Argo

Audit

Presented by:

OtterSec contact@osec.io

Robert Chen r@osec.io

Akash Gurugunti Sud0u53r.ak@osec.io

Contents

01 | Executive Summary

Overview

Argo engaged OtterSec to perform an assessment of the `argo-move` program. This assessment was conducted between October 3rd and October 21st, 2022.

Critical vulnerabilities were communicated to the team prior to the delivery of the report to speed up remediation. After delivering our audit report, we worked closely with the team over to streamline patches and confirm remediation. We delivered final confirmation of the patches October 23rd, 2022.

Key Findings

Over the course of this audit engagement, we produced 8 findings total.

Originally, Argo used a very modular design with many interacting submodules. While this design makes it easier to compose, it also exposes a lot of moving parts and audit complexity, as we opined prior to the start of the audit.

For example, we discovered two instances of broken access control (OS-ARG-ADV-00, OS-ARG-ADV-01) which could directly lead to loss of funds. We also found additional concerns in the liquidation business logic, oracle prices, and more.

We also made general recommendations around safer design choices and rate limits (OS-ARG-SUG-00, OS-ARG-SUG-01).

Overall, we commend the Argo team for being very responsive to feedback, even in light of our recommendations for large architectural changes.

02 | Scope

The source code was delivered to us in a git repository at github.com/argodao/argo-move. This audit was performed against commit `0adc35c`.

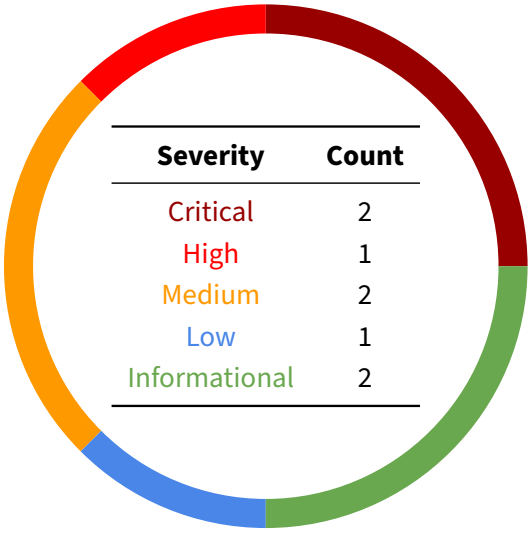
A brief description of the programs is as follows.

Name	Description
argo-move	Argo protocol smart contracts for minting of overcollateralized stablecoins

03 | Findings

Overall, we report 8 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 don't have an immediate impact but will help mitigate future vulnerabilities.



Severity	Count
Critical	2
High	1
Medium	2
Low	1
Informational	2

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-ARG-ADV-00	Critical	Resolved	Verify namespace addresses between `cap` and `manage_cap` in `lib_capability`
OS-ARG-ADV-01	Critical	Resolved	Broken liquidation access control allows liquidators to skip repayment
OS-ARG-ADV-02	High	Resolved	Lack of delay in liquidation marking prevents liquidations in certain circumstances.
OS-ARG-ADV-03	Medium	Resolved	Vaults close to the minimum debt threshold cannot be liquidated
OS-ARG-ADV-04	Medium	Resolved	Check oracle confidence before using prices
OS-ARG-ADV-05	Low	Resolved	`required_repay_amount_internal` should round up the required repayment

OS-ARG-ADV-00 [crit] [resolved] | Missing MeterCapability Checks

Description

In `meter_capability`, the `add_meter_cap_usage` and `sub_meter_cap_usage` functions are security critical checks on the minting/burning of tokens.

lib_capability/sources/meter_capability.move RUST

```
    public fun add_meter_cap_usage<Feature>(
        cap: &MeterCap<Feature>,
        add_amount: u64,
        manage_cap: &ManageMeterCap<Feature>,
    ) acquires GlobalMeter {
        let global_meter =
↪   borrow_global_mut<GlobalMeter<Feature>>(manage_cap.namespace_addr);
        let meter = vector::borrow_mut(&mut global_meter.caps, cap.id);
        let new_global_usage = global_meter.usage + add_amount;
        let new_usage = meter.usage + add_amount;
```

