Hydration Security Review

Pashov Audit Group

Conducted by: Koolex, FrankCastle, ubermensch

October 17th - October 22th

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **galacticcouncil/hydration-node** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Hydration

Hydration is a DeFi protocol on Polkadot, offering an 'Omnipool' that combines all assets into a single, highly efficient trading pool, reducing slippage and increasing capital efficiency. Through features like single-sided liquidity provisioning, incentivized rewards, and advanced security, Hydration minimizes impermanent loss and ensures safer, more streamlined trading for liquidity providers and DAOs alike.

5. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - 90eb2543cbe037626ed2c5f263f876bc5db6825a

fixes review commit hash - 5d4121cbe3fd852b8e8341d2358c19cf118946bc

Scope

The following smart contracts were in scope of the audit:

- `lib.rs`
- `erc20_currency.rs`
- `executor.rs`
- `evm.rs`
- `erc20_mapping.rs`
- `multicurrency.rs`
- `permit.rs`

7. Executive Summary

Over the course of the security review, Koolex, FrankCastle, ubermensch engaged with Hydration to review Hydration. In this period of time a total of **9** issues were uncovered.

Protocol Summary

Protocol Name	Hydration
Repository	https://github.com/galacticcouncil/hydration-node
Date	October 17th - October 22th
Protocol Type	Liquidity provision protocol

Findings Count

Severity	**Amount**
High	1
Medium	3
Low	5
Total Findings	**9**

Summary of Findings

ID	Title	Severity	Status
[H-01]	TransferFrom is incorrectly treated as a view function	High	Resolved
[M-01]	Unbounded storage iteration in EVM address registration migration	Medium	Acknowledged
[M-02]	Failure to verify ERC20 function return values in handle_result()	Medium	Resolved
[M-03]	Unbounded memory growth via EVM Error message allocations	Medium	Acknowledged
[L-01]	EIP-2 signature malleability in Permit validation	Low	Acknowledged
[L-02]	Missing logging in runtime upgrade implementation	Low	Acknowledged
[L-03]	EVM exit status misclassification in handle_result()	Low	Acknowledged
[L-04]	Oversized EVM error messages due to full value encoding	Low	Acknowledged
[L-05]	Unchecked message size in call()	Low	Acknowledged

8. Findings

8.1. High Findings

[H-01] `TransferFrom` is incorrectly treated as a view function

Severity

Impact: High

Likelihood: Medium

Description

In the `execute` function of the `MultiCurrencyPrecompile` module, the `check_function_modifier` is intended to ensure that function calls are compatible with the execution context, particularly regarding whether they are payable or non-payable functions. However, the `Function::TransferFrom` case is missing from the match statement:

```
handle.check_function_modifier(match selector {
    Function::Transfer => FunctionModifier::NonPayable,
    // Function::TransferFrom is not included here
    _ => FunctionModifier::View,
})?;
```

As a result, `TransferFrom` defaults to `FunctionModifier::View`, which is incorrect because `TransferFrom` is a state-changing function that should be marked as non-payable. Treating it as a view function can lead to unexpected errors or failures when it's invoked, as the execution environment might restrict state changes in contexts meant for view-only operations.

Recommendations

Include `Function::TransferFrom` in the `check_function_modifier` match statement and assign it `FunctionModifier::NonPayable`, similar to the

`Transfer` function:

```
handle.check_function_modifier(match selector {
    Function::Transfer => FunctionModifier::NonPayable,
    Function::TransferFrom => FunctionModifier::NonPayable, // Add this line
    _ => FunctionModifier::View,
})?;
```

This adjustment ensures that `TransferFrom` is correctly recognized as a state-changing function, preventing it from being erroneously treated as a view function and maintaining the integrity of execution contexts.

