Maseer Protocol Security Report

This report was produced for the Maseer Protocol by Prototech Labs

Mail: info@prototechlabs.dev
Twitter: @Prototech_Labs
Web: www.prototechlabs.dev

Versions:
Initial: 6th April 2025
Final: 14th April 2025

Contents

1. Executive Summary

This report was prepared for the Maseer team by Prototech Labs, a smart contract consultancy providing security, technical advisory, and code review services. Prototech Labs would like to thank the Maseer team for giving us the opportunity to review the current state of their protocol.

This document outlines the findings, limitations, and methodology of our review, which is broken down by issue and categorized by severity. It is our hope that this provides valuable findings and insights into the current implementation.

2. Project Overview

This review focused on identifying protocol invariants in order to develop a set of test cases to assess protocol correctness. This then enabled us to execute a test suite against Maseer's contracts to record and analyze the results and any deviations from expected behavior, which were then captured as issues.

Project Details:

- **Timeline:** through 17th March 2024 to 4th April 2025
- **Code Repo:** https://github.com/Maseer-LTD/maseer-one/releases/tag/v0.0.1
- **Commit hash:** 87c1715700057d95737486a238bd7b4e418507d6

3. Introduction

The Maseer Protocol is tokenizing sustainable real-world assets and bringing them on-chain for use in DeFi. This novel initiative involves matching off chain assets and their pricing with on-chain token issuance. As a result, Prototech Labs has paid particular attention to oracle pricing/authorizations, on-chain market configurations, and a comprehensive analysis of buy/sell rounding calculations, redemption and mint paths.

In summary, the codebase provides a high level of security and no critical issues were found. It was observed and appreciated that the architectural design utilises MakerDAO-like naming conventions for brevity thereby reducing cognitive overhead and making it easier to infer technical relationships across the protocol. As a final mention, it is worth stating that due to the protocols' upgradability design and reliance on multisig wallets for privileged access control we recommend industry standard monitoring alongside continued test suite analysis for any proposed changes as a best practice.

4. Limitations and Report Use

Disclaimer: No assessment can guarantee the absolute safety or security of a software-based system. Further, a system can become unsafe or insecure over time as it and/or its environment evolves. This assessment aimed to discover as many issues and make as many suggestions for improvement as possible within the specified timeframe. Undiscovered issues, even serious ones, may remain. Issues may also exist in components and dependencies not included in the assessment scope.

The software systems herein are emergent technologies and carry with them high levels of technical risk and uncertainty. This report and related analysis of projects to not constitute statements, representations or warranties of Prototech Labs in any respect, including regarding the security of the project, utility of the project, suitability of the project's business model, a project's regulatory or legal status or any other statements, representations or warranties about fitness of the project, including those related to its bug free status. You may not rely on the reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. Our complete terms of service can be reviewed here.

Specifically, for the avoidance of doubt, any report published by Prototech Labs does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of any client or project, and is not a guarantee as to the absolute security of any project. Prototech Labs does not owe you any duty by virtue of publishing these reports.

5. Findings Framework

Findings and recommendations are listed in the below section, grouped into broad categories. It is up to the team behind the code to ultimately decide whether the items listed here qualify as issues that need to be fixed, and whether any suggested changes are worth adopting. When a response from the team regarding a finding is available, it is provided.

Findings are given a severity rating based on their likelihood of causing harm in practice and the potential magnitude of their negative impact. Severity is only a rough guideline as to the risk an issue presents, and all issues should be carefully evaluated.

Severity Level Determination		Impact		
		High	Medium	Low
Likelihood	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low

Additionally, issues that do not present any quantifiable risk are given a severity of Informational.

6. Findings Overview

Below is an overview of the findings, split by severity, illustrating their status (Fixed/Acknowledged or Nofix - where item is not considered an issue):

7. Critical Risks	
None	

8. High Risks	[1]
8.1 Rounding Issues in MaseerGate and MaseerOne	Fixed

9. Medium Risks	[1]
9.1 Oracle Authorization System in MaseerPrice Creates Unnecessary Trust Assumptions	Fixed

10. Low Risks	[7]
10.1 Issuer's Ability to Burn Tokens from Blacklisted Users Creates Authorization Model Inconsistency	Acknowledged
10.2 Blacklisted Users Can Execute Precommitments, Bypassing Authorization Model	Fixed
10.3 Blacklisted Users Can Exit Precommitments, Bypassing Authorization Model	Fixed
10.4 Blacklisted Users Can Move Funds Through Conduit, Bypassing Authorization Model	Acknowledged

10.5 Chain ID Replay Attack Vulnerability in Permit System	Fixed
10.6 Zero Claim Redemptions Create Poor User Experience	Fixed
10.7 Ambiguous Oracle Decimals and Price Normalization	Acknowledged

11. Informational Findings	**[5]**
11.1 Multisig Governance Risks and Mitigations	Acknowledged
11.2 Contract Upgradeability Risks	Acknowledged
11.3 Phantom Overflow in Price Adjustments Can Break Core Protocol Functions	Acknowledged
11.4 Inconsistent Zero Address Validation in MaseerConduit.move()	Fixed
11.5 ERC20 Transfers to Address(0) Allowed in MaseerOne	Acknowledged

12. Gas Optimization	**[2]**
12.1 Inconsistent Error Handling Styles Between MaseerToken and MaseerOne	Acknowledged
12.2 MaseerGate mixes paradigms for setting state	Acknowledged

7. Critical Risks

none

8. High Risks

8.1 Rounding Issues in MaseerGate and MaseerOne

Context

MaseerGate.sol#L189
MaseerOne.sol#L385

Description

Two rounding issues have been identified that affect the protocol's price calculations and token operations:

1. **MaseerGate._adjustBurnPrice() Extra Wei**
    - The function adds an extra +1 wei to the burn price calculation
    - This causes `invariant_MP_priceAdjustment()` to fail as `act.burncost(_price) <= _price` no longer holds
    - Results in users receiving more USDT in redemptions than intended
2. **MaseerOne._wmul() Aggressive Rounding**
    - Current implementation rounds towards the user with `(x * y + WAD/2) / WAD`
    - This aggressive rounding accumulates over multiple operations
    - When combined with the extra wei in `_adjustBurnPrice()`, creates significant rounding discrepancies

Impact

1. **Price Adjustment Invariant Failure**
    - The protocol's price adjustment invariant fails to hold
    - This indicates a deviation from expected price behavior
2. **Redemption Value Discrepancy**
    - Users receive more USDT than mathematically correct in redemptions
    - With 90,618 loops, the discrepancy reaches 191.294598 USDT, but could be worse
    - This allows an attacker to extract more USDT value than price and output bps merit

Test Results

Here is a test that can reproduce this if included into @MaseerOne.operations.t.sol

Unset

```
diff --git a/test/MaseerOne.operations.t.sol
b/test/MaseerOne.operations.t.sol
index 0505bd4..6dd3bb6 100644
--- a/test/MaseerOne.operations.t.sol
+++ b/test/MaseerOne.operations.t.sol
@@ -298,4 +298,63 @@ contract MaseerOneOperationsTest is
MaseerTestBase {
         maseerOne.smelt(bob, 100 * 1e18);
     }
```

```
+    function testBurncost() public {
+        uint256 _loops = 90_618;
+
+        uint256 _price = 523;
+        uint256 _bps = 50;
+
+        uint256 _usdt = 100_000 * 1e6;
+        uint256 _unit = (_price * 10_000) / (10_000 + _bps);
+
+        vm.prank(actAuth);
+        act.setCapacity(1_000_000_000 * 1e18); // 1B Cana total
supply
+
+        vm.prank(pipAuth);
+        pip.poke(_price);
+        vm.prank(actAuth);
+        act.setCooldown(1);
+        vm.prank(actAuth);
+        act.setBpsin(0);
+        vm.prank(actAuth);
+        act.setBpsout(_bps);
+        vm.prank(actAuth);
+        act.setHaltBurn(block.timestamp + 365 days);
+
+        vm.prank(alice);
+        usdt.approve(address(maseerOne), _usdt);
+
+        vm.prank(alice);
+        maseerOne.mint(_usdt);
+
+        uint256 _amt = maseerOne.balanceOf(alice);
+        uint256 _quantity = _amt/_loops;
+        uint256 _sum = 0;
+        uint256 _claim = 0;
+        for (uint256 i = 0; i < _loops; i++) {
+            vm.prank(alice);
```

```
+            uint256 _id = maseerOne.redeem(_quantity);
+
+            vm.warp(block.timestamp + maseerOne.cooldown() + 1);
+
+            vm.prank(alice);
+            _sum += maseerOne.exit(_id);
+
+            (uint256 _left,,) = maseerOne.redemptions(_id);
+            assertEq(_left, 0, "there is nothing left over");
+
+            _claim += ((_quantity * _unit) + (WAD / 2)) / WAD;
+            // _claim += (_quantity * _unit) / WAD;
+        }
+
+        console.log("Sum:", _sum);
+        console.log("Claim:", _claim);
+        console.log("Delta:", _sum - _claim);
+        assertLe(_sum - _claim, 191294598, "delta limit at loops
90618");
+        // Logs:
+        // Sum: 99617636052
+        // Claim: 99426341454
+        // Delta: 191.294598
+
+    }
```

If you run this test you should see the output that is commented at the end. If you remove the +1 wei in `_adjustBurnPrice()`, the delta will be better because the code now matches for sum and delta matches:

```
Unset
[PASS] testBurncost() (gas: 7941624897)
Logs:
  Sum: 99426341454
  Claim: 99426341454
```

```
  Delta: 0
```

