Security Audit Report

COMMUNE AI-SUBSPACE

Substrate Pallet

by xfactor.toml@gmail.com

Jan 17, 2024

Table of Content

Audit Summary

Scope of initial review

- Repository: https://github.com/commune-ai/subspace
- Commit: `cc51b72b33a178af699e0cd37222a374fbf30b2e`
- Pallet: `./pallets/subspace`
- Node: `./node`
- Runtime: `./runtime`

Report objectives

1. Report all issues in **pallets** alongside recommendations
2. Report all issues in **node** alongside recommendations
3. Report all issues in **runtime** alongside recommendations
4. Report all issues in **test** alongside recommendations
5. Report all **other** issues alongside recommendations

Audit Methodology

- Tooling Analysis
- Manual Review

Issues

Number of issues reported in the initial review and remaining in the final review:

Severity	Reported			Remaining		
	Code	Test	Other	Code	Test	Other
Critical	1	0	0	-	-	-
Major	2	1	0	-	-	-

Medium	1	1	0	-	-	-
Minor	0	1	0	-	-	-

Disclaimer

The security audit report makes no statements or warranties, either expressed or implied, regarding the security of the code, the information herein or its usage. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bug free status or any other statements of the contract.

This report does not constitute legal or investment advice. It is for informational purposes only and is provided on an "as-is" basis. You acknowledge that any use of this report and the information contained herein is at your own risk. The authors of this report shall not be liable to you or any third parties for any acts or omissions undertaken by you or any third parties based on the information contained herein.

General Verification

Introduction

Subspace is a Substrate-based chain that were not developed by Parity Technologies. A general verification has been performed, utilizing Tooling Analysis to check if the codebase contains malicious or unverified libraries.

Methodology

Tooling analysis for General Verification

- cargo-audit (https://rustsec.org)

  The RustSec Advisory Database is a repository of security advisories filed against Rust crates published via crates.io maintained by the Rust Secure Code Working Group.

- Static analyzers (https://github.com/rust-lang/rust-analyzer)

  One of the examples of static analyzers is rust-analyzer. It is a modular compiler frontend for the Rust language. It is a part of a larger rls-2.0 effort to create excellent IDE support for Rust.

- Testing (cargo test)

  Writing tests and checking them will help you to be sure that all scripts available for the system will work correctly. Integration tests between pallets are particularly important.

Findings

1. Unresolved Merge Conflict on voting.rs (Major)
   Location: `pallets/subspace/src/voting.rs:20:1`
   Approach: `cargo build -release`
   Outcome:

```
    Compiling pallet-utility v4.0.0-dev (https://github.com/paritytech/substrate?branch=polkadot-v1.0.0#948fbd2f)
    Compiling pallet-evm-chain-id v1.0.0-dev (https://github.com/paritytech/frontier?branch=polkadot-v1.0.0#301d7c8f)
    Compiling pallet-transaction-payment-rpc v4.0.0-dev (https://github.com/paritytech/substrate?branch=polkadot-v1.0.0#948fbd2f)
error: encountered diff marker
  --> pallets/subspace/src/voting.rs:20:1
   |
20 | <<<<<<< HEAD
   | ------- after this is the code before the merge
...
30 | =======
   | -------
70 | >>>>>>> 8c7ebe5037e266c1395b42708c95cfb38b9e6b0
   | ------- above this are the incoming code changes

   = help: if you're having merge conflicts after pulling new code, the top section is the code you already had and the bottom section is the remote code
   = help: if you're in the middle of a rebase, the top section is the code being rebased onto and the bottom section is the code coming from the current commit being rebased
   = note: for an explanation on these markers from the `git` documentation, visit <https://git-scm.com/book/en/v2/Git-Tools-Advanced-Merging#_checking_out_conflicts>

error: could not compile `pallet-subspace` (lib) due to 1 previous error
```

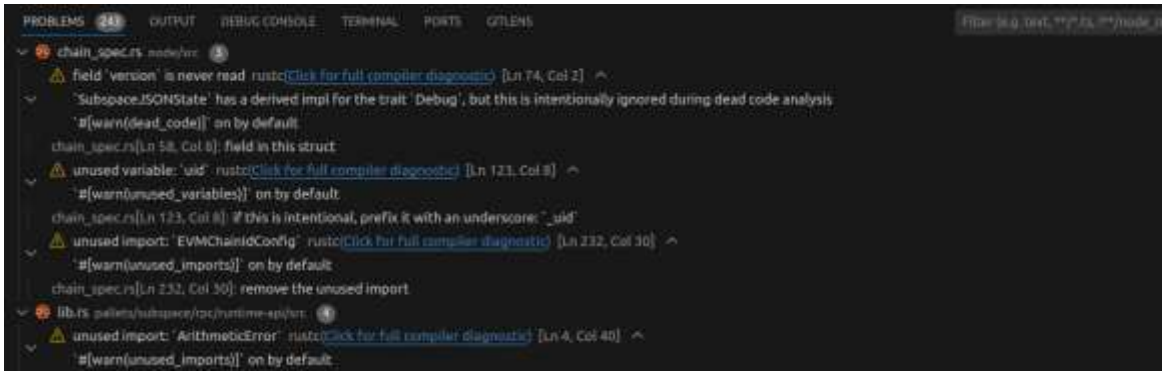
   Recommendation: Remove line:20~line:70 to resolve conflict

