Fuel VM

Security Assessment

June 25th, 2024 — Prepared by OtterSec

James Wang james.wang@osec.io

Alpha Toure shxdow@osec.io

Robert Chen r@osec.io

Table of Contents

01 — Executive Summary

Overview

Fuel Labs engaged OtterSec to assess the `fuel-vm` program. This assessment was conducted between April 17th and June 13th, 2024. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 12 findings throughout this audit engagement.

In particular, we identified a critical vulnerability involving an incomplete memory range check in the code copy functionality. This vulnerability does not properly verify the memory ownership of a memory copy destination buffer, allowing attackers to write to unowned memory (OS-FVM-ADV-00). We also discussed several inconsistencies with the code size, including an issue where the current code size update panics (OS-FVM-ADV-01). Additionally, we noted the underestimation of the size of the callee contract code due to neglecting padding bytes, which may result in incorrect tracking within `CallFrames` (OS-FVM-ADV-02).

We also made recommendations to utilize `checked_arithmetic` to prevent overflow issues (OS-FVM-SUG-03) and advised introducing memory segment separation by creating distinct memory regions for the stack and heap, such that the stack would have its own dedicated address space, preventing its released memory from being reused by the heap (OS-FVM-SUG-01). Additionally, we suggested the removal of redundancy in the system and ensured adherence to best coding practices (OS-FVM-SUG-02).

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/FuelLabs/fuel-vm. This audit was performed against commit 795d2ee.

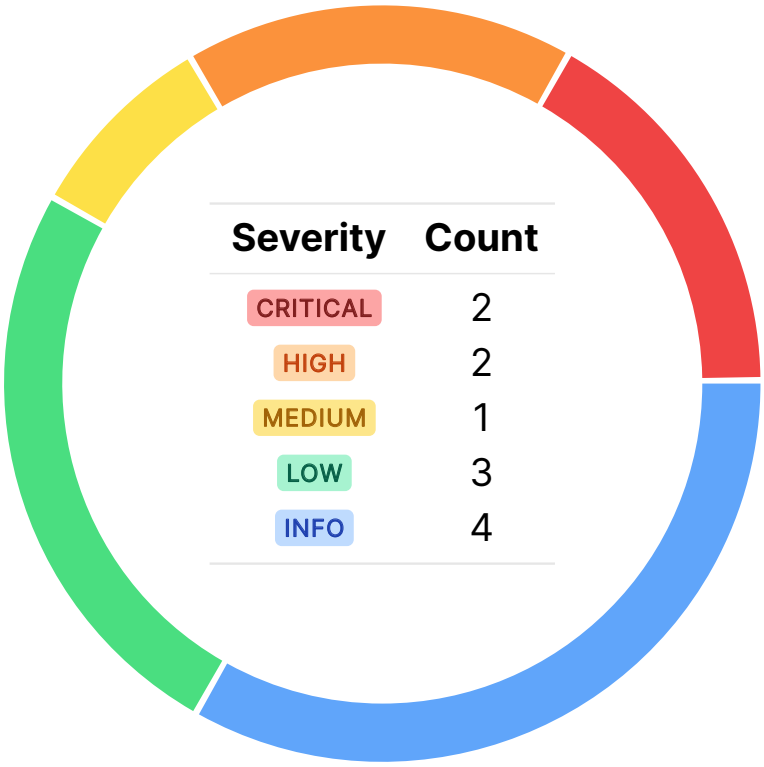
A brief description of the programs is as follows:

Name	Description
fuel-vm	A Rust interpreter for the Fuel Virtual Machine, which executes fuel-asm opcodes generated by the Sway compiler.

03 — Findings

Overall, we reported 12 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	2
MEDIUM	1
LOW	3
INFO	4

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 Appendix A.