But is this correct? Assuming there are no outgoing bips, this should be equal to what was deposited. To test this, you can set the outgoing bps to 0 and run the test again:

```
Unset
Failing tests:
Encountered 1 failing test in
test/MaseerOne.operations.t.sol:MaseerOneOperationsTest
[FAIL: there is nothing left over: 44012 != 0] testBurncost()
(gas: 9926965646)
```

If you comment out that assert, you can see that we have already extracted `100_000` USDT, and that there is `44012 wei` left. This isn't ideal, as this does feel like, given no bps scraped for `mint()` or `burn()`, we are rounding towards the user. This over-aggressive rounding, however, is a result of the `MaseerOne._wmul()`. A simple `_wmul()` would just be:

```
Unset
    function _wmul(uint256 x, uint256 y) internal pure returns
(uint256 z) {
        z = (x * y) / WAD;
    }
```

In fact, if we change `_claim` to the commented out version in our test that matches this `_wmul()`, we should get this output:

```
Unset
[PASS] testBurncost() (gas: 7525440193)
Logs:
  Sum: 100000000000
  Claim: 99999953394
  Delta: 46606
```

This seems to be rounding against the user on the redemption now by `46606`. If I fix the `_wmul()` to the one we suggested (you can uncomment the assert now too) and run the tests again we get:

```
Unset
[PASS] testBurncost() (gas: 7922198638)
Logs:
  Sum: 99999953394
  Claim: 99999953394
  Delta: 0
```

We are rounding against the user, but why is it so harsh? This should not be any worse than 1 wei against the user. Well, this is because we do many loops and we are allowing that 1 wei to go against the user under this regime. If you change loops to 1 you'll get:

```
Unset
Ran 1 test for
test/MaseerOne.operations.t.sol:MaseerOneOperationsTest
[PASS] testBurncost() (gas: 279272)
Logs:
  Sum: 100000000000
  Claim: 100000000000
  Delta: 0

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in
182.62ms (24.61ms CPU time)

Ran 1 test suite in 365.10ms (182.62ms CPU time): 1 tests passed,
0 failed, 0 skipped (1 total tests)
```

Recommendation

1. Remove Extra Wei in _adjustBurnPrice()
2. Simplify _wmul() Implementation

Suggested Code Changes

Unset

```
1. **Remove Extra Wei in _adjustBurnPrice()**

  diff --git a/src/MaseerGate.sol b/src/MaseerGate.sol
  index a160575..39e3064 100644
  --- a/src/MaseerGate.sol
  +++ b/src/MaseerGate.sol
  @@ -186,7 +186,7 @@ contract MaseerGate is MaseerImplementation
{
        }

        function _adjustBurnPrice(uint256 _price, uint256 _bps)
internal pure returns (uint256) {
  -         return (_price * 10_000) / (10_000 + _bps) + 1; //
Round down
  +         return (_price * 10_000) / (10_000 + _bps); // Round
down
        }
    }


2. **Simplify _wmul() Implementation**

  diff --git a/src/MaseerOne.sol b/src/MaseerOne.sol
  index 6646518..2b93143 100644
  --- a/src/MaseerOne.sol
  +++ b/src/MaseerOne.sol
  @@ -382,7 +382,7 @@ contract MaseerOne is MaseerToken {
        }

        function _wmul(uint256 x, uint256 y) internal pure returns
(uint256 z) {
  -         z = ((x * y) + (WAD / 2)) / WAD;
  +         z = (x * y) / WAD;
        }
```

```
        function _wdiv(uint256 x, uint256 y) internal pure returns
(uint256 z) {
```

Auditor Notes

The rounding issues were discovered through invariant testing and detailed analysis of redemption operations. The test case demonstrates a walk-through of how to chase the problem out and resolve it. The recommended changes align with standard mathematical rounding practices and ensure protocol invariants hold true.

Client Response

Fixed.

Recognized as low impact in the scale and relative costs to exploit this, it's nonetheless an accounting bug that warranted a fix. The intention here was to round up the calculated prices for both the `mintcost` and the `burncost` of the asset in MaseerGate in terms of the underlying asset. The supplied logic added 1 wei of USDT per unit priced, which became immediately obvious with a unit test demonstrating a `burncost` calculation of `bpsout == 0`. A more robust `divup` function was added to perform this calculation as expected and only round up when there was a remainder.
The aggregate sale calculation in `MaseerOne` also included a function to round up to the nearest wei on the total sale of the mint or burn. This has been corrected and in both cases now round down the bulk total.
A number of tests have been added to demonstrate the expected behavior using high and low prices, as well as small and no `bps` adjustments. In practical applications, this would have less noticeable implications as the token price has 6 decimal places of precision, but it has been fixed to handle all circumstances.

To summarize the expected behavior:

MasserGate:

- Round up unit mint cost (user pays remainder on bps adjustment)
- Round up unit burn cost (Maseer loses remainder on bps adjustment)
  MaseerOne:
- Round down on aggregate mint output. (Protocol trims remainder of output tokens)
- Round down on aggregate burn output. (Protocol trims remainder of required payment)

Full implementation of fix and tests were added at https://github.com/Maseer-LTD/maseer-one/pull/9

9. Medium Risks

9.1 Oracle Authorization System in MaseerPrice Creates Unnecessary Trust Assumptions

Context

MaseerPrice.sol#L26-L28

Description

The `MaseerPrice` contract's authorization system creates unnecessary trust assumptions by allowing any authorized oracle to perform administrative functions. The `poke()` function, which updates the price feed, uses the same `auth` modifier as administrative functions like `rely()`, `deny()`, and `file()`:

```
Unset
function poke(uint256 read_) public auth {
    _poke(read_);
}

function file(bytes32 what, bytes32 data) external auth {
    if      (what == "price")    _poke(uint256(data));
    else if (what == "name")     _setVal(_NAME_SLOT, data);
    else if (what == "decimals" && uint256(data) <= 18)
_setVal(_DECIMALS_SLOT, data);
    else    revert UnrecognizedParam(data);
    emit    File(what, data);
}
```

This design has several significant risks:

1. **Unnecessary Trust Transfer**: The multisig's trust in an oracle's price feed accuracy is implicitly extended to trust in administrative functions. There is no technical reason why an oracle needs the ability to modify authorization or contract parameters.
2. **Operational Risk**: Over time, the constraint that oracles should not use administrative functions may be forgotten, especially as new team members join or operational pressures increase.

3. **Technical Debt**: The lack of separation between price updates and administrative functions creates technical debt that could lead to security issues in future updates or integrations.
4. **Critical Failure Mode**: In the worst case, an oracle could:
    - Deny the multisig's authorization using `deny()`
    - Brick the entire MaseerOne token since the price feed is an immutable constructor argument
    - Manipulate contract parameters through `file()`
5. **Audit Complexity**: The need to audit oracle contracts for proper restraint in using administrative functions adds unnecessary complexity to the security review process.

Recommendation

Implement a separate authorization system for oracles using a `bud` mapping and dedicated modifiers:

Unset

```
contract MaseerPrice is MaseerImplementation {
    mapping(address => uint256) public bud;

    error NotOracle(address);

    modifier fren() {
        if (bud[msg.sender] != 1) revert NotOracle(msg.sender);
        _;
    }

    function kiss(address usr) external auth {
        bud[usr] = 1;
    }

    function diss(address usr) external auth {
        bud[usr] = 0;
    }

    function poke(uint256 read_) public fren {
        _poke(read_);
    }
}
```

This change would:

1. Separate oracle authorization from administrative functions
2. Prevent oracles from performing administrative actions
3. Maintain clear trust boundaries
4. Simplify oracle contract auditing requirements
5. Prevent potential bricking of the MaseerOne token

Suggested Code Changes

Unset

```
diff --git a/src/MaseerPrice.sol b/src/MaseerPrice.sol
index 4fc6364..c0f33af 100644
--- a/src/MaseerPrice.sol
+++ b/src/MaseerPrice.sol
@@ -20,6 +20,8 @@ contract MaseerPrice is MaseerImplementation{
     bytes32 internal constant _NAME_SLOT     =
keccak256("maseer.price.name");
     bytes32 internal constant _PRICE_SLOT    =
keccak256("maseer.price.price");
     bytes32 internal constant _DECIMALS_SLOT =
keccak256("maseer.price.decimals");
+
+    mapping(address => uint256) public bud;

     // Allocate slots 0-49
     uint256[50] private __gap;
@@ -28,6 +30,12 @@ contract MaseerPrice is MaseerImplementation{
     event File(bytes32 indexed what, bytes32 data);
     event PriceUpdate(uint256 indexed price, uint256 indexed
timestamp);

+    error NotOracle(address);
+
+    modifier fren() {
+        if (bud[msg.sender] != 1) revert NotOracle(msg.sender);
+        _;
+    }
```

```
+
     constructor() {
         _rely(msg.sender);
     }
@@ -52,7 +60,7 @@ contract MaseerPrice is MaseerImplementation{
         return (_uint256Slot(_PRICE_SLOT) == 0) ? true : false;
     }

-    function poke(uint256 read_) public auth {
+    function poke(uint256 read_) public fren {
         _poke(read_);
     }

@@ -60,6 +68,14 @@ contract MaseerPrice is MaseerImplementation{
         else if (what == "name")     _setVal(_NAME_SLOT, data);
         else if (what == "decimals" && uint256(data) <= 18)
_setVal(_DECIMALS_SLOT, data);
         else    revert UnrecognizedParam(data);
         emit    File(what, data);
     }
+
+    function kiss(address usr) external auth {
+        bud[usr] = 1;
+    }
+
+    function diss(address usr) external auth {
+        bud[usr] = 0;
+    }
```

Auditor Notes

This issue was discovered during our review of the authorization system in MaseerPrice. The current design creates unnecessary trust assumptions that could lead to critical failures in the protocol. The separation of oracle authorization from administrative functions is a fundamental security improvement that should be implemented before deployment.

