OtterSec

Fuel Core

Security Assessment

October 18th, 2024 — Prepared by OtterSec

Alpha Toure shxdow@osec.io

James Wang james.wang@osec.io

Table of Contents

01 — Executive Summary

Overview

Fuel Labs engaged OtterSec to assess the `fuel-core` and auxiliary updates. This assessment was conducted between August 5th and October 5th, 2024. For more information on our auditing methodology, refer to chapter 07.

Key Findings

We produced 6 findings throughout this audit engagement.

In particular, we discovered attacks against `txpool` which may bring down validators due to improper transaction dependency handling (OS-FCR-ADV-00), and improper utilization of speculative outputs which allows attackers to degrade chain performance by flushing `txpool` transactions at near zero cost (OS-FCR-ADV-02). We also disclosed incorrect usages of `IS` register to locate contract address within the VM which may result in breakage of `blob` usage within `proxy` contracts (OS-FCR-ADV-01).

We also made recommendations around a race condition in the new `txpool` implementation that allows attackers to squeeze out valid transactions (OS-FCR-SUG-00).

02 — Scope

The source code was delivered to us in a git repository at https://github.com/FuelLabs/fuel-core. This audit was performed against commit 38cbc38, with focus on executor, upgradeable-executor, importer, storage, consensus, relayer. Auxiliary updates include contract proxy loader, and blob support for scripts and predicates in 90edebd, 0adb5f8 and 2a67e3b.

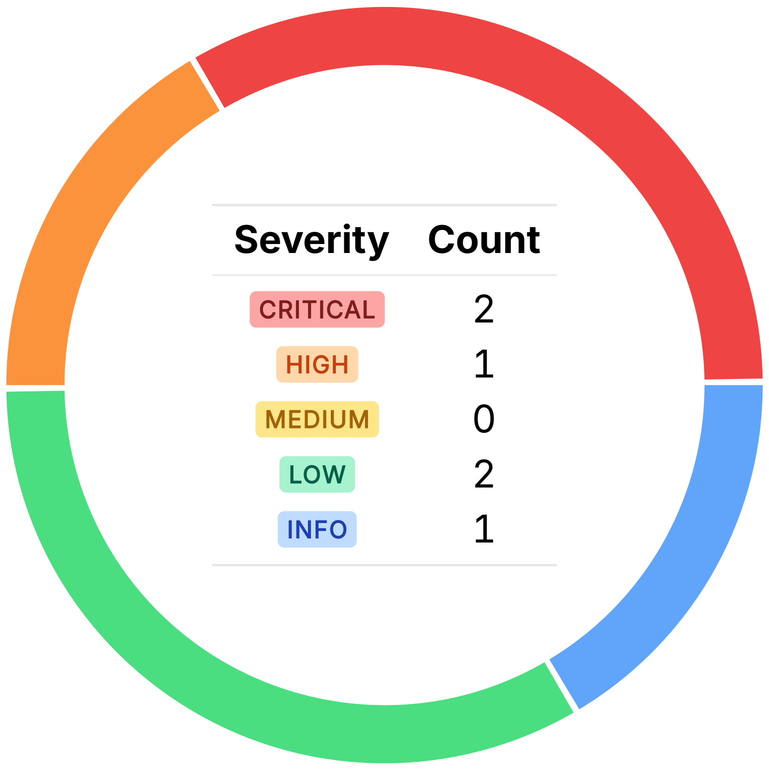
A brief description of the programs is as follows:

Name	Description
executor	The native transaction executor responsible for executing transactions within a block.
upgradeable-executor	A wrapper around executor with additional supports for `wasm` transition function to allow forkless upgrade and execution of historical blocks.
importer	A service for importing new blocks into local storage.
storage	The module responsible for managing and persisting chain storage.
consensus	Implements `PoA` consensus algorithm to handle signing and creation of new blocks.
relayer	Listens for L1 transactions and relay deposit and forced transaction events to the fuel chain.
proxy loader	Implements a loader contract in the sdk to facilitate simple proxy contract deployment.
predicate and script blob	Allows storage of predicate and script as blobs on chain to reduce tx size.