These functions take in a `MeterCap` which corresponds to the ability to mint some strictly limited amount tokens in a given namespace. This namespace is defined and managed by the `ManageMeterCap`, as seen in `laboratory::mint`.

argo_core/sources/laboratory.move RUST

```
    /// Mints `mint_amount` USDA.
    public fun mint(
        mint_amount: u64,
        cap: &MeterCap<USDASupplyFeature>,
    ): Coin<USDA> acquires Laboratory {
        let laboratory = borrow_global<Laboratory>(@argo_core);
        meter_capability::add_meter_cap_usage(
            cap,
            mint_amount,
            &laboratory.usda_supply_manage_cap,
        );
```

Unfortunately, the `MeterCap` type is not unique.

Anybody is able to create a `GlobalMeter<USDASupplyFeature>` and claim the corresponding MeterCap. Note that MeterCap's id would overlap with an existing id on the namespace, allowing a malicious user to essentially forge a MeterCap.

lib_capability/sources/meter_capability.move RUST

```
    /// Receive a MeterCap<Feature> that has a limit default of 0.
    public fun claim_cap<Feature>(
        namespace_addr: address
    ): MeterCap<Feature> acquires GlobalMeter {
```

From here it's trivial to mint arbitrary amounts of USDA.

Remediation

Verify that `cap` and `manage_cap` have the same namespace address.

Patch

Resolved in 8711245.

lib_capability/sources/meter_capability.move RUST

```
  assert!(
      cap.namespace_addr == manage_cap.namespace_addr,
      error::invalid_argument(ENAMESPACE_MISMATCH),
  );
```

After discussion with the Argo team, they also redesigned their architecture to remove `lib_capability` and `lib_rate_limit`.

OS-ARG-ADV-01 [crit] [resolved] | Broken Liquidation Access Control

Description

Argo implements liquidations via a flashloan system using the hot potato method, returning a LiquidateIOU object with no abilities.

argo_engine/sources/engine_v1.move RUST

```
    public fun liquidate_withdraw<NamespaceType, CoinType>(
        owner_addr: address,
        liquidate_amount: u64,
        cap: &Cap<LiquidateFeature<NamespaceType, CoinType>>,
    ): (Coin<CoinType>, LiquidateIOU<NamespaceType, CoinType>) acquires
     ↪  Engine, Vault {
```

The intended behavior is to interact directly with `argo_liquidate` as a wrapper over the underlying Argo Engine functions.

argo_liquidate/sources/liquidate_v1.move RUST

```
public fun liquidate_withdraw<NamespaceType, CoinType>(
    params_addr: address,
    owner_addr: address,
    liquidate_amount: u64,
): (Coin<CoinType>, LiquidateIOU<NamespaceType, CoinType>) acquires
     ↪  LiquidateParams {
    let params = borrow_global<LiquidateParams<NamespaceType,
     ↪  CoinType>>(params_addr);
    return engine_v1::liquidate_withdraw<NamespaceType, CoinType>(
        owner_addr,
        liquidate_amount,
        &params.liquidate_cap,
    )
}
```

Critical security checks are also performed in the `argo_liquidate` handler, such as asserting that the correct amount is repaid by the liquidator.

argo_liquidate/sources/liquidate_v1.move RUST

```
        assert!(
            max_repay_amount >= required_repay_amount,
```

```
            error::invalid_argument(EREPAY_NOT_ENOUGH),
        );
```

Access control between `argo_liquidate` and `argo_engine` is enforced through the use of a LiquidateFeature capability.

Unfortunately, this capability access control requirement is not enforced on `liquidate_repay`.

argo_engine/sources/engine_v1.move RUST

```
    public fun liquidate_repay<NamespaceType, CoinType>(
        to_repay: Coin<USDA>,
        liquidation_tax: Coin<USDA>,
        iou: LiquidateIOU<NamespaceType, CoinType>,
    ) acquires Engine, Vault {
```

This means a liquidator can simply payback 1 token.