Client Response

Fixed.

The assumption was that new permissioned modules would be scoped to the relevant function, however this suggestion allows for additional flexibility of authorization and ensures that access to these values can be controlled more granularly.

Implemented at https://github.com/Maseer-LTD/maseer-one/pull/5

10. Low Risks

10.1 Issuer's Ability to Burn Tokens from Blacklisted Users Creates Authorization Model Inconsistency

Context

MaseerOne.sol#L252-L264

Description

The `MaseerOne.smelt(address usr, uint256 amt)` function allows issuers to burn tokens from any user's account without checking if that user is blacklisted. This creates an inconsistency with the rest of the protocol's authorization model, where blacklisted users are generally prevented from any token interactions.

The current implementation:

```
Unset
function smelt(address usr, uint256 amt) external issuer
pass(msg.sender) {
    _burn(usr, amt);
    emit Smelted(usr, amt);
}
```

While the issuer is checked against `pass()`, the target user (`usr`) is not. This differs from other functions in the protocol that interact with user balances, such as `transferFrom`:

```
Unset
    function transferFrom(address src, address dst, uint256 wad)
public override pass(msg.sender) pass(src) pass(dst) returns
(bool) {
        if (dst == address(this)) revert TransferToContract();
        return super.transferFrom(src, dst, wad);
    }
```

This inconsistency in the authorization model could lead to:

1. Confusion about the intended behavior of blacklisted users
2. Potential security issues if other functions are implemented without considering this precedent
3. Difficulty in maintaining consistent authorization checks across the protocol
4. Challenges in auditing the protocol's authorization model

Recommendation

Add the `pass(usr)` modifier to the issuer version of `smelt()` to maintain consistency with the protocol's authorization model:

```
Unset
function smelt(address usr, uint256 amt) external issuer
pass(msg.sender) pass(usr) {
    _burn(usr, amt);
    emit Smelted(usr, amt);
}
```

However, if this is intended business logic to allow issuers to remove tokens from blacklisted accounts, we recommend:

1. Documenting this capability clearly in the protocol documentation
2. Establishing clear governance procedures for when this power can be exercised
3. Implementing additional safeguards such as:
    - Time delays on forced burns
    - Multi-signature requirements
    - Governance votes for specific burns
    - Maximum burn amounts per transaction
4. Adding events that clearly indicate when tokens are burned from blacklisted accounts

Suggested Code Changes

```
Unset
diff --git a/src/MaseerOne.sol b/src/MaseerOne.sol
index 4fc6364..c0f33af 100644
--- a/src/MaseerOne.sol
+++ b/src/MaseerOne.sol
@@ -89,6 +89,7 @@ contract MaseerOne is ERC20,
MaseerImplementation {
     // Issuer smelting function to burn for a specific user
-    function smelt(address usr, uint256 amt) external issuer
pass(msg.sender) {
+    function smelt(address usr, uint256 amt) external issuer
pass(msg.sender) pass(usr) {
         _burn(usr, amt);
         emit Smelted(usr, amt);
     }
```

Auditor Notes

This issue highlights an inconsistency in the protocol's authorization model. While the ability to burn tokens from blacklisted accounts might be intended functionality, it creates confusion about the intended behavior of blacklisted users and makes it more difficult to maintain consistent authorization checks across the protocol. The protocol team should carefully consider whether this capability aligns with their authorization model and implement appropriate safeguards if it is to be maintained.

Client Response

Acknowledged. `issuer` is a designated treasury address and the intended usage is to burn balances in blacklisted accounts similar to the Tether model.

10.2 Blacklisted Users Can Execute Precommitments, Bypassing Authorization Model

Context

MaseerPrecommit.sol#L58-L66

Description

The `MaseerPrecommit.exec()` function allows blacklisted users to execute precommitments on behalf of other users, which is inconsistent with `MaseerOne`'s authorization model. In `MaseerOne`, blacklisted users cannot perform `transferFrom` operations, but `MaseerPrecommit.exec()` does not enforce this restriction on the caller.

The current implementation only checks if the target user (`_usr`) is blacklisted:

```
Unset
function exec(uint256 idx) external {
    (address usr, uint256 amt) = deal(idx);
    // ... other checks ...
    one.mint(usr, amt);  // This internally calls transferFrom
}
```

However, `MaseerOne.transferFrom()` enforces blacklist checks on both the caller and the target:

```
Unset
    function transferFrom(address src, address dst, uint256 wad)
public override pass(msg.sender) pass(src) pass(dst) returns
(bool) {
        if (dst == address(this)) revert TransferToContract();
        return super.transferFrom(src, dst, wad);
    }
```

This inconsistency creates a business logic bug where blacklisted users can still move funds through the precommit system, bypassing the intended restrictions.

Recommendation

Add a `canPass()` check on `msg.sender` in `MaseerPrecommit.exec()`, similar to how it's done in `pact()`:

```
Unset
function exec(uint256 idx) external {
    if (!One(one).canPass(msg.sender)) revert NotAuthorized();
```

```
    (address usr, uint256 amt) = deal(idx);
    // ... rest of function ...
}
```

This change would:

1. Maintain consistency with `MaseerOne`'s authorization model
2. Prevent blacklisted users from executing precommitments
3. Align with the principle that blacklisted users should not be able to move funds
4. Fix the business logic inconsistency

Suggested Code Changes

Unset

```
diff --git a/src/MaseerPrecommit.sol b/src/MaseerPrecommit.sol
index 4fc6364..c0f33af 100644
--- a/src/MaseerPrecommit.sol
+++ b/src/MaseerPrecommit.sol
@@ -89,6 +89,7 @@ contract MaseerPrecommit is
MaseerImplementation {
     function exec(uint256 idx) external {
         (address usr, uint256 amt) = deal(idx);
+        if (!One(one).canPass(msg.sender)) revert
NotAuthorized();
         require(amt > 0, "InsufficientAmount()");
         require(one.mintable(), "MarketClosed()");
         uint256 unit = one.mintcost();
```

Auditor Notes

This issue was discovered during our review of the authorization system in MaseerPrecommit. The inconsistency between MaseerPrecommit and MaseerOne's authorization models creates a potential security risk where blacklisted users can still move funds through the precommit system. The fix is straightforward and maintains consistency with the rest of the protocol's authorization model.

Client Response

Fixed.

Blacklisted users do not touch funds in this scenario but we don't want them acting in the system. Added a pass modifier to `pact` and `exec` here https://github.com/Maseer-LTD/maseer-one/pull/6

10.3 Blacklisted Users Can Exit Precommitments, Bypassing Authorization Model

Context

MaseerPrecommit.sol#L68-L70

Description

The `MaseerPrecommit.exit()` function allows blacklisted users to exit their precommitments, which is inconsistent with `MaseerOne`'s authorization model. In `MaseerOne`, blacklisted users cannot perform `transferFrom` operations, but `MaseerPrecommit.exit()` does not enforce this restriction on the caller.

The current implementation only checks if the contract addresses are blacklisted:

```
Unset
function exit() external {
    // ... other checks ...
    gem.transferFrom(address(this), msg.sender, amt);  // This
internally calls transferFrom
}
```

However, `MaseerOne.transferFrom()` enforces blacklist checks on both the caller and the target:

```
Unset
    function transferFrom(address src, address dst, uint256 wad)
public override pass(msg.sender) pass(src) pass(dst) returns
(bool) {
```

```
        if (dst == address(this)) revert TransferToContract();
        return super.transferFrom(src, dst, wad);
    }
```

This inconsistency creates a business logic bug where blacklisted users can still move funds through the precommit system by exiting their precommitments, bypassing the intended restrictions.

Recommendation

Add a `canPass()` check on `msg.sender` in `MaseerPrecommit.exit()`, similar to how it's done in `pact()`:

Unset
```
function exit() external {
    if (!One(one).canPass(msg.sender)) revert NotAuthorized();
    // ... rest of function ...
}
```

This change would:

1. Maintain consistency with `MaseerOne`'s authorization model
2. Prevent blacklisted users from exiting precommitments
3. Align with the principle that blacklisted users should not be able to move funds
4. Fix the business logic inconsistency

Suggested Code Changes

Unset
```
diff --git a/src/MaseerPrecommit.sol b/src/MaseerPrecommit.sol
index 4fc6364..c0f33af 100644
--- a/src/MaseerPrecommit.sol
+++ b/src/MaseerPrecommit.sol
@@ -89,6 +89,7 @@ contract MaseerPrecommit is
MaseerImplementation {
```

```
     function exit() external {
+        if (!One(one).canPass(msg.sender)) revert
NotAuthorized();
         uint256 amt = gem.balanceOf(address(this));
         if (amt > 0) {
             gem.transferFrom(address(this), msg.sender, amt);
```

Auditor Notes

This issue was discovered during our review of the authorization system in MaseerPrecommit. The inconsistency between MaseerPrecommit and MaseerOne's authorization models creates a potential security risk where blacklisted users can still move funds through the precommit system by exiting their precommitments. The fix is straightforward and maintains consistency with the rest of the protocol's authorization model.