03 — Findings

Overall, we reported 6 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will aid in mitigating future vulnerabilities.



Severity	Count
CRITICAL	2
HIGH	1
MEDIUM	0
LOW	2
INFO	1

04 — Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in chapter 06.

ID	Severity	Status	Description
OS-FCR-ADV-00	CRITICAL	RESOLVED ✓	In transaction pool, dependency checks during insertion may panic due to existence of invalid UTXOs in the pool due to prior removal of transactions without proper re-validation.
OS-FCR-ADV-01	CRITICAL	RESOLVED ✓	`loader` incorrectly utilizes the `IS` register for loading blob contents, which results in errors when executed by a proxy.
OS-FCR-ADV-02	HIGH	RESOLVED ✓	The `txpool` may be flooded with dependent transactions, allowing them to evict a crucial initial transaction with a high `ratio_gas_tip`, potentially clearing the pool at minimal cost.
OS-FCR-ADV-03	LOW	RESOLVED ✓	It is possible to exceed the 65,535 transaction limit in a fuel block, which will result in chain halting if the gas limit parameter is not set appropriately.
OS-FCR-ADV-04	LOW	RESOLVED ✓	`load_memory_code` copies the entire padded length instead of the actual `length_unpadded`, resulting in copying excess data.

Panic due to Removal of Transaction Dependencies CRITICAL OS-FCR-ADV-00

Description

Within `dependency::insert`, it is possible for panics to occur due to the incorrect handling of dependencies within the transaction pool (`TxPool`) when transactions are removed and new ones are inserted (highlighted below). Specifically, the issue arises due to changes in the pool's state during the insertion process, without re-validating the new state before completing the transaction's insertion. This may result in a scenario where invalid UTXOs (unspent transaction outputs) are retained in the dependency graph, which will result in a panic during subsequent operations.

>_ *fuel-core/crates/services/txpool/src/containers /dependency.rs* RUST

```rust
pub(crate) fn insert<'a, DB>(
    &'a mut self,
    txs: &'a HashMap<TxId, TxInfo>,
    db: &DB,
    tx: &'a ArcPoolTx,
) -> Result<Vec<ArcPoolTx>, Error>
where
    DB: TxPoolDb,
{
    let (max_depth, db_coins, db_contracts, db_messages, collided) =
        self.check_for_collision(txs, db, tx)?;

    // now we are sure that transaction can be included. remove all collided transactions
    let mut removed_tx = Vec::new();
    for collided in collided.into_iter() {
        let collided = txs
            .get(&collided)
            .expect("Collided should be present in txpool");
        removed_tx.extend(
            self.recursively_remove_all_dependencies(txs, collided.tx().clone()),
        );
    }
    [...]
}
```

More specifically, when a new transaction is inserted into the pool, the pool checks for potential collisions with existing transactions. If a collision is found, conflicting transactions may be removed to make space for the new one, but the system does not re-validate the new transactions after removing collisions. If a transaction that generates a UTXO is removed (due to a collision), but the pool still considers the UTXO valid, it results in an invalid UTXO. When future transactions try to reference this UTXO, the system will fail to locate the transaction that generated it, resulting in a panic.

For example, such a panic may occur in the `unwrap` call in `check_for_collision`, where the system tries to look up the transaction that generated a UTXO. If `utxo_id.tx_id()` refers to a removed transaction, the call to `unwrap` will panic.