Proof of Concept

1. Call `argo_liquidate::liquidate_withdraw` and withdraw all of an underwater position's collateral
2. Call `argo_engine::liquidate_repay` and repay 1 USDA.

Patch

Similar to OS-ARG-ADV-00, Argo removed `argo_liquidate` and flattened their architecture.

OS-ARG-ADV-02 [high] [resolved] | Liquidation Remarking

Description

Argo uses a descending auction system to process liquidations. When a vault is undercollateralized and eligible for liquidation, it becomes ”marked” and the descending auction begins.

argo_engine/sources/engine_v1.move RUST

```
/// Mark a Vault for liquidation. A Vault can only be marked if it is
    ↪  below the
/// maintenance_collateral_ratio and the Safe is fresh.
public fun mark_vault<NamespaceType, CoinType>(
  marker: &signer,
  owner_addr: address,
) acquires Engine, Vault {
```

Unfortunately, this function does not ensure that the vault was not previously marked. As a result, a user attempting to prevent the liquidation of their vault can repeatedly mark their own vault to reset the descending auction.

argo_engine/sources/engine_v1.move RUST

```
    /// Gas-efficient calculation of auction_price
    fun auction_price_internal<NamespaceType, CoinType>(
        engine: &Engine<NamespaceType, CoinType>,
        liquidator_addr: address,
        owner_addr: address,
    ): u64 acquires Vault {
```

Note that there are some preconditions for exploitation.

The descending auction price starts at `oracle_free_price_internal` which represents the expected collateral price derived from the maintenance ratio and debt value. There is also a liquidation delay which could make this issue more impactful.

A liquidator could potentially atomically mark and liquidate the vault if the initial price for the auction is higher than the actual collateral value, depending on how `liquidate_delay` and `marker_advantage` are set.

Remediation

Ensure that the vault is not already marked in `mark_vault`.

Patch

Resolved in 2c31c5c.

argo_engine/sources/engine_v1.move RUST

```
        assert!(vault.mark_info.marker_addr == @0,
    ↪   error::invalid_state(EALREADY_MARKED));
```

OS-ARG-ADV-03 [med] [resolved] | Liquidate Minimum Debt Vaults

Description

Argo enforces a minimum debt threshold when repaying vaults.

Unfortunately, `liquidate_repay` also enforces that the collateral ratio of the vault isn't repaid fully.

argo_engine/sources/engine_v1.move RUST

```
        let collateral_ratio = collateral_ratio_internal(engine, vault);
        assert!(
            collateral_ratio < engine.liquidation_collateral_ratio,
            error::invalid_argument(ELIQUIDATE_TOO_MUCH),
        );
```

This means that vaults that are close to the minimum debt threshold cannot be liquidated.

Remediation

Rework the minimum collateral ratio check

Patch

Resolved in 2c31c5c.

argo_engine/sources/engine_v1.move RUST

```
        let collateral_ratio = collateral_ratio(
            coin::value(&vault.collateral),
            max(scaled_debt_internal(engine, vault), engine.minimum_debt),
            safe::price(engine.safe_addr),
            coin::decimals<CoinType>(),
        );
        assert!(
            collateral_ratio < engine.liquidation_collateral_ratio,
            error::invalid_argument(ELIQUIDATE_TOO_MUCH),
        );
```

OS-ARG-ADV-04 [med] [resolved] | Oracle Confidence Checks

Description

High oracle confidence values indicate that providers disagree on the actual price. Pyth, for example, represents confidence as the difference between the 25/75th quartile and the median price.

In this case, it's safer to ignore the value than to use a potentially inaccurate value.

Remediation

Check the confidence of oracles.

