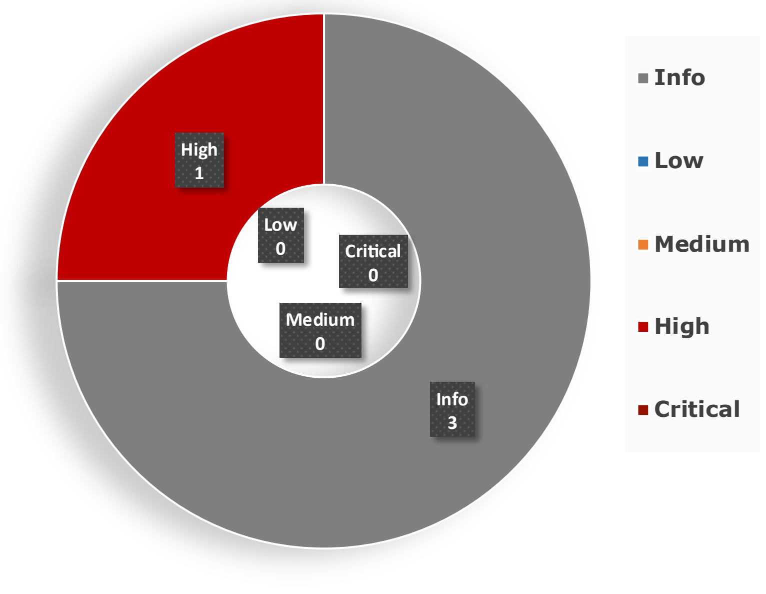
Findings Summary

In performing testing for this assessment, Atredis Partners identified **one (1) high** and **three (3) informational** findings.

Atredis defines vulnerability severity ranking as follows:

- **Critical:** These vulnerabilities expose systems and applications to immediate threat of compromise by a dedicated or opportunistic attacker.
- **High:** These vulnerabilities entail greater effort for attackers to exploit and may result in successful network compromise within a relatively short time.
- **Medium:** These vulnerabilities may not lead to network compromise but could be leveraged by attackers to attack other systems or applications components or be chained together with multiple medium findings to constitute a successful compromise.
- **Low:** These vulnerabilities are largely concerned with improper disclosure of information and should be resolved. They may provide attackers with important information that could lead to additional attack vectors or lower the level of effort necessary to exploit a system.

Findings by Severity



Findings and Recommendations

The following section outlines findings identified via manual and automated testing over the course of this engagement. Where necessary, specific artifacts to validate or replicate issues are included, as well as Atredis Partners' views on finding severity and recommended remediation.

Findings Summary

The below tables summarize the number and severity of the unique issues identified throughout the engagement.

CRITICAL	HIGH	MEDIUM	LOW	INFO
0	1	0	0	3

Findings Detail

FINDING NAME	SEVERITY
Metrics: Excessive Connections Leads to Denial of Service	High
Solana: Unvalidated Raw Pointers at FFI Boundary	Info
Net Tile: ARP Table Size Error Ignored	Info
fdctl: Segmentation Fault When Parsing Additional Command Line Arguments	Info

Metrics: Excessive Connections Leads to Denial of Service

Severity: High

Finding Overview

Pushing the maximum number of simultaneous connections made on the Metrics tile HTTP server triggers an error which forces Firedancer to exit making the node unavailable. This type of resource exhaustion may be used by attackers to implement a Denial of Service (DoS) attack on Firedancer nodes with publicly exposed metrics endpoints.