Proof of Concept

The following is a step-by-step illustration of the above vulnerability:

1. `txA` does not take any inputs (aside from the gas fees) and generates an output `{coinA}`. `{coinA}` is a UTXO created by `txA`.
2. `txB` takes `{coinA}` as input and generates a new output `{coinB}`.
3. `txC` takes both `{coinA}` and `{coinB}` as inputs. Since `{coinA}` is shared between `txB` and `txC`, this creates a collision between `txB` and `txC`.
4. If `txC` has a higher `ratio_gas_tip` (a greater incentive for validators), it will be considered more favorable than `txB`, and `txB` will be removed from the pool.
5. When `txB` is removed, its output `{coinB}` is also effectively invalidated because the transaction that generated it (`txB`) no longer exists in the pool. However, `txC` remains in the pool and still references both `{coinA}` and `{coinB}`.
6. Now `txD` attempts to utilize `{coinB}` as an input. When `txD` is processed, the system tries to look up the transaction that generated `{coinB}` (i.e., `txB`), but since `txB` has been removed, the lookup fails, and the system panics.

Remediation

Redesign the `txpool` transaction dependency handling algorithm to properly handle collisions.

Patch

`Txpool` is redesigned in 6111cef, fdcec1c and 328b42c.

Incorrect Base Register Utilization CRITICAL OS-FCR-ADV-01

Description

The vulnerability pertains to the utilization of the `IS` (Instruction Start) register inappropriately as a base address for loading the blob contents. `loader` moves the contents of the `IS` register into register `0x10`. The purpose of this is to locate the address of blob IDs, as they are positioned after the contract's code in memory.

>_ *fuels-rs/packages/fuels-programs/src/contract /loader.rs* RUST

```rust
pub fn loader_contract_asm(blob_ids: &[BlobId]) -> Result<Vec<u8>> {
    const BLOB_ID_SIZE: u16 = 32;
    let get_instructions = |num_of_instructions, num_of_blobs| {
        // There are 2 main steps:
        // 1. Load the blob contents into memory
        // 2. Jump to the beginning of the memory where the blobs were loaded
        // After that the execution continues normally with the loaded contract reading our
        // prepared fn selector and jumps to the selected contract method.
        [
            // 1. Load the blob contents into memory
            // Find the start of the hardcoded blob IDs, which are located after the code ends.
            op::move_(0x10, RegId::IS),
            [...]
        ]
    };
    [...]
}
```

However, `loader` may be loaded by a proxy contract. In this case, the `IS` register does not reflect the start of the loader itself but the start of the proxy contract. Since `loader` begins after the proxy contract's code, `IS` points to the start of the proxy, not the loader. This results in an incorrect base address for loading blob contents, resulting in potential errors when trying to load or execute the blobs.

Remediation

Utilize `PC` instead of `IS`.

Patch

Resolved in fc6eb56.

Costless Flushing of Txpool via Speculative Inputs HIGH OS-FCR-ADV-02

Description

There is a potential Denial of Service (DoS) attack on the transaction pool. It exploits the mechanics of how transactions are processed and prioritized within the pool. by Specifically, by submitting a transaction with multiple `Output::Variable` outputs, which are not fully validated until execution, which allow subsequent transactions to utilize them as inputs with arbitrary values, enabling the attacker to generate a large number of transactions with inflated `ratio_gas_tip`, without any risk of executing them.

Consequently, they may submit a new transaction that forces the `txpool` to evict the initial one resulting in a complete flush of the `txpool` at zero cost.

Proof of Concept

Below is an illustration of the attack:

1. An attacker sends a transaction (`Tx1`) that includes `Output::Variable` outputs, with a relatively high `ratio_gas_tip` to ensure that it evicts other transactions in the pool.
2. The attacker then sends multiple transactions that depend on the outputs of `Tx1`. These dependent transactions (let's call them `Tx2`, `Tx3`, etc.) can have arbitrary amounts for the `Input::Coin` since the validity checks for the `Output::Variable` are postponed. As a result, the attacker can artificially inflate the `ratio_gas_tip` of these transactions without incurring any actual cost since the checks will only be applied at the time of execution.
3. The attacker sends a final transaction with a high `ratio_gas_tip` that depends on the output of the first transaction. Due to the elevated gas tip, the first transaction is pushed out of the transaction pool, and since the final transaction depends on the first one's output, it is also removed.
4. If the attack fully succeeds, the attacker can flush the transaction pool at no cost.
5. If the first transaction is executed before the attacker fills the pool or the `ratio_gas_tip` is insufficient to evict all other transactions, the attacker can still flush a significant portion of the transaction pool, incurring only the cost of the first transaction.