8.2. Medium Findings

[M-01] Unbounded storage iteration in EVM address registration migration

Severity

Impact: Medium

Likelihood: Medium

Description

`SetCodeForErc20Precompile::on_runtime_upgrade()` performs an unbounded iteration over all assets in the registry to register their EVM addresses. This approach poses several risks:

- **Block Space Exhaustion**: with a large number of assets (>1000), the migration could exceed block weight limits, causing the upgrade to fail.
- **Network Disruption**: A failed upgrade due to exceeded block limits would require network coordination to resolve, potentially leading to downtime.

```
fn on_runtime_upgrade() -> frame_support::weights::Weight {
    pallet_asset_registry::Assets::<Runtime>::iter().for_each(|(asset_id, _)| {
        // ... processing each asset in a single block
    });
}
```

References: link

Recommendations

- Implement a scheduled multi-block migration using the Scheduler pallet:

```
- Define migration state storage
- Process assets in configurable batches (e.g., 100 per block)
- Use the Scheduler pallet to ensure consistent execution
- Track progress for migration resumption
```

[M-02] Failure to verify ERC20 function return values in `handle_result()`

Severity

Impact: High

Likelihood: Low

Description

The `handle_result` function in the `Erc20Currency` implementation is responsible for processing the results of ERC20 contract calls such as `transfer` and `approve`. Currently, this function only checks the `exit_reason` to determine if the EVM call succeeded:

```
fn handle_result(result: CallResult) -> DispatchResult {
    let (exit_reason, value) = result;
    match exit_reason {
        ExitReason::Succeed(ExitSucceed::Returned) => Ok(()),
        ExitReason::Succeed(ExitSucceed::Stopped) => Ok(()),
        _ => Err(DispatchError::Other(&*Box::leak(
            format!("evm:0x{}", hex::encode(value)).into_boxed_str(),
        ))),
    }
}
```

However, some ERC20 tokens return a boolean value indicating the success (`true`) or failure (`false`) of the operation. By not checking the returned data (`value`), the function may incorrectly assume that the operation was successful when, in fact, it was not. This oversight can lead to situations where transfers or approvals are considered successful by the system, even though the ERC20 contract has signaled a failure through its return value.

Recommendations

Modify the `handle_result` function to check the returned data when its length is non-zero. If the data represents a boolean value, decode it and verify that it is `true`. If the decoded boolean is `false`, the function should revert the transaction to prevent misinterpretation of the operation's outcome.

[M-03] Unbounded memory growth via EVM Error message allocations

Severity

Impact: High

Likelihood: Low

Description

In `executor.rs`, the EVM error handling code creates permanent memory allocations for each unique error value through the use of `Box::leak`. While these are not traditional memory leaks from dropping references, they represent a security risk through unbounded memory growth:

```
fn handle_result(result: CallResult) -> DispatchResult {
    let (exit_reason, value) = result;
    match exit_reason {
        ExitReason::Succeed(ExitSucceed::Returned) => Ok(()),
        ExitReason::Succeed(ExitSucceed::Stopped) => Ok(()),
        _ => Err(DispatchError::Other(&*Box::leak(
            format!("evm:0x{}", hex::encode(value)).into_boxed_str(),
        ))),
    }
}
```

Each unique error value creates a new permanent memory allocation because:

- The error value is hex encoded (`hex::encode(value)`)
- This creates a new string for each unique value
- `Box::leak` makes this allocation permanent
- The memory cannot be freed during runtime

An attacker could exploit this by:

- Generating transactions that cause EVM errors
- Ensuring each error has unique data (e.g. ERC20 impl that returned counter++ error everytime transfer is called)
- Each unique error consumes additional memory
- Memory usage grows linearly with unique errors

Example attack pattern:

```
// Each creates a permanent allocation
tx1 -> error [1,2,3] -> "evm:0x010203"
tx2 -> error [1,2,4] -> "evm:0x010204"
tx3 -> error [1,2,5] -> "evm:0x010205"
// Memory grows with each unique error
```

The severity is critical because:

- Memory growth is unbounded
- Allocations are permanent for the node's lifetime
- Affects all network nodes
- Could be used as DoS vector
- Memory cannot be reclaimed without node restart