Finding Detail

The Metrics tile server registers a hard coded `MAX_CONNS` value in `fd_metric.c`.

```
#define MAX_CONNS 128
[SNIPPED]
fd_topo_run_tile_t fd_tile_metric = {
  .name                     = "metric",
  .mux_flags                = FD_MUX_FLAG_MANUAL_PUBLISH | FD_MUX_FLAG_COPY,
  .burst                    = 1UL,
  .rlimit_file_cnt          = MAX_CONNS+1,
  .mux_ctx                  = mux_ctx,
  .mux_before_credit        = before_credit,
  .populate_allowed_seccomp = populate_allowed_seccomp,
  .populate_allowed_fds     = populate_allowed_fds,
  .scratch_align            = scratch_align,
  .scratch_footprint        = scratch_footprint,
  .privileged_init          = privileged_init,
  .unprivileged_init        = unprivileged_init,
};
```

src/app/fdctl/run/tiles/fd_metric.c

The `.rlimit_file_cnt` structure member is eventually used by the sandbox initialization logic to set the maximum number of open file descriptors with a call to `setrlimit()`.

```
struct rlimit limit = { .rlim_cur = rlimit_file_cnt, .rlim_max = rlimit_file_cnt };
FD_TESTV( !setrlimit( RLIMIT_NOFILE, &limit ));
```

src/util/sandbox/fd_sandbox.c:260

Atredis was able force a large number of simultaneous connections which eventually cause a system call to fail, likely when attempting to open a file descriptor once the resource limit has been reached.

A proof-of-concept C test utility was authored and used to generate many connections while viewing the node logs.

```
./test_metric_conn 100.96.12.53 7999 256
Starting to send requests with 256 simultaneous connections
Finished sending requests.
```

Running Proof of Concept Code on Local Machine

```
ERR     05-27 23:15:13.919803 42788  f0   pidns src/app/fdctl/run/run.c(316): tile metric:0
exited with signal 31 (SIGSYS-Bad system call)
atredis@asym-research-solana:~$
```

System Error Observed in Logs

The proof-of-concept code used above has been provided to Asymmetric Research to confirm the issue across different server environments and aid in reproduction of the issue.

Recommendation(s)

Assess how the Metrics tile handles new connections when tile resource limits have been reached to ensure exceeding maximum connection is gracefully handled by the Metrics tile.

References

CWE-400: Uncontrolled Resource Consumption
https://cwe.mitre.org/data/definitions/400.html

Solana: Unvalidated Raw Pointers at FFI Boundary

Info

Finding Overview

The Firedancer team determined that this issue will not be fixed because the code on both sides of the FFI boundary is tightly controlled.

Raw pointers passed in from C code are often not checked to be non-null before dereferencing in Rust. Secure Rust guidelines recommend always validating foreign pointers prior to dereferencing to avoid undefined behavior.

Finding Detail

Firedancer does not treat FFI boundary null checks consistently. Certain Rust functions do validate foreign pointers implicitly via `as_ref` or explicitly via a null check such as the following example.

```
pub extern "C" fn fd_ext_poh_signal_leader_change( sender: *mut c_void ) {
    if sender.is_null() {
        return;
    }

    let sender: &Sender<bool> = unsafe { &*(sender as *mut Sender<bool>) };
    match sender.try_send(true) {
        Ok(()) | Err(TrySendError::Full(_)) => (),
        err => err.unwrap(),
    }
}
```

Explicit Null Check in `poh/src/firedancer_poh_recorder.rs:31`

Many other functions do not perform any validation to the raw pointer being passed in.

```
pub extern "C" fn fd_ext_poh_register_tick( bank: *const c_void, hash: *const u8 ) {
    let hash = unsafe { std::slice::from_raw_parts(hash, 32) };
    let hash = Hash::new(hash);
    unsafe { (*(bank as *const Bank)).register_tick(&hash) };
}
```

No Null Check on `bank` in `poh/src/firedancer_poh_recorder.rs:44`

Atredis noted the following areas which would benefit from a null check:

- `solana/poh/src/firedancer_poh_recorder.rs:44` - `bank, hash`
- `solana/ledger/src/blockstore.rs:260` - `blockstore`, `shred_bytes`
- `solana/ledger/src/blockstore.rs:278` - `ledger_path`, `shred_bytes`
- `solana/runtime/src/bank.rs:291` - `bank`, `address_table_lookups`
- `solana/runtime/src/bank.rs:370` - `bank`
- `solana/runtime/src/bank.rs:376` - `bank`

- `solana/runtime/src/bank.rs:382` - `load_and_execute_output`

Recommendation(s)

The Firedancer team expressed a preference for not performing checks for null values and allowing segmentation faults versus checking the value and triggering a panic. The Secure Rust guidance is to perform an explicit null check or use `as_ref` or `as_mut` whenever dereferencing foreign pointers in Rust. Whichever approach is chosen should be implemented consistently.