ID	Severity	Status	Description
OS-FVM-ADV-00	CRITICAL	RESOLVED	There is an incomplete memory range check in `code_copy` , which misuses the memory access `length` as the end of the memory range, rendering the source contract's memory ownership unsound.
OS-FVM-ADV-01	CRITICAL	RESOLVED	The current code size update in `load_contract_code` confuses the size field length and the size field offset. This results in `ldc` always panicking when called from a contract.
OS-FVM-ADV-02	HIGH	RESOLVED	The current code incorrectly records the size of the contract code in `CallFrame` by neglecting padding bytes, which may result in incorrect execution results for contracts relying on this data.
OS-FVM-ADV-03	HIGH	RESOLVED	During calls, the callee's contract may inherit the caller's flag register state, allowing the caller to potentially manipulate the flags and alter the callee's behavior.
OS-FVM-ADV-04	MEDIUM	RESOLVED	There are several instances within the protocol where the charged gas is used inappropriately.
OS-FVM-ADV-05	LOW	RESOLVED	`Create` transactions in Fuel VM skips the preparation for transaction signing. This omission may result in untrusted transaction input and output being carried over to execution time.
OS-FVM-ADV-06	LOW	RESOLVED	Documentation for Fuel VM's `Upgrade` states that either the `Coin` or `Message` input owner must be privileged for validation. However, the current code only checks `Coin` input ownership.
OS-FVM-ADV-07	LOW	TODO	`Bytes64` to `PublicKey` conversion in Fuel VM is flawed. It expects the raw public key data, but valid `SEC1` encoded keys have a leading tag byte.

Out Of Bounds Memory Access CRITICAL OS-FVM-ADV-00

Description

The vulnerability lies in the ownership check performed by `blockchain::code_copy` in the Fuel virtual machine module. The code intends to check ownership for the destination memory range (`dst_addr to dst_addr + length`) utilizing `self.owner.has_ownership_range`, but incorrectly checks (`dst_addr to length`) instead.

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

```rust
 pub(crate) fn code_copy(
    mut self,
    dst_addr: Word,
    contract_id_addr: Word,
    contract_offset: Word,
    length: Word,
) -> IoResult<(), S::DataError>
where
    [...]
{
    let contract_id = ContractId::from(self.memory.read_bytes(contract_id_addr)?);
    let offset: usize = contract_offset
        .try_into()
        .map_err(|_| PanicReason::MemoryOverflow)?;
    // Check target memory range ownership
    if !self.owner.has_ownership_range(&(dst_addr..length)) {
        return Err(PanicReason::MemoryOverflow.into())
    }
    [...]
}
```

By breaking the memory ownership checks, attackers may write to memory they do not own. In this specific instance, attackers may write to the heap memory of caller contracts or scripts and manipulate the behavior of caller contracts.

Remediation

Check against memory range (`dst_addr to dst_addr + length`) instead of (`dst_addr to length`).

Patch

Resolved in 5156767.

Unchecked Code Size Update CRITICAL OS-FVM-ADV-01

Description

The vulnerability lies in the potential incorrect update of the code size within `load_contract_code`. `load_contract_code` is responsible for loading a contract's code from storage onto the stack.

The vulnerability arises from the way the new code size is fetched and written back into memory. First, `CallFrame::code_size_offset().saturating_add(WORD_SIZE)` is incorrectly treated as the `size` of `code_size`. Second, the `new_code_size` is incorrectly loaded from `fp`. Finally, the result is written back to the incorrect address `fp` with the incorrect `size`.

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

```rust
pub(crate) fn load_contract_code(
    mut self,
    contract_id_addr: Word,
    contract_offset: Word,
    length_unpadded: Word,
) -> IoResult<(), S::DataError>
where
    I: Iterator<Item = &'vm ContractId>,
    S: InterpreterStorage,
{
    [...]
    // Update frame pointer, if we have a stack frame (e.g. fp > 0)
    if fp > 0 {
        let size = CallFrame::code_size_offset().saturating_add(WORD_SIZE);
        let old_code_size = Word::from_be_bytes(self.memory.read_bytes(fp)?);

        let new_code_size = old_code_size
            .checked_add(length as Word)
            .ok_or(PanicReason::MemoryOverflow)?;

        self.memory
            .write_noownerchecks(fp, size)?
            .copy_from_slice(&new_code_size.to_be_bytes());
    }
    inc_pc(self.pc)?;
    Ok(())
}
```

Due to the size difference between the length of the `new_code_size` and `size`, the `copy_from_slice` will always panic.

Remediation

Use `WORD_SIZE` as `size`, fetch `old_code_size` from the address `fp + CallFrame::code_size_offset()`, and write it back to the same address after calculating `new_code_size`.