Recommendations

- Use static error messages without dynamic data (recommended):

```
const EVM_ERROR: &'static str = "EVM execution error";

fn handle_result(result: CallResult) -> DispatchResult {
    let (exit_reason, value) = result;
    match exit_reason {
        ExitReason::Succeed(ExitSucceed::Returned) => Ok(()),
        ExitReason::Succeed(ExitSucceed::Stopped) => Ok(()),
        _ => {
            // Log error details separately if needed
            log::error!("EVM error: 0x{}", hex::encode(&value));
            Err(DispatchError::Other(EVM_ERROR))
        }
    }
}
```

- Implement a bounded error cache (e.g. BTreeMap).

The recommended approach is option 1, as it:

- Completely eliminates the memory growth vector
- Maintains error logging capability
- Is simple to implement and maintain
- Has no performance overhead

8.3. Low Findings

[L-01] EIP-2 signature malleability in Permit validation

Description

The `validate_permit` function in permit.rs does not validate that the `s` value of the ECDSA signature is in the lower half of the curve order, which fails to enforce EIP-2 requirements. This allows signature malleability where multiple valid signatures can exist for the same message. While this doesn't create an immediate security risk, it deviates from Ethereum standards and best practices.

References: https://eips.ethereum.org/EIPS/eip-2

Recommendations

Add validation for the `s` value to ensure it's in the lower half of the curve order.

[L-02] Missing logging in runtime upgrade implementation

`SetCodeForErc20Precompile::on_runtime_upgrade` lacks logging mechanisms to track the progress and results of the upgrade process. This makes it difficult to:

- Monitor the upgrade progress in production
- Debug issues if the upgrade fails or behaves unexpectedly
- Audit the changes after the upgrade is completed
- Verify that all assets were processed correctly

Consider adding structured logging throughout the upgrade process.

```rust
impl frame_support::traits::OnRuntimeUpgrade for SetCodeForErc20Precompile {
    fn on_runtime_upgrade() -> frame_support::weights::Weight {
        log::info!("Starting ERC20 precompile code setup");

        // Track statistics
        let mut assets_processed = 0;
        let mut assets_updated = 0;

        // Log individual updates
        pallet_asset_registry::Assets::<Runtime>::iter().for_each(|
          (asset_id, _)| {
            assets_processed += 1;
            // Log progress...
        });

        // Log final results
        log::info!(
            "Completed setup. Processed: {}, Updated: {}",
            assets_processed,
            assets_updated
        );

        // Return weight...
    }
}
```

And consider adding pre/post upgrade logging (and validation if necessary), for example:

```rust
#[cfg(feature = "try-runtime")]
fn pre_upgrade() -> Result<Vec<u8>, &'static str> {
    log::info!("Starting pre_upgrade checks");
    Ok(Vec::new())
}
```

[L-03] EVM exit status misclassification in `handle_result()`

The `handle_result` function only considers `ExitSucceed::Returned` and `ExitSucceed::Stopped` as successful outcomes while treating all other exit reasons as errors:

```rust
fn handle_result(result: CallResult) -> DispatchResult {
    let (exit_reason, value) = result;
    match exit_reason {
        ExitReason::Succeed(ExitSucceed::Returned) => Ok(()),
        ExitReason::Succeed(ExitSucceed::Stopped) => Ok(()),
        _ => Err(DispatchError::Other(&*Box::leak(
            format!("evm:0x{}", hex::encode(value)).into_boxed_str(),
        ))),
    }
}
```

This implementation incorrectly treats `ExitSucceed::Suicided` as an error condition, despite it being a valid and intentional outcome in EVM operations. When a smart contract executes a self-destruct operation (SELFDESTRUCT opcode), it returns `ExitSucceed::Suicided` to indicate successful contract destruction.