References

Rule `FFI-CKPTR` https://anssi-fr.github.io/rust-guide/07_ffi.html

Net Tile: ARP Table Size Error Ignored

Severity: Info

Finding Overview

The ARP table stored by the Net tile cannot grow when the system's ARP table grows. If an attacker can fill the system ARP table (with valid entries or with pending entries), then the Net tile may lose access to physical addresses needed for normal communication. This may lead to degradation of communications and dropped packets.

Finding Detail

The Firedancer system uses an instance of its Net tile to communicate over the network. For performance reasons, Firedancer uses XDP to pass its packets to the network. This method of network communication does not use the Linux kernel for the creation of packet headers and requires the Firedancer system to keep a table of local physical addresses used to craft the Ethernet headers of the packets.

To keep this table of physical addresses, Firedancer works with the Linux kernel over netlink sockets. This allows Firedancer to retrieve the system's ARP table and clone the relevant contents to its own tables. It also allows Firedancer to make an entry in the system's table when it crafts an ARP packet to try and discover a peer. These added entries allow the Linux kernel to handle the discovery of the peer's physical address, which will later be pulled down by the Net tiles.

The size of the local table that the Net tiles keep is a fixed size, set at 256. This value is used as a constant and is not configurable.

```
void *
fd_ip_new( void * shmem,
           ulong  arp_entries,
           ulong  route_entries ) {
  /*...*/
  /* use 256 as a default */
  if( arp_entries   == 0 ) arp_entries   = 256;
  /*...*/
  ip->num_arp_entries      = arp_entries;
  /*...*/
}
```

`fd_ip.c` uses 256 as the default size for the ARP table

```
  /* init fd_ip */
  ctx->ip = fd_ip_join( fd_ip_new( FD_SCRATCH_ALLOC_APPEND( l, fd_ip_align(),
fd_ip_footprint( 0UL, 0UL ) ),
                                   0UL, 0UL ) );
```

`fd_net.c:privileged_init` uses `0` as argument for arp_entries, which uses the `256` default value

The fixed size of the ARP table can become an issue when pulling from the kernel's ARP table over a netlink socket. The function `fd_ip_arp_fetch` is invoked if a valid physical address for the route is not found, and an ARP probe is sent if the fetch does not find.

`fd_nl_load_arp_table` is called which will retrieve the entries from the kernel's table. This function will exit early with an error if it runs out of room in the Firedancer ARP table.

```
while( NLMSG_OK(h, msglen) ) {
   /* are we still within the bounds of the table? */
   if( arp_entry_idx >= (long)arp_table_cap ) {
     /* we filled the table */
     FD_LOG_ERR(( "fd_nl_load_arp_table, but table larger than reserved storage" ));
     return FD_IP_ERROR; /* return failure */
   }
   /*...*/
 }
```

`fd_netlink.c:fd_nl_load_arp_table` loops over entries and exits when it runs out of space