Patch

Resolved in f90f29b.

Padding Omission In Code Size HIGH OS-FVM-ADV-02

Description

`flow::call_frame` creates a `CallFrame` object with a `code_size` based on the output of `contract_size`. However, `contract_size` does not include the null bytes padded after the `code` for alignment purposes.

>_ *fuel-vm/src/interpreter/flow.rs* rust

```rust
fn call_frame<S>(
    registers: [Word; VM_REGISTER_COUNT],
    storage: &S,
    call: Call,
    asset_id: AssetId,
) -> IoResult<CallFrame, S::Error>
where
    S: StorageSize<ContractsRawCode> + ?Sized,
{
    let (to, a, b) = call.into_inner();

    let code_size = contract_size(storage, &to)?;

    let frame = CallFrame::new(to, asset_id, registers, code_size, a, b);

    Ok(frame)
}
```

Further reliance on the `code_size` may result in unexpected execution results. An instance of this is `ldc` deriving `new_code_size` from the original `code_size`.

Remediation

Record the padded `contract_size` instead of the unpadded one in the `CallFrame`.

Patch

Resolved in f90f29b.

Incorrect Register State Inheritance HIGH OS-FVM-ADV-03

Description

The vulnerability involves the potential inheritance of the callee contract's `flag` register state from the caller context. The `flag` register is used to toggle the behavior of the Fuel VM on invalid arithmetic operations, including division by zero and overflows. Allowing the caller to set the initial `flag` register values for the callee contract might create exploitable instances, such as allowing silent underflow when subtracting the user balance of a coin. This is especially relevant since the Sway compiler does not explicitly clear the `flag` value on contract entry.

Remediation

Set the `flag` register to `zero` on contract calls instead of inheriting the value from the `caller` context.

Patch

Resolved in e568069.

Incorrect Gas Consumption MEDIUM OS-FVM-ADV-04

Description

1. In `fee::gas_used_by_inputs`, `Input` signature verification utilizes `secp256k1`, however, the gas is collected via `gas_costs.ecr1()`. Utilize `eck1()` to collect gas instead of `erc1()`.
2. `load_contract_code` charges gas for loading the contract after fetching it from storage, potentially causing a denial-of-service if the virtual machine (VM) attempts to fetch a large contract, consuming significant resources on the Fuel node. The VM should first query the storage to determine the contract size and calculate the gas cost for loading. Only if the user has enough available gas should the VM proceed.
3. In `instruction`, `wdop` and `wqop` operations are charged incorrect gas (`gas_costs().wdcm()` and `gas_costs().wqcm()`). Instead , it should be `gas_costs().wdop()` and `gas_costs().wqop()`.
4. `cfei`, `cfe`, and `aloc` operations in the Fuel VM handle memory allocation, zeroing memory, and copying memory, but they currently charge a constant gas fee. This incentivizes unnecessary allocation of large memory chunks, resulting in inefficient resource utilization. Ensure gas costs scale linearly with the allocated memory size for optimal memory utilization.
5. Some opcodes iterate through `input` and `output` entries to find specific ones but do not charge gas fees proportionally to their count. Although the `input` and `output` counts are relatively limited, and this may be accounted for in the constant gas cost, it is crucial to consider all aspects of this approach to ensure overall protocol security.
6. `ldc` consumes gas linearly to the `code_size` of the loaded contract. However, a user may specify an even larger buffer size, and the VM will zero the trailing buffer memory after copying the contract code. The zeroing is currently not charged and presents a denial-of-service opportunity.

Remediation

Ensure gas is always charged proportionally to the resource consumed.

Patch

Resolved in d4830e5, e041eff, f94bb46, 882d029, 1192221 and 77c9d9c.

Skipping Transaction Signing Phase LOW OS-FVM-ADV-05

Description

There is a discrepancy in how `Create` transactions are prepared for execution in the Fuel VM's `Interpreter` module. The function `prepare_init_execute` is supposed to prepare the transaction for execution. In the current code for `Create` transactions, this function is empty (`{}`). Skipping `prepare_sign` for `Create` transactions implies that untrusted data within the transaction's `Input` or `Output` fields will not be cleared. Presently, other transaction checks block potential attack opportunities. However, untrusted values in `Input` / `Output` are not appropriate from the standpoint of coding best practices.