As a result:

- Valid contract self-destruct operations will be marked as failed transactions
- This diverges from standard EVM behavior where self-destruct is a valid operation

Suicide code from SputnikVM:

```
machine: &mut Machine<S>,
        handler: &mut H,
) -> Control<Tr> {
        .
        match machine.stack.perform_pop1_push0(|target| {
        .
        .
                Ok(((), ()))
        }) {
                Ok(()) => Control::Exit(ExitSucceed::Suicided.into()),
                Err(e) => Control::Exit(Err(e)),
        }
}
```

interpreter/src/eval/system.rs#L370

SputnikVM is an EVM engine used in frontier by moonbeam-foundation (which is a fork from polkadot-evm/frontier), which is utilized in Hydration.

```
[[package]]
name = "pallet-evm"
version = "6.0.0-dev"
source =
// "git+https://github.com/moonbeam-foundation/frontier?branch=moonbeam-polkadot-v1.11
```

hydration-node/blob/evm-binding/Cargo.lock#L8237

Note: given that the use of `handle_result` is currently limited to ERC20 operations, the likelihood of self-destruct opcode is low. However, it still does not follow the standard EVM behavior.

The handle_result function should be modified to properly handle all ExitSucceed variants as successful operations:

```rust
fn handle_result(result: CallResult) -> DispatchResult {
    let (exit_reason, value) = result;
    match exit_reason {
        ExitReason::Succeed(_) => Ok(()), // Accept all ExitSucceed variants
        _ => Err(DispatchError::Other(&*Box::leak(
            format!("evm:0x{}", hex::encode(value)).into_boxed_str(),
        ))),
    }
}
```

[L-04] Oversized EVM error messages due to full value encoding

The `handle_result` function in `erc20_currency.rs` implements error handling for EVM calls by encoding the entire return value into error messages using hex encoding.

```rust
fn handle_result(result: CallResult) -> DispatchResult {
    let (exit_reason, value) = result;
    match exit_reason {
        ExitReason::Succeed(ExitSucceed::Returned) => Ok(()),
        ExitReason::Succeed(ExitSucceed::Stopped) => Ok(()),
        _ => Err(DispatchError::Other(&*Box::leak(
            format!("evm:0x{}", hex::encode(value)).into_boxed_str(),
        ))),
    }
}
```

The function blindly encodes the entire `value` parameter into the error message for all failed EVM operations. This design becomes problematic when dealing with EVM operations that can produce large return values.

Given Polkadot's EVM compatibility layer, this poses some risks:

- Large error messages require significant memory allocation for string formatting and hex encoding.
- If these errors are logged on-chain or stored in network history, they create unnecessary storage bloat.

Limit the error information, for example, truncate it to a reasonable size or return a hash of it, if it exceeds a certain size limit.

[L-05] Unchecked message size in `call()`

In the executor's `call` function, the input data is accepted without size validation, as demonstrated in the snippet below:

```
fn call
  (context: CallContext, data: Vec<u8>, value: U256, gas: u64) -> CallResult {
    Self::execute(context.origin, gas, |executor| {
        executor.transact_call
          (context.sender, context.contract, value, data, gas, vec![])
    })
}
```

This function processes the `data` vector, which represents the message being passed to the contract. However, there is no size validation for the input data, meaning a very large message could be sent without restriction.

Allowing large, unchecked messages to be processed can result in excessive gas consumption, potentially depleting gas resources and disrupting execution. This vulnerability could lead to Denial of Service (DoS) attacks where the system is forced to process large messages, resulting in unintended behavior or crashing due to running out of gas.

Before executing the message, validate the size of the input data against a maximum allowable limit to prevent overly large messages from being processed. For example, add a check to reject any message whose size exceeds the predefined maximum, ensuring gas resources are not needlessly consumed. Example:

```
fn call
  (context: CallContext, data: Vec<u8>, value: U256, gas: u64) -> CallResult {
    let max_size: usize = 1024; // Set an appropriate maximum size limit
    if data.len() > max_size {
        return Err("Data size exceeds maximum allowed limit");
    }

    Self::execute(context.origin, gas, |executor| {
        executor.transact_call
          (context.sender, context.contract, value, data, gas, vec![])
    })
}
```

This will prevent large messages from causing gas exhaustion and protect the system from potential DoS attacks.