The `fd_nl_load_arp_table` function also contains a misleading comment; while a comment states that some entries should be skipped based on their state, the state of the ARP entry does not matter. This means that entries for failed, pending, and incomplete ARP requests are still copied into the local table.

```
    /* we want to skip some entries based on state
       here are the states:
           NUD_INCOMPLETE   a currently resolving cache entry
           NUD_REACHABLE    a confirmed working cache entry
           NUD_STALE        an expired cache entry
           NUD_DELAY        an entry waiting for a timer
           NUD_PROBE        a cache entry that is currently reprobed
           NUD_FAILED       an invalid cache entry
           NUD_NOARP        a device with no destination cache
           NUD_PERMANENT    a static entry

       Keeping:
           NUD_REACHABLE    valid, so use it
           NUD_STALE        probably better to use the existing address
                              than wait or discard packet
           NUD_DELAY        an entry waiting for a timer
           NUD_PROBE        being reprobed, so it's probably valid
           NUD_PERMANENT    a static entry, so use it */
```

`fd_netlink.c:fd_nl_load_arp_table` a comment specifying which entries to keep that is not reflected in the code's logic

The function `fd_ip_arp_fetch` does not propagate the error returned, and failures due to a filled ARP table are ignored.

```
void
fd_ip_arp_fetch( fd_ip_t * ip ) {
  fd_ip_arp_entry_t * arp_table     = fd_ip_arp_table_get( ip );
  ulong               arp_table_cap = ip->num_arp_entries;
  fd_nl_t *           netlink       = fd_ip_netlink_get( ip );

  long num_entries = fd_nl_load_arp_table( netlink, arp_table, arp_table_cap );

  if( num_entries < 0L ) {
    return;
  }

  ip->cur_num_arp_entries = (ulong)num_entries;
}
```

`fd_ip.c:fd_ip_arp_fetch` does not pass on error information, and returns a `void`

Because the local ARP table has a limited size, and entries can be taken by ARP requests that are pending or failed, an attacker who can cause the table to fill with irrelevant entries may cause the local ARP table to not have access to information needed to send packets to or through valid peers. This can cause valid packets to be dropped, and can lead to degraded communications and possible denial of service.

In order to exploit this issue, attackers must be able to fill the systems ARP table with more entries than can be handled by the Net tiles. Attackers could possibly use something such as the gossip mechanism to cause the system ARP table to fill beyond what can be handled by the table kept by the Net tiles. Other services on the same machine could also expose flaws that allow an attacker to attempt to fill the ARP tables.

Filling the system's ARP table may require repeated attacks and depend on the configuration of the host. The ARP table will be cleared by the kernel as entries time out and requests fail.

Recommendation(s)

The Firedancer system should better handle the case where the host's ARP table is filled. The Firedancer system should discard entries that are not relevant, such as ones with state `NUD_FAILED` or `NUD_NONE`. Adding a configuration parameter to expand the Net tile's ARP tables from the default `256` may also help administrators deal with local ARP tables that are not sufficiently big.

References

CWE-252: Unchecked Return Value https://cwe.mitre.org/data/definitions/252.html

fdctl: Segmentation Fault When Parsing Additional Command Line Arguments

Severity: Info

Finding Overview

When executing `fdctl` with additional arguments set for the `configure` subcommand, the argument parsing logic will attempt to access an out-of-bounds memory location which leads to segmentation fault.

This issue is not exploitable to a remote attacker and is being reported as an informational issue to improve code quality and user experience.

Finding Detail

Invoking `fdctl` with multiple arguments will cause a segmentation fault as demonstrated below.

```
$ sudo fdctl configure init d d
Log at "/tmp/fd-0.0.0_125061_firedancer_ubuntu-m-4vcpu-32gb-intel-nyc3-
01_2024_05_14_19_16_43_981543891_GMT+00"
Segmentation fault
```

Segmentation Fault When Setting Additional Arguments

```
(gdb)set args configure init d d
(gdb) p i
$15 = 0
(gdb) p *pargv[ i ]
$16 = 0x7fffffffe736 "d"
(gdb) n
23      if( FD_UNLIKELY( !strcmp( *pargv[ i ], "all" ) ) ) {
(gdb) p i
$17 = 1
(gdb) p *pargv[ i ]
Cannot access memory at address 0x0
(gdb) n

Program received signal SIGSEGV, Segmentation fault.
0x00005555594d8aab in configure_cmd_args (pargc=0x7fffffffbf7c, pargv=0x7fffffffbf88,
args=0x7fffffffc1e0) at src/app/fdctl/configure/configure.c:23
23      if( FD_UNLIKELY( !strcmp( *pargv[ i ], "all" ) ) ) {
```