>_ *fuel-vm/src/interpreter.rs* rust

```rust
impl ExecutableTransaction for Create {
    fn as_create(&self) -> Option<&Create> {
        Some(self)
    }
    fn as_create_mut(&mut self) -> Option<&mut Create> {
        Some(self)
    }
    fn transaction_type() -> Word {
        TransactionRepr::Create as Word
    }
    fn prepare_init_execute(&mut self) {}
}
```

Remediation

Ensure `prepare_sign` is called for `Create` transactions as well.

Patch

Resolved in 3816c54.

Inconsistency In Documentation And Implementation LOW OS-FVM-ADV-06

Description

The issue highlights an inconsistency between the documented behavior and the actual implementation for validating `Upgrade` transactions in the Fuel VM. According to the documentation, a transaction is deemed valid if either the owner of a `Coin` input or the owner of a `Message` input matches the privileged address (`InputType.Coin.owner == PRIVILEGED_ADDRESS` or `InputType.Message.owner == PRIVILEGED_ADDRESS`). The code snippet only checks the owner of the `Coin` input via `input.input_owner()` ; it does not explicitly check for `Message` input ownership.

>_ *fuel-tx/src/transaction/types/upgrade.rs* rust

```rust
fn check_unique_rules(
    &self,
    consensus_params: &ConsensusParameters,
) -> Result<(), ValidityError> {
    // At least one of inputs must be owned by the privileged address.
    self.inputs
        .iter()
        .find(|input| {
            if let Some(owner) = input.input_owner() {
                owner == consensus_params.privileged_address()
            } else {
                false
            }
        })
        .ok_or(ValidityError::TransactionUpgradeNoPrivilegedAddress)?;
    [...]
}
```

Thus, the current code implementation may overlook a valid transaction with a `Message` input owned by the privileged address and reject it due to the missing ownership check for `Message` inputs. If legitimate `Upgrade` transactions with `Message` inputs are rejected, it may prevent authorized upgrades from being applied.

Remediation

Update the code to align with the documented behavior.

Patch

Resolved in bd97218.

Incorrect Public Key Conversion LOW OS-FVM-ADV-07

Description

`public::try_from` attempts to create a `PublicKey` from a `Bytes64` array. However, the `secp256k1` public key in uncompressed `SEC1` format actually requires 65 bytes:

1. 1 byte for the tag indicating the point format (`uncompressed` in this case).
2. 32 bytes for the `x-coordinate`.
3. 32 bytes for the `y-coordinate`.
4. The `Bytes64` array may only hold 64 bytes.

>_ *fuel-crypto/src/secp256/public.rs* rust

```rust
fn try_from(b: Bytes64) -> Result<Self, Self::Error> {
    match VerifyingKey::from_sec1_bytes(&*b) {
        Ok(_) => Ok(Self(b)),
        Err(_) => Err(Error::InvalidPublicKey),
    }
}
```

The current code takes a `Bytes64` as the argument. Without the leading tag, the provided argument will never be a valid 65-byte `SEC1` encoded public key; thus, `VerifyingKey::from_sec1_bytes` will always fail.

Remediation

Update `try_from` to take `SEC1` encoded public keys.

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-FVM-SUG-00	Fuel transaction validation does not check for the existence of `Output::Change` for each coin that has an initial balance, potentially resulting in the unforeseen loss of user funds.
OS-FVM-SUG-01	The current Fuel VM design combines stack and heap memory, which are susceptible to pointer value type confusion if stack frames are released and the memory is later reused by the heap.
OS-FVM-SUG-02	Suggestions to ensure adherence to coding best practices.
OS-FVM-SUG-03	Recommendation to utilize `checked_arithmetic` to prevent overflow issues.

Insufficient Output check

OS-FVM-SUG-00

Description

Fuel adopts the `UTXO` approach for resource management and burns inputs after transactions, regardless of whether the transaction succeeded or not. In cases where the transaction fails or there is a remaining input balance that is not spent, `update_outputs` will attempt to look for a matching `Output::Change` record to distribute the remaining coins.