Remediation

Disallow usage of speculative inputs to raise the cost of flooding `txpool`.

Patch

`Txpool` is redesigned in 6111cef, fdcec1c and 328b42c.

DOS due to Exceeding Transaction Limit LOW

OS-FCR-ADV-03

Description

In the Fuel blockchain, a block is limited to 65,536 transactions (due to the utilization of `u16` as `tx.count`). Thus, the invariant here is that the sum of L1 and L2 transaction counts must never exceed 65,535. If this invariant is violated, the mint transaction (`mintTx`) would fail in `produce_mint_tx` (shown below) due to insufficient space in the block, resulting in a chain halt.

>_ *fuel-core/crates/services/executor/src/executor.rs* RUST

```rust
fn produce_mint_tx<TxSource, T>(
    &self,
    block: &mut PartialFuelBlock,
    components: &Components<TxSource>,
    storage_tx: &mut BlockStorageTransaction<T>,
    data: &mut ExecutionData,
    memory: &mut MemoryInstance,
) -> ExecutorResult<()>
where
    T: KeyValueInspect<Column = Column>,
{
    [...]
    self.execute_transaction_and_commit(
        block,
        storage_tx,
        data,
        MaybeCheckedTransaction::Transaction(coinbase_tx.into()),
        gas_price,
        coinbase_contract_id,
        memory,
    )?;
    Ok(())
}
```

However, this invariant is brittle because it relies on the gas limit which is upgradable. The gas limit in deciding the amount of L1 and L2 transactions fetched must be tight enough to guarantee total transactions never exceed 65535. Currently, the `MIN_GAS_PER_TX` for Forced Transaction Inclusions (FTIs) is set to one, implying theoretically, the system will allow for more than 65,535 transactions from L1 alone.

>_ *fuel-core/crates/services/executor/src/executor.rs* RUST

```rust
fn process_da<D>(
    &mut self,
    block_height: BlockHeight,
    da_block_height: DaBlockHeight,
    execution_data: &mut ExecutionData,
    block_storage_tx: &mut BlockStorageTransaction<D>,
    memory: &mut MemoryInstance,
) -> ExecutorResult<Vec<CheckedTransaction>>
where
    D: KeyValueInspect<Column = Column>,
{
    [...]
        Event::Transaction(relayed_tx) => {
           let id = relayed_tx.id();
           let checked_tx_res = Self::validate_forced_tx(
               relayed_tx,
               block_height,
               &self.consensus_params,
               memory,
           );[...]
        }
    [...]
}
```

`validate_forced_tx` (shown above) is critical as it mitigates this risk by raising the bar on the gas required for each FTI. Nevertheless, it is still relying on the block gas limit parameter / transaction gas metering to never get upgraded to a value unfortunate enough to allow 65536 transactions to pass the check. Therefore, the current mechanism does not provide a hard limit on the number of transactions per block, thus forkless upgrades to raise per-block gas limit may still render the chain vulnerable.

Given the identified risks and brittleness of the current system due to reliance on mutable parameters, implementing an additional check on transaction counts is imperative to serve as a safeguard.

Remediation

Explicitly add a check to enforce the transaction count as a limit of the number of transactions selected to reinforce the integrity of the block production process.

Patch

Resolved in 79ca0d0 and ca6a705.