Client Response

Fixed.

Blacklisted users do not custody funds in this scenario but we don't want them acting in the system. Added a pass modifier to `exit` here https://github.com/Maseer-LTD/maseer-one/pull/7

10.4 Blacklisted Users Can Move Funds Through Conduit, Bypassing Authorization Model

Context

MaseerConduit.sol#L50-L58

Description

The `MaseerConduit.move()` functions allow blacklisted users to move funds through the conduit, which is inconsistent with `MaseerOne`'s authorization model. In `MaseerOne`, blacklisted users cannot perform `transferFrom` operations, but `MaseerConduit.move()` does not enforce this restriction on either the caller or the target address.

The current implementation only checks for operator and bud permissions:

```
Unset
function move(address _token, address _to) external operator
buds(_to) returns (uint256 _amt) {
    _amt = Gem(_token).balanceOf(address(this));
    _move(_token, _to, _amt);
}
function move(address _token, address _to, uint256 _amt) external
operator buds(_to) returns (uint256) {
    _move(_token, _to, _amt);
    return _amt;
}
```

However, `MaseerOne.transferFrom()` enforces blacklist checks on both the caller and the target:

```
Unset
function transferFrom(address src, address dst, uint256 wad)
public override pass(msg.sender) pass(src) pass(dst) returns
(bool) {
    if (dst == address(this)) revert TransferToContract();
    return super.transferFrom(src, dst, wad);
}
```

This inconsistency creates a business logic bug where blacklisted users can still move funds through the conduit system, bypassing the intended restrictions.

Recommendation

1. Add `MaseerGuard` as a constructor argument to `MaseerConduit`
2. Add `pass()` modifier checks to both `move()` functions:

```
Unset
contract MaseerConduit is MaseerImplementation {
    MaseerGuard public immutable guard;
    constructor(address _guard) {
```

```
        guard = MaseerGuard(_guard);
        _rely(msg.sender);
    }

    function move(address _token, address _to) external operator
buds(_to) pass(msg.sender) pass(_to) returns (uint256 _amt) {
        _amt = Gem(_token).balanceOf(address(this));
        _move(_token, _to, _amt);
    }

    function move(address _token, address _to, uint256 _amt)
external operator buds(_to) pass(msg.sender) pass(_to) returns
(uint256) {
        _move(_token, _to, _amt);
        return _amt;
    }
}
```

This change would:

1. Maintain consistency with `MaseerOne`'s authorization model
2. Prevent blacklisted users from moving funds through the conduit
3. Prevent blacklisted users from receiving funds through the conduit
4. Align with the principle that blacklisted users should not be able to move funds
5. Fix the business logic inconsistency

Suggested Code Changes

Unset

```
diff --git a/src/MaseerConduit.sol b/src/MaseerConduit.sol
index 4fc6364..c0f33af 100644
--- a/src/MaseerConduit.sol
+++ b/src/MaseerConduit.sol
@@ -20,6 +20,8 @@ contract MaseerConduit is MaseerImplementation
{
```

```
     // Slot 2
     mapping (address => uint256) public bud;
     // Allocate Slots 3-49
+    MaseerGuard public immutable guard;
+
     uint256[48] private __gap;

     error ZeroAddress();
@@ -28,7 +30,7 @@ contract MaseerConduit is MaseerImplementation
{
     error NotOperator(address);

     function hope(address usr) external auth { can[usr] = 1; }
-    constructor() {
+    constructor(address _guard) {
+        guard = MaseerGuard(_guard);
         _rely(msg.sender);
     }
@@ -89,7 +91,7 @@ contract MaseerConduit is MaseerImplementation
{
-    function move(address _token, address _to) external operator
buds(_to) returns (uint256 _amt) {
+    function move(address _token, address _to) external operator
buds(_to) pass(msg.sender) pass(_to) returns (uint256 _amt) {
         _amt = Gem(_token).balanceOf(address(this));
         _move(_token, _to, _amt);
     }
@@ -97,7 +99,7 @@ contract MaseerConduit is MaseerImplementation
{
-    function move(address _token, address _to, uint256 _amt)
external operator buds(_to) returns (uint256) {
+    function move(address _token, address _to, uint256 _amt)
external operator buds(_to) pass(msg.sender) pass(_to) returns
(uint256) {
         _move(_token, _to, _amt);
         return _amt;
```

```
    }
```

Auditor Notes

This issue was discovered during our review of the authorization system in MaseerConduit. The inconsistency between MaseerConduit and MaseerOne's authorization models creates a potential security risk where blacklisted users can still move funds through the conduit system. The fix requires adding the MaseerGuard dependency and implementing proper pass() checks to maintain consistency with the rest of the protocol's authorization model.

Client Response

Acknowledged.

These functions already contain the `operator` modifier, which enforces a whitelisted known associate who is authorized to move the funds. The additional check to ensure the whitelisted actor is not on the external blacklist is therefore unnecessary here.

10.5 Chain ID Replay Attack Vulnerability in Permit System

Context

MaseerToken.sol#L84

Description

The current implementation has a severe flaw in its permit signature verification system. While it attempts to prevent permit replay attacks by checking the chain ID, this approach is insufficient because:

1. When a chain is forked, users on the forked chain can:
    - Build a permit signature using the forked chain's `one.DOMAIN_SEPARATOR()`
    - Publish this signed message on the forked chain
    - The permit will fail on the forked chain due to the chain ID check
    - However, the signed message remains valid and can be replayed on the main chain
2. This creates a replay attack vector where:
    - An attacker can trick a user into signing a permit message on a forked chain
    - The signature, while invalid on the fork, is cryptographically valid
    - The attacker can then replay this signature on the main chain
    - The user's funds can be moved without their consent on the main chain

3. The current implementation's chain ID check is ineffective because:
    - It only prevents the permit from executing on the forked chain
    - It does not prevent the creation of valid signatures that can be replayed
    - The signature verification process is not properly tied to the chain ID

Recommendation

Implement a proper domain separator system in `MaseerToken.sol` that:

1. Stores the initial chain ID and domain separator at contract deployment
2. Uses a ternary operator in `DOMAIN_SEPARATOR()` to recompute the separator when the chain ID changes
3. Removes the chain ID check from the `permit()` function, allowing Maseer to decide if permits should work on forks
4. Ensures signatures are properly tied to their specific chain through the domain separator

This approach will prevent signature replay attacks by ensuring that signatures created on one chain cannot be replayed on another, regardless of whether permits are allowed on forks.

Suggested Code Changes

Unset

```
diff --git a/src/MaseerToken.sol b/src/MaseerToken.sol
index 663431f..80b85ab 100644
--- a/src/MaseerToken.sol
+++ b/src/MaseerToken.sol
@@ -14,16 +14,16 @@ abstract contract MaseerToken {
     uint256                                   constant internal
WAD = 1e18;

     // EIP-2612
-    bytes32 immutable           public DOMAIN_SEPARATOR;
+    bytes32 internal immutable INITIAL_DOMAIN_SEPARATOR;
+    uint256 internal immutable INITIAL_CHAIN_ID;
     mapping(address => uint256) public nonces;
-    uint256 immutable         internal CHAIN_ID; // token is
linked to deployment chain id. Forked chains are not supported.

     constructor(string memory name_, string memory symbol_) {
         name   = name_;
```

```
         symbol = symbol_;

-        CHAIN_ID = block.chainid;
-        DOMAIN_SEPARATOR = _domainSeparator();
+        INITIAL_CHAIN_ID = block.chainid;
+        INITIAL_DOMAIN_SEPARATOR = computeDomainSeparator();
     }

     event Approval(address indexed src, address indexed usr,
uint256 wad);
@@ -81,7 +81,6 @@ abstract contract MaseerToken {
         bytes32 r,
         bytes32 s
     ) public virtual {
-        require(block.chainid == CHAIN_ID,   "INVALID_CHAIN");
         require(deadline >= block.timestamp,
"PERMIT_DEADLINE_EXPIRED");

         // Unchecked because the owner's nonce is uint256 which
cannot realistically overflow.
@@ -90,7 +89,7 @@ abstract contract MaseerToken {
                 keccak256(
                     abi.encodePacked(
                         "\x19\x01",
-                        DOMAIN_SEPARATOR,
+                        DOMAIN_SEPARATOR(),
                         keccak256(
                             abi.encode(
                                 keccak256(
@@ -136,12 +135,16 @@ abstract contract MaseerToken {
         emit Transfer(usr, address(0), wad);
     }

-    function _domainSeparator() internal view returns (bytes32)
{
```

```
+    function DOMAIN_SEPARATOR() public view virtual returns
(bytes32) {
+        return block.chainid == INITIAL_CHAIN_ID ?
INITIAL_DOMAIN_SEPARATOR : computeDomainSeparator();
+    }
+
+    function computeDomainSeparator() internal view virtual
returns (bytes32) {
         return keccak256(abi.encode(
             keccak256("EIP712Domain(string name,string
version,uint256 chainId,address verifyingContract)"),
             keccak256(bytes(name)),
             keccak256(bytes("1")),
-            CHAIN_ID,
+            block.chainid,
             address(this)
         ));
     }
```

Auditor Notes

This issue was discovered during our review of the invariant testing framework. The current implementation's approach to chain ID verification is fundamentally flawed and creates a significant security risk. The recommended solution follows the established pattern used in battle-tested implementations like Solmate's ERC20 contract, which properly handles chain changes through the domain separator system.

Client Response

Fixed.