2. Vulnerable Crate in dependency tree (Medium)
   Approach:
   `cargo install cargo-audit`
   `cargo audit`
   Outcome:
   https://rustsec.org/advisories/RUSTSEC-2022-0093

```
      Fetching advisory database from `https://github.com/RustSec/advisory-db.git`
        Loaded 588 security advisories (from /home/ubuntu/.cargo/advisory-db)
      Updating crates.io index
      Scanning Cargo.lock for vulnerabilities (866 crate dependencies)
Crate:     ed25519-dalek
Version:   1.0.1
Title:     Double Public Key Signing Function Oracle Attack on `ed25519-dalek`
Date:      2022-06-11
ID:        RUSTSEC-2022-0093
URL:       https://rustsec.org/advisories/RUSTSEC-2022-0093
Solution:  Upgrade to >=2
Dependency tree:
ed25519-dalek 1.0.1
└── sp-io 23.0.0
    ├── try-runtime-cli 0.10.0-dev
    │   └── node-subspace 4.0.0-dev
```

   Recommendation: Upgrade to >=2

3. Rust language server recommended warnings (Minor)
   Approach: Check with Rust-Analyzer
   Outcome: 243 warnings



Recommendation:

- Remove unused variables
- Remove unused imports
- Remove unnecessary braces
- Resolve unused `Result` that must be used

Summary

- Codebase is not following the rust programming convention. It makes the codebase difficult to understand and maintain. It also leads too many warnings and unnecessary memory usage on runtime.
- The dependencies are outdated. The latest polkadot-sdk version is `release-polkadot-v1.6.0` but subspace is using `release-polkadot-v1.0.0`. Parity Technologies is continuously resolve known issues and update polkadot-sdk.

Code Verification

Introduction

Code Verification analyzes the logic and algorithms implemented in the codebase. The goal is to ensure that the logic is sound, efficient, and aligned with the desired outcomes. It reviews line-by-line of the code to identify bugs, logic errors, or any unintended behavior.

Methodology

- Understanding the business logic
- Manual review line by line

Findings

4. Insecure Randomness (Major)
   Location:
   `./pallets/subspace/src/registration.rs:224`

```
registration.rs 2 ×
pallets > subspace > src > registration.rs > {} impl Pallet<T> > add_subnet_from_registration
    // Returns a random index in range 0..n.
    pub fn random_idx(n: u16) -> u16 {
        let block_number: u64 = Self::get_current_block_as_u64();
        // take the modulos of the blocknumber
        let idx: u16 = ((block_number % u16::MAX as u64) % (n as u64)) as u16;
        return idx
    }
```

Vulnerability:

Weak random numbers can allow attackers to guess or change the numbers to trick the system.

Recommendation:

Choose a strong method for generating random numbers.

- Find a custom trusted oracle.
- Otherwise, use a method VRF, which Polkadot uses in processes like auctions.

5. Bad Extrinsics' Weights (Critical)

Location:

`./pallets/subspace/src/lib.rs`

Vulnerability: 18 Zero Weights are found.

If the weights are set incorrectly, then spam transactions can cause DoS of a node.

```
lib.rs ×
pallets > subspace > src > lib.rs > {} pallet > {} impl Pallet<T>
    #[pallet::call]
    impl<T: Config> Pallet<T> {
        #[pallet::weight((Weight::zero(), DispatchClass::Normal, Pays::No))]
        pub fn set_weights(
            origin: OriginFor<T>,
            netuid: u16,
            uids: Vec<u16>,
            weights: Vec<u16>,
        ) -> DispatchResult {
            Self::do_set_weights(origin, netuid, uids, weights)
        }
```

Recommended:

Add missing benchmarking and use autogen_weights.rs

6. Insufficient Benchmarking (Major)

Location:

`./pallets/subspace/src/benchmarking.rs`

Vulnerability:

Insufficient Benchmarking for batch extrinsic calls.

benchmarking.rs

pallets > subspace > src > benchmarking.rs > {} benchmarks > transfer_stake

```rust
        Ok(())
    }

    #[benchmark]
    fn add_stake_multiple() -> Result<(), BenchmarkError> {
        let caller: T::AccountId = account("caller", 0, SEED);

        let network: Vec<u8> = b"network".to_vec();
        let address: Vec<u8> = b"address".to_vec();

        let module_keys: Vec<T::AccountId> = (0..10).map(|i| account("key", i, SEED)).collect();
        let amounts: Vec<u64> = (0..10).map(|i| i + MIN_STAKE).collect();

        let mut netuid: u16 = 0;
```

Incorrect or missing benchmarking can slow down the network and it can also let attackers spam the system.

Recommendation:

Run benchmarks using the worst-case scenario conditions. An example is that the benchmark should cover the execution path where more DB reads and writes happen in an extrinsic. The primary goal is to keep the runtime safe, and the secondary goal is to be as accurate as possible to maximize throughput.

7. Unsafe Arithmetic in balance calculation (Medium)

Vulnerability:

Unsafe maths operations can lead to wrong calculation results due to overflows/underflows. This might open a door for attackers to trick the system and cause serious inconsistencies.

Recommendation:

Use safe maths functions that check for errors like `checked_add` or `checked_sub`, proofread the code for unsafe maths and fix them.