Improper Data Loading for Mode 2 ldc LOW OS-FCR-ADV-04

Description

In the current implementation, `blockchain::load_memory_code` copies data utilizing the padded length (`length`) rather than the unpadded length (`length_unpadded`). As a result, the function may be copying more bytes than requested. In Mode 2, only the unpadded length (`length_unpadded`) should be copied from the source address (`src`), and the remaining space (`length` - `length_unpadded`) should be explicitly filled with null bytes (zeroes).

>_ *fuel-vm/fuel-vm/src/interpreter/blockchain.rs* RUST

```rust
pub(crate) fn load_memory_code(
    mut self,
    input_src_addr: Word,
    input_offset: Word,
    length_unpadded: Word,
) -> IoResult<(), S::DataError>
where
    S: InterpreterStorage,
{
    [...]
    // Copy the code.
    self.memory.memcopy(dst, src, length, owner)?;
    [...]
}
```

Remediation

Ensure to copy only the unpadded length and fill any remaining space in the padded region with null bytes.

Patch

Resolved in de235f8.

05 — General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may result in security issues in the future.

ID	Description
OS-FCR-SUG-00	A race condition in the `txpool`, allows inserting invalid transactions by exploiting delayed UTXO validation, potentially enabling a zero-cost transaction pool flush.

Degrading Chain Performance via Race Condition OS-FCR-SUG-00

Description

An attacker may leverage the asynchronous nature of transaction validation and UTXO updates across the `txpool` and the node's persistent storage. When a transaction (`txA`) that spends an existing UTXO (`utxoA`) is selected for inclusion in a block, it is removed from the `txpool`. However, the UTXO data update in persistent storage occurs only after block execution finishes, creating a window between the removal of `txA` from the `txpool` and the removal of `utxoA` from persistent storage, during which the `txpool` incorrectly believes that `utxoA` is still available.

Since `txB` relies on an invalid UTXO (`utxoA`), it, along with all its dependent transactions, will never be executed. Thus, the attacker may specify an extremely high gas tip ratio for `txB`, knowing that the transaction will never execute. The attacker may repeatedly submit such transactions based on both `utxoA` and outputs of `txA` to increase the likelihood of successfully exploiting the timing window, as the transaction will only be accepted after `txA` is removed from the `txpool`.

Proof of Concept

1. An existing transaction (`txA`) in the `txpool` spends `utxoA`.
2. When `txA` is included in a block, it is removed from the `txpool`, as are all its related records (such as from `graph_storage` or `collision_manager`).
3. A new transaction (`txB`) utilizing the same `utxoA` is received by the node. It may be inserted without any problems since `txA` is removed, so the `txpool` believes no other transactions use `utxoA`.
4. Execution of the block ends, and `utxoA` is removed from persistent storage. Thus `txB` now becomes an invalid transaction within the `txpool`.

Remediation

Add checks in `txpool` against UTXOs utilized by block transactions currently undergoing execution.

Patch

328b42c removes skipped transactions and their dependencies from `txpool`, which limits the impact of `txpool` flushing from time-to-live of a `txpool` transaction to one block. While this does not fully fix the issue, it greatly limits the impact of the attack, and the remaining risk is accepted for now.

06 — Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the General Findings.

CRITICAL

Vulnerabilities that immediately result in a loss of user funds with minimal preconditions.

Examples:

- Misconfigured authority or access control validation.
- Improperly designed economic incentives leading to loss of funds.

HIGH

Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit.

Examples:

- Loss of funds requiring specific victim interactions.
- Exploitation involving high capital requirement with respect to payout.

MEDIUM

Vulnerabilities that may result in denial of service scenarios or degraded usability.

Examples:

- Computational limit exhaustion through malicious input.
- Forced exceptions in the normal user flow.

LOW

Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk.

Examples:

- Oracle manipulation with large capital requirements and multiple transactions.

INFO

Best practices to mitigate future security risks. These are classified as general findings.

Examples:

- Explicit assertion of critical internal invariants.
- Improved input validation.

07 — Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that others may have missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.