`gdb` Session Indicating Out-of-Bounds Access on `pargv`

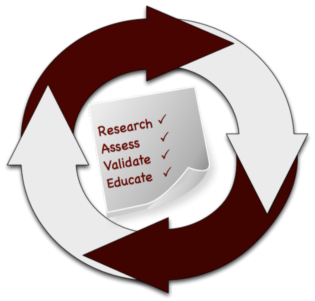
Recommendation(s)

Review the affected argument parsing logic in `configure.c` and ensure additional arguments are safely ignored.

References

CWE-125: Out-of-bounds Read https://cwe.mitre.org/data/definitions/125.html

Appendix I: Assessment Methodology



Atredis Partners draws on our extensive experience in penetration testing, reverse engineering, hardware/software exploitation, and embedded systems design to tailor each assessment to the specific targets, attacker profile, and threat scenarios relevant to our client's business drivers and agreed upon rules of engagement.

Where applicable, we also draw on and reference specific industry best practices, regulations, and principles of sound systems and software design to help our clients improve their products while simultaneously making them more stable and secure.

Our team takes guidance from industry-wide standards and practices such as the National Institute of Standards and Technology's (NIST) Special Publications, the Open Web Application Security Project (OWASP), and the Center for Internet Security (CIS).

Throughout the engagement, we communicate findings as they are identified and validated, and schedule ongoing engagement meetings and touchpoints, keeping our process open and transparent and working closely with our clients to focus testing efforts where they provide the most value.

In most engagements, our primary focus is on creating purpose-built test suites and toolchains to evaluate the target, but we do utilize off-the-shelf tools where applicable as well, both for general patch audit and best practice validation as well as to ensure a comprehensive and consistent baseline is obtained.

Research and Profiling Phase

Our research-driven approach to testing begins with a detailed examination of the target, where we model the behavior of the application, network, and software components in their default state. We map out hosts and network services, patch levels, and application versions. We frequently use a number of private and public data sources to collect Open Source Intelligence about the target, and collaborate with client personnel to further inform our testing objectives.

For network and web application assessments, we perform network and host discovery as well as map out all available application interfaces and inputs. For hardware assessments, we study the design and implementation, down to a circuit-debugging level. In reviewing source code or compiled application code, we map out application flow and call trees and develop a solid working understand of how the application behaves, thus helping focus our validation and testing efforts on areas where vulnerabilities might have the highest impact to the application's security or integrity.

Analysis and Instrumentation Phase

Once we have developed a thorough understanding of the target, we use a number of specialized and custom-developed tools to perform vulnerability discovery as well as binary, protocol, and runtime analysis, frequently creating engagement-specific software tools which we share with our clients at the close of any engagement.

We identify and implement means to monitor and instrument the behavior of the target, utilizing debugging, decompilation and runtime analysis, as well as making use of memory and filesystem forensics analysis to create a comprehensive attack modeling testbed. Where they exist, we also use common off-the-shelf, open-source and any extant vendor-proprietary tools to aid in testing and evaluation.

Validation and Attack Phase

Using our understanding of the target, our team creates a series of highly-specific attack and fault injection test cases and scenarios. Our selection of test cases and testing viewpoints are based on our understanding of which approaches are most relevant to the target and will gain results in the most efficient manner, and built in collaboration with our client during the engagement.

Once our test cases are validated and specific attacks are confirmed, we create proof-of-concept artifacts and pursue confirmed attacks to identify extent of potential damage, risk to the environment, and reliability of each attack scenario. We also gather all the necessary data to confirm vulnerabilities identified and work to identify and document specific root causes and all relevant instances in software, hardware, or firmware where a given issue exists.