The code here was intended to brick permit transactions on forked chains, but the unintended side-effect of this was that it enabled permit transactions signed on those chains to be replayed on mainnet.

This modification has been implemented at https://github.com/Maseer-LTD/maseer-one/pull/8

Users will now be able to issue permit transactions on forked chains that will not be replayable on the deployment chain.

10.6 Zero Claim Redemptions Create Poor User Experience

Context

MaseerOne.sol#L172-L194

Description

The protocol allows redemptions that result in zero claims, which violates a key invariant and creates poor user experience.

When a user attempts to redeem tokens, the claim amount may be zero due to either a small redemption amount or a low price valuation. In such cases:

- The MaseerOne tokens are burned
- The user receives no gem tokens
- The redemption is recorded with a zero amount

This creates a poor user experience as users may accidentally burn their tokens without receiving anything in return. Worse, malicious UIs could grief users by encouraging these zero-claim redemptions. Given Maseer's position as a company, such situations would likely result in requests for multisig intervention to rectify the problem.

Note: This missing functionality is the mirror of the `DustThreshold()` check in `mint()`.

1. **Invariant Violation**
    - The following invariant fails to hold:

```
Unset
function invariant_MO_redemptionDateAmountRelations() public leap {
    uint256 _count = one.redemptionCount();
    for (uint256 i = 0; i < _count; i++) {
        uint256 _amount = one.redemptionAmount(i);
        uint256 _date = one.redemptionDate(i);
        uint256 _now = block.timestamp;

        // If amount is 0, the date must have passed
        if (_amount == 0) {
            require(
                _date <= _now,
                "Redemption Invariant
MO_redemptionDateAmountRelations_zeroAmount"
```

```
            );
        }
    }
}
```

2. **Root Cause**
    - Zero claims can occur due to:
        - Small redemption amounts
        - Low price valuations
        - Rounding in the claim calculation
    - When this happens:
        - MaseerOne tokens are burned
        - User receives no gem tokens
        - Redemption is recorded with zero amount
3. **Impact**
    - Poor user experience
    - Potential for user error
    - Risk of UI griefing
    - Likely to result in support requests
    - May require multisig intervention to rectify

Recommendation

Add a check in the `redeem()` function to prevent zero claims:

```
Unset
// Calculate the redemption amount
uint256 _claim = _wmul(amt, _unit);

// don't allow 0 claims
if (_claim == 0) {
    revert ClaimIsZero();
}
```

This change would:

1. Prevent zero claim redemptions
2. Improve user experience
3. Reduce support burden
4. Maintain protocol invariants
5. Prevent potential griefing

Suggested Code Changes

Unset

```
diff --git a/src/MaseerOne.sol b/src/MaseerOne.sol
index 6646518..ab53c6c 100644
--- a/src/MaseerOne.sol
+++ b/src/MaseerOne.sol
@@ -88,6 +88,7 @@ contract MaseerOne is MaseerToken {
     error ClaimableAfter(uint256 time);
     error NoPendingClaim();
     error DustThreshold(uint256 min);
+    error ClaimIsZero();

     constructor(
         address gem_,
@@ -171,6 +172,11 @@ contract MaseerOne is MaseerToken {
         // Calculate the redemption amount
         uint256 _claim = _wmul(amt, _unit);

+        // don't allow 0 claims
+        if (_claim == 0) {
+            revert ClaimIsZero();
+        }
+
         // Add to the total pending redemptions
         totalPending += _claim;
```

Auditor Notes

This issue was discovered during invariant testing. While not a security vulnerability unless triggered by a griefing UI, it represents a significant user experience concern that could lead to support issues and potential reputational damage. The fix is straightforward and aligns with the protocol's goal of providing a robust and user-friendly experience.

Client Response

Fixed at https://github.com/Maseer-LTD/maseer-one/pull/10

The suggested fix is good, and would protect users from this experience, however after internal discussion we've decided to up the minimum redemption amount to one full token unit (i.e. `1e18`) of the MaseerOne token. This better conceptually aligns with the idea of selling a minimum of one off-chain unit and limits the ability of the user to sell small amounts that would result in a `0` token payout.

The resulting code has been added to the `redeem` function:

```
Unset
        // Assert minimum redemption amount of one token to avoid dust

        if (amt < WAD) revert DustThreshold(WAD);
```

This does create an interesting scenario during a wind-down situation, where a user may not be able to accumulate enough of the token to redeem via this process, however this can be resolved internally via re-issuance of remaining book assets, or defi market operations to purchase and burn the tokens.

10.7 Ambiguous Oracle Decimals and Price Normalization

Context

MaseerOne.sol#L384-L386
MaseerPrice.sol#L22-L24

Description

The MaseerPrice contract's decimals functionality is ambiguous and disconnected from the protocol's actual price handling:

1. **Decimals Disconnection**
    - MaseerPrice allows setting decimals up to 18 via `file()`
    - These decimals are not linked to the gem's decimals in MaseerOne
    - The decimals value is unused in price calculations
    - Oracles are trusted to provide prices matching gem decimals without enforcement
2. **Inconsistent Usage**
    - Deploy scripts set MaseerPrice decimals to 6
    - Test suite never sets decimals
    - Creates confusion and missed opportunities for precision handling
3. **Impact on Rounding**
    - Investigation revealed this could affect price precision
    - Current implementation may amplify rounding issues from issue 8.1
    - Could allow extraction of more value than intended in redemptions

The MaseerOracle allows setting decimals with the `file()` function, which ensures decimals do not exceed 18. However, these decimals are not linked to the gem's `decimals()` in MaseerOne, which handles exchanges between gem tokens and MaseerOne tokens. Furthermore, these decimals are

not connected to published prices. In fact, `pip.decimals()` is unused in the rest of the protocol, and oracles are simply trusted to provide price updates matching the gem decimals.

The codebase shows mixed intentions - deploy scripts set MaseerPrice decimals to 6, while the test suite never sets decimals. The result is that decimals serve only as a suggestion without actual enforcement, leading to potential confusion and missed opportunities regarding issue 8.1.