Patch

Resolved in 2c31c5c.

argo_engine/sources/engine_v1.move RUST

```
        let confidence_bps = scale_ceil(conf, BPS_PRECISION, magnitude);
        if (confidence_bps > oracle.max_conf_bps) {
            return
        };
```

OS-ARG-ADV-05 [low] [resolved] | Incorrect Repay Rounding

Description

The required USDA repaid is calculated in `required_repay_amount_internal`. This function should round up instead of down to properly round against the user. Otherwise, for small repayment amounts, it might be possible to further decrease the health of the vault.

argo_liquidate/sources/liquidate_v1.move RUST

```
    /// Gas-efficient calculation of required_repay_amount
    fun required_repay_amount_internal<NamespaceType, CoinType>(
        params: &LiquidateParams<NamespaceType, CoinType>,
        liquidator_addr: address,
        owner_addr: address,
        liquidate_amount: u64,
    ): u64 {
        return math::scale_floor(
            liquidate_amount,
            auction_price_internal(params, liquidator_addr, owner_addr),
            PRICE_PRECISION
        )
    }
```

Remediation

Use `scale_ceil`.

Patch

Resolved in 2c31c5c.

argo_engine/sources/engine_v1.move RUST

```
    /// Gas-efficient calculation of required_repay_amount
    fun required_repay_amount_internal<NamespaceType, CoinType>(
        engine: &Engine<NamespaceType, CoinType>,
        liquidator_addr: address,
        owner_addr: address,
        liquidate_amount: u64,
    ): u64 acquires Vault {
        return scale_ceil(
            liquidate_amount,
```

```
            auction_price_internal(engine, liquidator_addr, owner_addr),
            PRICE_PRECISION
        )
    }
```

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 antipatterns and could lead to security issues in the future.

ID	Description
OS-ARG-SUG-00	Unify health checks for collateral ratio and minimum debt
OS-ARG-SUG-01	USDA rate limits can be bypassed by up to a factor of two on reset boundaries

OS-ARG-SUG-00 [resolved] | Unify Health Checks

Description

Argo currently uses a number of disjoint checks for each function that interacts with collateral ratio.

argo_engine/sources/engine_v1.move RUST

```
assert!(
  withdraw_passes_initial_collateral_ratio_internal(engine, vault,
    ↪  amount),
  error::invalid_argument(ECOLLATERAL_RATIO_TOO_LOW),
);
```

argo_engine/sources/engine_v1.move RUST

```
  assert!(
      mint_passes_minimum_debt_internal(engine, vault, amount),
      error::invalid_argument(EBELOW_MINIMUM_DEBT),
  );
```

It would be cleaner to unify these checks by checking against the collateral ratio after the relevant operations.

Patch

Resolved in 2c31c5c.

argo_engine/sources/engine_v1.move RUST

```
  // Check resulting debt is greater than the minimum debt and resulting
    ↪  collateral ratio is
  // above the initial collateral ratio
  assert!(
      scaled_debt_internal(engine, vault) >= engine.minimum_debt,
      error::invalid_argument(EBELOW_MINIMUM_DEBT),
  );
```

OS-ARG-SUG-01 [resolved] | USDA Timed Rate Limit

Description

USDA `usda_timed_limit` resets discretely after a period of `usda_timed_duration` seconds.

argo_engine/sources/engine_v1.move RUST

```
        fun update_timed_period(laboratory: &mut Laboratory) {
            let now = timestamp::now_seconds();
            let time_elapsed = now - laboratory.usda_timed_last_reset;
            if (time_elapsed > laboratory.usda_timed_duration) {
                laboratory.usda_timed_last_reset = now;
                laboratory.usda_timed_usage = 0;
            };
        }
```

Remediation

This isn't particularly impactful, and can likely be mitigated by adjusting the limits such that double the rate limit is still acceptable.

Patch

Argo acknowledges the rate limit behavior and will choose parameters accordingly.

A | Vulnerability Rating Scale

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

Critical	Vulnerabilities that immediately lead to 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 could lead to 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 could lead to denial of service scenarios or degraded usability. Examples: • Malicious input that causes computational limit exhaustion • Forced exceptions in normal user flow
Low	Low probability vulnerabilities which could still be exploitable but require extenuating circumstances or undue risk. Examples: • Oracle manipulation with large capital requirements and multiple transactions
Informational	Best practices to mitigate future security risks. These are classified as general findings. Examples: • Explicit assertion of critical internal invariants • Improved input validation