Education and Evidentiary Phase

At the conclusion of active testing, our team gathers all raw data, relevant custom toolchains, and applicable testing artifacts, parses and normalizes these results, and presents an initial findings brief to our clients, so that remediation can begin while a more formal document is created. Additionally, our team shares confirmed high-risk findings throughout the engagement so that our clients may begin to address any critical issues as soon as they are identified.

After the outbrief and initial findings review, we develop a detailed research deliverable report that provides not only our findings and recommendations but also an open and transparent narrative about our testing process, observations and specific challenges in developing attacks against our targets, from the real world perspective of a skilled, motivated attacker.

Automation and Off-The-Shelf Tools

Where applicable or useful, our team does utilize licensed and open-source software to aid us throughout the evaluation process. These tools and their output are considered secondary to manual human analysis, but nonetheless provide a valuable secondary source of data, after careful validation and reduction of false positives.

For runtime analysis and debugging, we rely extensively on off-the-shelf and platform-specific runtime debuggers, and develop fuzzing, memory analysis, and other testing tools as needed.

In source auditing, we typically work in IDEs, text editors, and other markup tools. For automated source code analysis, we will use the most appropriate toolchain for the target, unless client preference dictates another tool.

Engagement Deliverables

Atredis Partners deliverables include a detailed overview of testing steps and testing dates, as well as our understanding of the specific risk profile developed from performing the objectives of the given engagement.

In the engagement summary we focus on “big picture” recommendations and a high-level overview of shared attributes of vulnerabilities identified and organizational-level recommendations that might address these findings.

In the findings section of the document, we provide detailed information about vulnerabilities identified, provide relevant steps and proof-of-concept code to replicate these findings, and our recommended approach to remediate the issues, developing these recommendations collaboratively with our clients before finalization of the document.

Our team typically makes use of both DREAD and NIST CVE for risk scoring and naming, but as part of our charter as a client-driven and collaborative consultancy, we can vary our scoring model to a given client’s preferred risk model, and in many cases will create our findings using the client’s internal findings templates, if requested.

Sample deliverables can be provided upon request, but due to the highly specific and confidential nature of Atredis Partners’ work, these deliverables will be heavily sanitized, and give only a very general sense of the document structure.

Appendix II: Engagement Team Biographies

Jordan Whitehead, Research Consulting Director

Jordan Whitehead specializes in vulnerability research and binary exploitation. Jordan is able to quickly dive into large systems and find key weaknesses as a result of his significant experience in operating system internals.

Experience

During Jordan’s Computer Engineering degree schooling, he created and instructed collegiate courses and clubs on computer security. After college he worked as a CNO developer for ManTech International, developing tools and capabilities that involved deep exploration into modern operating systems for exploitable weaknesses. While in that position, Jordan also continued to help create and instruct a number of formal reverse engineering and exploitation courses. These courses detailed the system internals for Windows, Linux, and Android. He has worked with research teams developing custom virtualization and emulation tooling that enabled researchers to better assess otherwise unreachable systems.

Key Accomplishments

Jordan has helped publish papers at top academic conferences on computer security, including Usenix Security Symposium. He has also developed open-source tools related to vulnerability research and secure software. These include peer-reviewed tools that have helped provide useable security and trust on Linux and Windows platforms.

Bryan C. Geraghty, Principal Research Consultant

Bryan leads and executes highly technical application and network security assessments, as well as adversarial simulation assessments. He specializes in cryptography and reverse engineering.

Experience

Bryan has over 20 years of experience building and exploiting networks, software, and hardware systems. His deep background in systems administration, software development, and cryptography has been demonstrably beneficial for security assessments of custom or unique applications in industries such as healthcare, manufacturing, marketing, banking, utilities, and entertainment.

Key Accomplishments

Bryan is a creator and maintainer of several open-source security tools. He is also a nationally recognized speaker; often presenting research on topics such as software, hardware, and communications protocol attacks, and participating in offense-oriented panel discussions. Bryan is also an organizing-board member of multiple Kansas City security events, and a staff volunteer & organizer of official events at DEF CON.