Our investigation of this issue began with issue10.6, where `amount * burncost` (where `burncost` is `price * output fee`) indicated that certain amounts and prices in redemption could be too small to produce a `claim` amount. Additionally, the rounding issues in issue 8.1 could be mitigated if the `wmul()` calculation occurred with higher precision. This higher precision could be achieved by normalizing prices to WAD precision. To test this theory, we created several tests that can be patched into `MaseerOne.operations.t.sol`:

```
Unset
diff --git a/src/MaseerOne.sol b/src/MaseerOne.sol
index 6646518..989380b 100644
--- a/src/MaseerOne.sol
+++ b/src/MaseerOne.sol
@@ -13,6 +13,7 @@ interface Cop {

 interface Pip {
     function read() external view returns (uint256);
+    function decimals() external view returns (uint8);
 }

 interface Act {
@@ -381,8 +382,18 @@ contract MaseerOne is MaseerToken {
         return Gem(gem).balanceOf(address(this));
     }

-    function _wmul(uint256 x, uint256 y) internal pure returns
(uint256 z) {
-        z = ((x * y) + (WAD / 2)) / WAD;
+    function _wmul(uint256 x, uint256 y) internal view returns
(uint256 z) {
+        // existing oracle price using existing _wmul logic
+        // z = ((x * y) + (WAD / 2)) / WAD;
+
```

```
+        // normalized oracle price using exiting _wmul logic
+        z = ((x * y) + (WAD / 2)) / (WAD * (10 ** (18 -
Pip(pip).decimals())));
+
+        // existing oracle price using our suggested _wmul
+        // z = (x * y) / WAD;
+
+        // normalized oracle price using our suggested _wmul
+        // z = (x * y) / (WAD * (10 ** (18 -
Pip(pip).decimals())));
     }

     function _wdiv(uint256 x, uint256 y) internal pure returns
(uint256 z) {
diff --git a/src/MaseerPrice.sol b/src/MaseerPrice.sol
index 4fc6364..88317a1 100644
--- a/src/MaseerPrice.sol
+++ b/src/MaseerPrice.sol
@@ -20,9 +20,19 @@ contract MaseerPrice is MaseerImplementation{
     }

     function read() external view returns (uint256) {
-        return _uint256Slot(_PRICE_SLOT);
+        // To reduce gas costs for oracles, we normalize the
price to 18 decimals here
+        uint256 _price = _uint256Slot(_PRICE_SLOT);
+        uint256 _decimals = _uint256Slot(_DECIMALS_SLOT);
+        if (_price > type(uint256).max / 1e18) {
+            _price = _price / 1e18;
+        }
+        return _price * (10 ** (18 - _decimals));
     }

+    // function read() external view returns (uint256) {
+    //     return _uint256Slot(_PRICE_SLOT);
+    // }
```

```
+
     function poke(uint256 read_) public auth {
         _poke(read_);
     }
diff --git a/test/MaseerOne.operations.t.sol
b/test/MaseerOne.operations.t.sol
index 0505bd4..045aeaa 100644
--- a/test/MaseerOne.operations.t.sol
+++ b/test/MaseerOne.operations.t.sol
@@ -298,4 +298,178 @@ contract MaseerOneOperationsTest is
MaseerTestBase {
         maseerOne.smelt(bob, 100 * 1e18);
     }

+    // In this test, we expect to redeem 0.000000000000067896
tokens worth 0.000018 USDT per token
+    // we expect to get $0.000000_000000000001222128 USDT
+    //                           ^ 6 decimal places
+
+    // This is what we expect to get
+    /*
+        >>> scale=45
+        >>> (0000000000000067896 * 0000018) / 10^18
+        .000000000001222128000000000000000000000000
+    */
+
+    // This is what not normalizing the oracle price gives us
+    // NOTE: anything beyond the decimal point is lost do to WAD
division
+    /*
+        >>> scale=45
+        >>> ((67896 * 19) + (10^18 / 2)) / 10^18
+        .500000000001290024000000000000000000000000
+    */
+
+    // This is what normalizing the oracle price gives us
```

```
+    // NOTE: anything beyond the decimal point is lost do to WAD
division
+    /*
+        >>> scale=45
+        >>> (((67896 * 18000000000001) + (10^18 / 2)) / (10^18 *
(10^(18 - 6))))
+        .000000000001722128000000067896000000000000000
+    */
+    function testRedeemWithPrice01() public {
+        // Setup: Set out fee to 0
+        vm.prank(actAuth);
+        act.setBpsout(0);
+
+        // Setup: Issue tokens to alice
+        vm.prank(issuer);
+        maseerOne.issue(100_000 * 1e18);
+        vm.prank(issuer);
+        maseerOne.transfer(alice, 100_000 * 1e18);
+        assertEq(maseerOne.balanceOf(alice), 100_000 * 1e18);
+
+        // none of the tests set decimals aside from testing
decimals
+        vm.prank(pipAuth);
+        pip.file("decimals", bytes32(uint256(6))); // USDT price
has 6 decimals
+        assertEq(pip.decimals(), 6);
+
+        // Set price to 18
+        vm.prank(pipAuth);
+        pip.poke(18); // 18 "wei" of USDT per token
+
+        // Attempt redemption of 67896 tokens
+        vm.prank(alice);
+        uint256 _id = maseerOne.redeem(67896);
+
+        // Verify redemption details
```

```
+        assertEq(maseerOne.redemptionAmount(_id), 0); //
0.000000000000000001222128 USDT
+        assertEq(maseerOne.redemptionAddr(_id), alice);
+        assertEq(maseerOne.redemptionDate(_id), block.timestamp
+ maseerOne.cooldown());
+        assertEq(maseerOne.totalPending(), 0);
+    }
+
+
+    // In this test, we expect to redeem 0.000000000000067896
tokens worth 20000.000000 USDT per token
+    // we expect to get $1,357.920000_000000000000000000000000
USDT
+    //                                ^ 6 decimal places
+
+    // This is what we expect to get
+    /*
+        >>> scale=45
+        >>> (000000000000067896 * 20000000000) / 10^18
+        .001357920000000000000000000000000000000000000
+    */
+
+    // This is what not normalizing the oracle price gives us
+    // NOTE: anything beyond the decimal point is lost do to WAD
division
+    /*
+        >>> scale=45
+        >>> ((67896 * 20000000001) + (10^18 / 2)) / 10^18
+        .501357920000067896000000000000000000000000000
+    */
+
+    // This is what normalizing the oracle price gives us
+    // NOTE: anything beyond the decimal point is lost do to WAD
division
+    /*
+        >>> scale=45
```

```
+        >>> (((67896 * 2000000000000000000001) + (10^18 / 2)) /
(10^18 * (10^(18 - 6))))
+        .001357920000500000000000067896000000000000000
+    */
+    function testRedeemWithPrice02() public {
+        // Setup: Set out fee to 0
+        vm.prank(actAuth);
+        act.setBpsout(0);
+
+        // Setup: Issue tokens to alice
+        vm.prank(issuer);
+        maseerOne.issue(100_000 * 1e18);
+        vm.prank(issuer);
+        maseerOne.transfer(alice, 100_000 * 1e18);
+        assertEq(maseerOne.balanceOf(alice), 100_000 * 1e18);
+
+        // none of the tests set decimals aside from testing
decimals
+        vm.prank(pipAuth);
+        pip.file("decimals", bytes32(uint256(6))); // USDT price
has 6 decimals
+        assertEq(pip.decimals(), 6);
+
+        // Set price to 18
+        vm.prank(pipAuth);
+        pip.poke(20_000 * 1e6); // 20k USDT per token
+
+        // Attempt redemption of 67896 tokens
+        vm.prank(alice);
+        uint256 _id = maseerOne.redeem(67896);
+
+        // Verify redemption details
+        assertEq(maseerOne.redemptionAmount(_id), 0); //
0.001357920000000000
+        assertEq(maseerOne.redemptionAddr(_id), alice);
```

```
+        assertEq(maseerOne.redemptionDate(_id), block.timestamp
+ maseerOne.cooldown());
+        assertEq(maseerOne.totalPending(), 0);
+    }
+
+    // In this test, we expect to redeem 500.000000000000000000
tokens worth 1200.000000 USDT per token
+    // we expect to get $600,000.000000_00 USDT
+    //                                 ^ 6 decimal places
+
+    // This is what we expect to get
+    /*
+        >>> scale=45
+        >>> (500000000000000000000.0 * 1200000000.0) / 10^18
+
600000000000.000000000000000000000000000000000000000000
+    */
+
+    // This is what not normalizing the oracle price gives us
+    // NOTE: anything beyond the decimal point is lost do to WAD
division
+    /*
+        >>> scale=45
+        >>> ((500000000000000000000 * 1200000001) + (10^18 / 2))
/ 10^18
+
600000000500.500000000000000000000000000000000000000000
+    */
+
+    // This is what normalizing the oracle price gives us
+    // NOTE: anything beyond the decimal point is lost do to WAD
division
+    /*
+        >>> scale=45
+        >>> (((500000000000000000000 * 120000000000000000001) +
(10^18 / 2)) / (10^18 * (10^(18 - 6))))
```

```
+
600000000000.000000000500500000000000000000000000000000000
+    */
+    function testRedeemWithPrice03() public {
+        // Setup: Set out fee to 0
+        vm.prank(actAuth);
+        act.setBpsout(0);
+
+        // Setup: Issue tokens to alice
+        vm.prank(issuer);
+        maseerOne.issue(100_000 * 1e18);
+        vm.prank(issuer);
+        maseerOne.transfer(alice, 100_000 * 1e18);
+        assertEq(maseerOne.balanceOf(alice), 100_000 * 1e18);
+
+        // none of the tests set decimals aside from testing
decimals
+        vm.prank(pipAuth);
+        pip.file("decimals", bytes32(uint256(6))); // USDT price
has 6 decimals
+        assertEq(pip.decimals(), 6);
+
+        // Set price to 18
+        vm.prank(pipAuth);
+        pip.poke(1_200 * 1e6); // 1200 USDT per token
+
+        // Attempt redemption of 67896 tokens
+        vm.prank(alice);
+        uint256 _id = maseerOne.redeem(500 * 1e18);
+
+        // Verify redemption details
+        assertEq(maseerOne.redemptionAmount(_id), 600_000 *
1e6);
+        assertEq(maseerOne.redemptionAddr(_id), alice);
+        assertEq(maseerOne.redemptionDate(_id), block.timestamp
+ maseerOne.cooldown());
```

```
+         assertEq(maseerOne.totalPending(), 600_000 * 1e6);
+     }
  }
```

Note the results in the comments above `testRedeemWithPrice01`. We can see that the math for these two very small values doesn't produce a `claim`, but using `bc` shows that internal precision is more accurate with prices normalized to WAD. While the results are the same (a 0 amount), the normalized version provides a better approximation of the theoretical value.

In `testRedeemWithPrice02`, we increase the price to make each unit of one token (1e18) worth 20k USDT. This still produces a 0 `claim`, but we can see that our expected value matches in almost all significant bits before the rounding from issue 8.1 distorts the result.

Finally, in `testRedeemWithPrice03`, we test a realistic price for the assets. The expected price matches the normalized price in this range, with rounding artifacts appearing far beyond the decimal. However, the non-normalized version allows these rounding artifacts to produce a 500 "wei" USDT discrepancy.

The trend is clear: for larger values, these rounding artifacts can produce real value in favor of the user during redemption. If this can be looped to create an advantage in issue 8.1, users could extract more value for their MaseerTokens than the price would indicate. Since this issue only helps mitigate the rounding artifact and isn't the root cause (serving only as defense-in-depth), we believe this issue warrants no higher than Low severity.

We present these tests for Maseer to experiment with various price normalization strategies, but they also reveal additional problems from issue 8.1. IMPORTANT NOTE: If the suggestions from issue 8.1 are implemented, the computation results (with or without price-normalized oracles) match our expectations from `bc` exactly. This suggests that issue 8.1 is resolved by our suggestion, and any solutions should be tested against these three tests.