>_ fuel-vm/src/interpreter.rs rust

```rust
fn update_outputs<I>(
   [...]
) -> Result<(), ValidityError>
where
    I: for<'a> Index<&'a AssetId, Output = Word>,
{
    let gas_refund = self
        .refund_fee(gas_costs, fee_params, used_gas, gas_price)
        .ok_or(ValidityError::GasCostsCoinsOverflow)?;
    self.outputs_mut().iter_mut().try_for_each(|o| match o {

        Output::Change {
            asset_id, amount, ..
        } if revert && asset_id == base_asset_id => initial_balances.non_retryable
            [base_asset_id]
            .checked_add(gas_refund)
            .map(|v| *amount = v)
            .ok_or(ValidityError::BalanceOverflow),

        // If revert, reset any non-base asset to its initial balance
        Output::Change {
            asset_id, amount, ..
        } if revert => {
            *amount = initial_balances.non_retryable[asset_id];
            Ok(())
        }

        // The change for the base asset will be the available balance + unused gas
        Output::Change {
            asset_id, amount, ..
        } if asset_id == base_asset_id => balances[asset_id]
            .checked_add(gas_refund)
            .map(|v| *amount = v)
            .ok_or(ValidityError::BalanceOverflow),

        // Set changes to the remainder provided balances
        Output::Change {
            asset_id, amount, ..
        } => {
```

```
                *amount = balances[asset_id];
                Ok(())
            }
            [...]
        }
    }
```

In the current implementation, transaction validation does not check whether a `Output::Change` record exist for each input coin. When execution ends and `update_outputs` is called, it attempts to find the matching `Output::Change` field for coins with remaining balance. If no matching `Output::Change` type is present, the remaining balance is silently burnt.

Remediation

Ensure an `Output::Change` exists for each input coin in transaction validation.

Fuel VM Memory Safety

OS-FVM-SUG-01

Description

The Fuel VM currently utilizes a unified memory space for both the stack and the heap. While the stack and heap are designed not to overlap during their growth, there is a potential issue when stack frames are popped (memory is released). This released memory from the stack may be reused for heap allocations. If a contract relies on pointers to specific memory addresses for its data structures, this reuse may result in inconsistencies or even security vulnerabilities.

Remediation

Introduce memory segment separation. This involves creating distinct memory regions for the stack and the heap. The stack will have its own dedicated address space, preventing its released memory from being reused by the heap.

Code Maturity OS-FVM-SUG-02

Description

1. In the Fuel VM implementation, there are several instances of redundant and legacy code, as well as misleading comments. Remove these code instances and update the comments to accurately reflect the functionality. This will ensure improved code readability, better maintainability, and a reduced risk of errors.

>_ fuel-merkle/src/sparse/merkle_tree/node.rs rust

```rust
pub(crate) fn check_common_part<T>([...]) -> Result<(), ValidityError>
where
    T: canonical::Serialize + Chargeable + field::Outputs,
{
    [...]
    if tx
        .outputs()
        .iter()
        .filter_map(|output| match output {
            Output::Change { asset_id, .. } if input_asset_id == asset_id => {
                Some(())
            }
            Output::Change { asset_id, .. }
                if asset_id != base_asset_id && input_asset_id == asset_id =>
            {
                Some(())
            }
            _ => None,
        })
        .count()
        > 1
        {[...]);
    [...]
}
```

2. Ensure that test-only structures are clearly marked to improve code quality and prevent any possibility of misuse in the future.
3. Currently, Fuel VM relies on manually calculating byte offsets for accessing fields within serialized structures by hard-coding them. Implement a macro to replace these hard coded values. This will lower the maintenance cost as well as help prevent mistakes in manual offset calculation.
4. Update the specification to match the current implementation in `interpreter::metadata`.

Remediation

Implement the above-mentioned suggestions.

Prevention Of Overflow Risk OS-FVM-SUG-03

Description

Currently, within the Fuel VM, standard arithmetic operations perform calculations without explicit overflow checks. While faster, unchecked operations may result in unexpected behavior if overflows occur. Utilize `checked_arithmetic` operations whenever possible. This ensures that potential overflows are detected and handled appropriately.

Remediation

Implement `checked_arithmetic` to prevent overflow. For specific operations that are highly unlikely to overflow, explicitly document the expected behavior and the reasons why overflow is not a concern.

A — 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.

B — 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.