Loren Browman, Senior Research Consultant

Loren Browman has over 10 years of experience in both consulting and federal law enforcement environments. His experiences range from deep security research in federal government to product and application testing for Fortune 500 corporations. Loren is a recognized subject matter expert (SME) in securing IoT products and advanced hardware testing methodology. Areas of expertise include reverse engineering of hardware, firmware, and communication protocols.

Experience

Loren has conducted numerous large scale product security assessments including challenging black box security assessments and secure design reviews.

Prior to joining Atredis, Loren was an operations supervisor and security researcher for the Royal Canadian Mounted Police (RCMP). This role included providing technical expertise to support police investigations and leading security research efforts in order to circumvent security mechanisms and develop deployable capabilities.

Key Accomplishments

Loren has developed numerous tools for accelerating research on a wide range of products. This includes the development of a fuzzing suite for automotive Electronic Control Units over CAN bus vehicle networks, this led to the discovery of multiple hidden services and exploits. More recently, Loren published nrfsec, a tool for automating firmware recovery vulnerability on secured nrf51 System on Chips.

Loren has studied Electrical and Computer Engineering at the British Columbia Institute of Technology and has attended various specialized training sessions including the Arm IoT Exploit Laboratory, Power Analysis and Glitching and is an Offensive Security Certified Professional (OSCP).

Appendix III: About Atredis Partners

Atredis Partners was created in 2013 by a team of security industry veterans who wanted to prioritize offering quality and client needs over the pressure to grow rapidly at the expense of delivery and execution. We wanted to build something better, for the long haul.

In six years, Atredis Partners has doubled in size annually, and has been named three times to the Saint Louis Business Journal's "Fifty Fastest Growing Companies" and "Ten Fastest Growing Tech Companies". Consecutively for the past three years, Atredis Partners has been listed on the Inc. 5,000 list of fastest growing private companies in the United States.

The Atredis team is made up of some of the greatest minds in Information Security research and penetration testing, and we've built our business on a reputation for delivering deeper, more advanced assessments than any other firm in our industry.

Atredis Partners team members have presented research over forty times at the BlackHat Briefings conference in Europe, Japan, and the United States, as well as many other notable security conferences, including RSA, ShmooCon, DerbyCon, BSides, and PacSec/CanSec. Most of our team hold one or more advanced degrees in Computer Science or engineering, as well as many other industry certifications and designations. Atredis team members have authored several books, including *The Android Hacker's Handbook*, *The iOS Hacker's Handbook*, *Wicked Cool Shell Scripts*, *Gray Hat C#*, and *Black Hat Go*.

While our client base is by definition confidential and we often operate under strict nondisclosure agreements, Atredis Partners has delivered notable public security research on improving the security at Google, Microsoft, The Linux Foundation, Motorola, Samsung and HTC products, and were the first security research firm to be named in Qualcomm's Product Security Hall of Fame. We've received four research grants from the Defense Advanced Research Project Agency (DARPA), participated in research for the CNCF (Cloud Native Computing Foundation) to advance the security of Kubernetes, worked with OSTIF (The Open Source Technology Improvement Fund) and The Linux Foundation on the Core Infrastructure Initiative to improve the security and safety of the Linux Kernel, and have identified entirely new classes of vulnerabilities in hardware, software, and the infrastructure of the World Wide Web.

In 2015, we expanded our services portfolio to include a wide range of advanced risk and security program management consulting, expanding our services reach to extend from the technical trenches into the boardroom. The Atredis Risk and Advisory team has extensive experience building mature security programs, performing risk and readiness assessments, and serving as trusted partners to our clients to ensure the right people are making informed decisions about risk and risk management.

Inc. 5000 AMERICA'S FASTEST GROWING PRIVATE COMPANIES

FAST 50