Test Results

Three test cases demonstrate the impact:

1. **Small Values (testRedeemWithPrice01)**
    - Price: 18 wei USDT per token
    - Redemption: 0.000000000000067896 tokens
    - Expected: 0.000000000000000001222128 USDT
    - Result: 0 USDT (below precision threshold)
2. **Medium Values (testRedeemWithPrice02)**
    - Price: 20k USDT per token
    - Redemption: 0.000000000000067896 tokens

    - Expected: 0.001357920000000000 USDT
    - Result: 0 USDT (below precision threshold)
3. **Large Values (testRedeemWithPrice03)**
    - Price: 1200 USDT per token
    - Redemption: 500 tokens
    - Expected: 600,000 USDT
    - Result: 600,000 USDT (matches expected)

Recommendation

Four potential solutions:

1. **Status Quo**
    - Keep current ambiguous implementation
    - Risk: Ambiguity remains, and doesn't dampen precision issues
2. **Enforce Decimal Matching**
    - Require `pip.decimals()` to match `gem.decimals()` in the `MaseerOne` constructor
    - Makes intention clear but doesn't dampen precision issues
3. **Remove Decimals**
    - Remove all decimals related functionality from `MaseerPrice`
    - Solves ambiguity without adding complexity, but doesn't dampen precision issues
4. **Normalize to WAD** (Recommended)
    - Normalize all prices to WAD precision
    - Provides defense-in-depth against precision errors
    - Theoretical max price: `type(uint256).max/WAD`

Suggested Code Changes

Unset

```
Note, there are a number of ways to implement this solution. Here are some suggestions. We hacked the decimals into wmul, but would probably suggest a better separation of concerns on a final solution.

Updates to `MaseerOne.sol`:

diff --git a/src/MaseerOne.sol b/src/MaseerOne.sol
index 6646518..989380b 100644
--- a/src/MaseerOne.sol
+++ b/src/MaseerOne.sol
```

```
@@ -13,6 +13,7 @@ interface Cop {

 interface Pip {
     function read() external view returns (uint256);
+    function decimals() external view returns (uint8);
 }

 interface Act {
@@ -381,8 +382,18 @@ contract MaseerOne is MaseerToken {
         return Gem(gem).balanceOf(address(this));
     }

-    function _wmul(uint256 x, uint256 y) internal pure returns
(uint256 z) {
-        z = ((x * y) + (WAD / 2)) / WAD;
+    function _wmul(uint256 x, uint256 y) internal view returns
(uint256 z) {
+        // existing oracle price using existing _wmul logic
+        // z = ((x * y) + (WAD / 2)) / WAD;
+
+        // normalized oracle price using exiting _wmul logic
+        z = ((x * y) + (WAD / 2)) / (WAD * (10 ** (18 -
Pip(pip).decimals())));
+
+        // existing oracle price using our suggested _wmul
+        // z = (x * y) / WAD;
+
+        // normalized oracle price using our suggested _wmul
+        // z = (x * y) / (WAD * (10 ** (18 -
Pip(pip).decimals())));
     }

     function _wdiv(uint256 x, uint256 y) internal pure returns
(uint256 z) {
```

```
Updates to `MaseerPrice.sol`

diff --git a/src/MaseerPrice.sol b/src/MaseerPrice.sol
index 4fc6364..1378edb 100644
--- a/src/MaseerPrice.sol
+++ b/src/MaseerPrice.sol
@@ -20,7 +20,13 @@ contract MaseerPrice is MaseerImplementation{
     }

     function read() external view returns (uint256) {
-        return _uint256Slot(_PRICE_SLOT);
+        // To reduce gas costs for oracles, we normalize the
price to 18 decimals here
+        uint256 _price = _uint256Slot(_PRICE_SLOT);
+        uint256 _decimals = _uint256Slot(_DECIMALS_SLOT);
+        if (_price > type(uint256).max / 1e18) {
+            _price = _price / 1e18;
+        }
+        return _price * (10 ** (18 - _decimals));
     }

     function poke(uint256 read_) public auth {
```

Auditor Notes

The issue was discovered while investigating issue 8.1's rounding problems. While not the root cause, proper price normalization could provide defense-in-depth against precision errors. The recommended solution (option 4) would normalize all prices to WAD precision while maintaining protocol functionality.

Client Response

Acknowledged.

The oracles are intended to provide RWA pricing in the precision of the asset they are being compared against. Since real world prices are expected to be denominated to $0.01 on offchain markets, the precision of oracle prices in USDT terms will already have an additional four decimals

of precision at `0` value. Adding additional decimals of precision here complicates the logic of the MaseerOne contract and cost of interaction with minimal practical benefit to either Maseer or the end user. If a future deployment targets a gem using 18 decimals, a new oracle would be provided to price the asset against that token.

The `name()` and `decimals()` functions are provided as informational for external integrators to aid in parsing the value.

11. Informational Findings

11.1 Multisig Governance Risks and Mitigations

Context

The protocol relies on multisig wallets for privileged access control and critical parameter updates across multiple contracts. This includes:

- MaseerGate.sol: Market timing and fee controls
- MaseerPrice.sol: Oracle price feed management
- MaseerOne.sol: Token issuance and redemption controls

The following contracts are upgradeable through the MaseerProxy pattern:

- MaseerPrice: Oracle price feed implementation
- MaseerGate: Market timing and fee controls implementation
- MaseerTreasury: Issuer control implementation
- MaseerGuard: Compliance feed implementation
- MaseerConduit: Output conduit implementation

This upgradeability means the multisig can fundamentally change how any of these contracts work at any time. This risk becomes increasingly significant as the protocol's Total Value Locked (TVL) grows, as changes could affect larger amounts of user funds and have broader market implications.

Description

While multisig wallets provide a basic layer of security through distributed key management, they introduce several significant risks that must be carefully considered:

1. Identity Verification Challenges

The fundamental limitation of multisig schemes is the inability to definitively verify that each signer represents a unique, independent entity. As noted frequently by CS Bastiat (formerly Chris Blec) and others, there is no reliable way to ensure that multiple signers aren't controlled by the same

individual using different keys. This creates a false sense of security when the actual control is centralized.

2. Coercion and External Pressure

Multisig signers are vulnerable to various forms of coercion:

- Criminal actors may threaten physical harm to signers or their families
- State actors could apply legal pressure or regulatory threats
- Economic pressure through market manipulation or personal financial threats
- The Oasis protocol incident demonstrates how state-level actors can successfully coerce protocol changes through jurisdictional control:
    - UK courts ordered Oasis to use their multisig to exploit a "security vulnerability" in their own protocol
    - This allowed recovery of 120,690 wstETH and 3,213 rETH ($225M) stolen in the Wormhole hack
    - The incident highlights how multisig control can be compelled by legal authorities
    - Reference: UK Court Order Analysis
    - Reference: Jump Counter-Exploit Analysis

3. Signer Collusion

Multisig schemes are inherently vulnerable to signer collusion:

- A subset of signers could coordinate to execute malicious actions
- Economic incentives might encourage signers to collude for personal gain
- The probability of collusion increases with fewer total signers or lower threshold requirements

4. UI/UX Attack Vectors

Recent incidents have highlighted sophisticated UI/UX attacks against multisig schemes:

- The Bybit multisig attack demonstrated how even experienced signers can be tricked into signing malicious transactions through carefully crafted interfaces
- Attackers can manipulate transaction data or context to deceive signers
- Reference: Bybit Multisig Attack Analysis

5. Operational Risks

Multisig schemes face several operational challenges:

- Signers may become unavailable due to personal circumstances, technical issues, or loss of access
- Key management failures can render signers unable to perform their duties
- Lack of proper succession planning can lead to protocol paralysis

- Insufficient incentives may lead to signer apathy or neglect of responsibilities

Recommendation

To address these risks, we recommend implementing a comprehensive governance framework that includes:

1. Reduced Multisig Dependencies

- Implement automated, trustless mechanisms where possible
- Use timelocks and gradual parameter adjustments instead of immediate multisig actions
- Design systems that can operate safely with minimal privileged access

2. TVL-Based Security Thresholds

- Implement progressive security requirements based on protocol TVL
- Require higher multisig thresholds (e.g., 7/10 instead of 5/7) as TVL increases
- Add additional security measures like timelocks for high-TVL changes
- Consider implementing immutable contracts or removing upgradeability for core functions once the protocol reaches certain TVL thresholds

3. Timelock Controls

- Implement a timelock controller for all multisig actions
- Require a minimum delay period (e.g., 24-48 hours) before execution
- Enable market participants to exit positions if they disagree with proposed changes

4. Signer Selection and Distribution

- Mix known and unknown signers to reduce coercion risks
- Implement an N of M threshold scheme with sufficient signers to make collusion statistically unlikely
- Distribute signers across multiple jurisdictions to prevent state-level coercion
- Regular rotation of signers while maintaining institutional knowledge

5. Operational Safeguards

- Regular verification of signer availability and capability
- Clear succession planning and backup signer procedures
- Strong economic incentives for long-term signer participation
- Technical infrastructure for secure key management and signing

6. Damage Limitation

- Implement circuit breakers and emergency pause mechanisms
- Limit the scope of actions any single multisig can execute
- Require multiple multisigs for critical protocol changes
- Implement progressive decentralization where possible

Client Response

Acknowledged.

Maseer has chosen to launch with a 3/5 multisig strategy for initial launch, with internal users abiding industry best practices for key management.

The goal will be to optimize and automate market and oracle operations as the protocol progresses and TVL grows to reduce the reliance on administrative contact and overhead.

11.2 Contract Upgradeability Risks

Context

The protocol's core contracts are upgradeable through the MaseerProxy pattern, allowing their implementation to be changed at any time. Our audit is specifically of the current implementations at commit hash 87c1715700057d95737486a238bd7b4e418507d6:

- MaseerPrice: Oracle price feed implementation
- MaseerGate: Market timing and fee controls implementation
- MaseerTreasury: Issuer control implementation
- MaseerGuard: Compliance feed implementation
- MaseerConduit: Output conduit implementation

This upgradeability means that the behavior of these contracts can be completely changed at any time, potentially affecting:

- Price feed mechanisms and data sources
- Market timing controls and trading windows
- Fee structures and calculations
- Compliance checks and restrictions
- Fund flows and treasury management

Description

The ability to upgrade these contracts introduces several significant risks:

1. Complete Behavior Modification

Any of these contracts can be replaced with new implementations that have entirely different behaviors. For example:

- The price feed could be modified to return manipulated values
- Market timing controls could be changed to lock users out
- Fee structures could be altered to extract more value
- Compliance checks could be bypassed or modified
- Fund flows could be redirected

2. Breaking Changes

Upgrades could introduce breaking changes that:

- Invalidate existing integrations
- Break user assumptions about contract behavior
- Create inconsistencies in the protocol's state
- Lead to unexpected side effects in dependent systems

3. TVL-Dependent Risk

The risk of upgradeability becomes increasingly severe as the protocol's Total Value Locked (TVL) grows:

- Larger amounts of user funds are exposed to potential changes
- Market impact of changes becomes more significant
- Systemic risk to the broader ecosystem increases
- Incentives for malicious upgrades grow

Recommendation

To address these risks, we recommend:

1. TVL-Based Immutability

- Implement a TVL threshold (e.g., $100M) after which contracts become immutable
- Remove upgradeability for core functions once the protocol reaches maturity
- Consider implementing a progressive immutability schedule based on TVL milestones

2. Immediate Mitigation

The multisig can immediately reduce upgradeability risks by:

- Calling `denyProxy()` on each proxy contract to revoke upgrade permissions

- This would prevent future implementation changes while maintaining current functionality
- Reference: MaseerProxy.sol

3. Gradual Decentralization

- Implement a plan to gradually remove upgradeability as the protocol matures
- Start with less critical contracts and move towards core functionality
- Consider implementing timelocks and governance controls for any remaining upgradeable contracts

Client Response

Acknowledged.

We intend to automate and ossify as much of the periphery as possible during growth. In the launch phase, many of these calculations are based on expected interactions with offchain markets and market settings may need to be tweaked during growth. We also will be keeping an eye out for efficiencies and optimizations of all of these dependencies.

The current guard relies on an external source and it may be necessary to provide that information in-house, or as a result of a better solution presenting itself, which will necessitate an upgraded implementation.

The oracle is also left upgradeable for now while we seek partnerships with providers who can securely manage the on-chain pricing of an off-chain asset with somewhat opaque price discovery. The token itself is an immutable primitive, and as such, avoids the additional on-chain costs associated with upgradeable proxy patterns. By limiting upgrades specifically to certain contract functions we expect that the uncertainty around upgradeability can be scoped while still proving flexibility during the project's growth.

11.3 Phantom Overflow in Price Adjustments Can Break Core Protocol Functions

Context

MaseerGate.sol#L184-L190
MaseerPrice.sol#L54-L57

Description

The protocol's price adjustment functions in `MaseerGate.sol` are vulnerable to phantom overflows when handling large price values. The issue occurs in `_adjustMintPrice()` and `_adjustBurnPrice()`:

```
Unset
function _adjustMintPrice(uint256 _price, uint256 _bps) internal
pure returns (uint256) {
    return (_price * (10_000 + _bps)) / 10_000; // Round up
}

function _adjustBurnPrice(uint256 _price, uint256 _bps) internal
pure returns (uint256) {
    return (_price * 10_000) / (10_000 + _bps) + 1; // Round down
}
```

These functions are called by `mintcost()` and `burncost()` which are used throughout the protocol for critical operations. The issue arises because:

1. `_bps` can be up to 10,000 (100%)
2. This means `(10_000 + _bps)` can be up to 20,000
3. When multiplying a large `_price` by up to 20,000, the result can overflow uint256
4. This causes a Panic 0x11 (arithmetic overflow) error

For example, if `_price` is greater than `type(uint256).max / 20_000`, the multiplication will overflow. This means that a sufficiently large price poked into the `MaseerPrice` contract can cause all functions that call `mintcost()` or `burncost()` to fail.

Our testing found this during fuzzing when we encountered prices that would cause the multiplication to exceed uint256.max. This is particularly concerning because:

- It can break core protocol functionality
- It's triggered by valid price inputs
- The error is not gracefully handled
- It affects multiple critical operations

Recommendation

We recommend implementing a price cap to prevent the overflow. There are three potential implementation points, each with different tradeoffs:

1. In `MaseerPrice.poke()`:

```
Unset
function _poke(uint256 read_) internal {
    if (read_ > type(uint256).max / 20_000) {
        read_ = type(uint256).max / 20_000;
    }
    _setVal(_PRICE_SLOT, bytes32(read_));
    emit PriceUpdate(read_, block.timestamp);
}
```

Pros:

- Prevents invalid prices from entering the system
- Single point of validation

Cons:

- Increases gas cost for oracle updates

2. In `MaseerPrice.read()`:

```
Unset
function read() external view returns (uint256) {
    uint256 price = _uint256Slot(_PRICE_SLOT);
    if (price > type(uint256).max / 20_000) {
        price = type(uint256).max / 20_000;
    }
    return price;
}
```

Pros:

- Transparent to all price consumers
- No changes needed to oracle update logic

Cons:

- Additional gas cost for every price read

3. In `MaseerGate._adjustMintPrice()` and `_adjustBurnPrice()`:

```
Unset
function _adjustMintPrice(uint256 _price, uint256 _bps) internal
pure returns (uint256) {
```

```
    if (_price > type(uint256).max / 20_000) {
        _price = type(uint256).max / 20_000;
    }
    return (_price * (10_000 + _bps)) / 10_000;
}
```

Pros:

- Most gas efficient
- Only affects price adjustments

Cons:

- Multiple points of validation
- Less transparent

We recommend implementing option 1 or 2, as they provide clearer guarantees about price validity throughout the system. The choice between these options depends on the protocol's gas optimization priorities and oracle update frequency.

Suggested Code Changes

Unset

```
diff --git a/src/MaseerPrice.sol b/src/MaseerPrice.sol
index 4fc6364..c0f33af 100644
--- a/src/MaseerPrice.sol
+++ b/src/MaseerPrice.sol
@@ -52,6 +52,9 @@ contract MaseerPrice is MaseerImplementation{
     }

     function _poke(uint256 read_) internal {
+        if (read_ > type(uint256).max / 20_000) {
+            read_ = type(uint256).max / 20_000;
+        }
         _setVal(_PRICE_SLOT, bytes32(read_));
         emit PriceUpdate(read_, block.timestamp);
     }
```

Auditor Notes

During our invariant testing, we discovered this issue when fuzzing price values. We mitigated this with a `_MAX_PRICE = type(uint256).max / 20_000` in the `poke()` and `file()` handlers, The issue is particularly concerning because it can break core protocol functionality with valid price inputs, and the error handling is not graceful.

Client Response

Acknowledged.

Prices are not realistically expected to exceed these values, and oracle updates are handled by permissioned actors.

If a price is pushed that exceeds this value the operations will fail, which is expected and desirable. The additional gas costs are not justified in working around this.

Should a price exceed this value, the oracle can be updated or the price can be updated to be within this range.

11.4 Inconsistent Zero Address Validation in MaseerConduit.move()

Context

MaseerConduit.sol#L50-L64

Description

The `MaseerConduit` contract has two variants of the `move()` function with inconsistent zero address validation:

```
Unset
function move(address _token, address _to) external operator
buds(_to) returns (uint256 _amt) {
    _amt = Gem(_token).balanceOf(address(this));
    _move(_token, _to, _amt);
}
```

```
function move(address _token, address _to, uint256 _amt) external
operator buds(_to) returns (uint256) {
    _move(_token, _to, _amt);
    return _amt;
}
```

The issue is that the first variant attempts to call `balanceOf()` on the token address without first checking if it's zero, while the second variant properly validates the token address in the internal `_move()` function:

```
Unset
function _move(address _token, address _to, uint256 _amt)
internal {
    if (_token == address(0)) revert ZeroAddress();
    _safeTransfer(_token, _to, _amt);
    emit Move(_token, _to, _amt);
}
```

This inconsistency leads to:

1. Different error handling behavior between the two variants
2. Potential EVM errors when calling `balanceOf()` on `address(0)`
3. Unclear error messages for users and developers
4. Inconsistent validation patterns within the same contract

Recommendation

Standardize the zero address validation across both `move()` variants by adding the check before any token operations:

```
Unset
function move(address _token, address _to) external operator
buds(_to) returns (uint256 _amt) {
```

```
    if (_token == address(0)) revert ZeroAddress();
    _amt = Gem(_token).balanceOf(address(this));
    _move(_token, _to, _amt);
}

function move(address _token, address _to, uint256 _amt) external
operator buds(_to) returns (uint256) {
    if (_token == address(0)) revert ZeroAddress();
    _move(_token, _to, _amt);
    return _amt;
}
```

This change would:

1. Provide consistent error handling across both variants
2. Prevent EVM errors from invalid token addresses
3. Give clear error messages to users
4. Follow the principle of failing early and explicitly

Suggested Code Changes

Unset

```
diff --git a/src/MaseerConduit.sol b/src/MaseerConduit.sol
index a468992..6e05330 100644
--- a/src/MaseerConduit.sol
+++ b/src/MaseerConduit.sol
@@ -48,17 +48,18 @@ contract MaseerConduit is
MaseerImplementation {
     }

     function move(address _token, address _to) external operator
buds(_to) returns (uint256 _amt) {
+        if (_token == address(0)) revert ZeroAddress();
         _amt = Gem(_token).balanceOf(address(this));
         _move(_token, _to, _amt);
```

```
     }

     function move(address _token, address _to, uint256 _amt)
external operator buds(_to) returns (uint256) {
+        if (_token == address(0)) revert ZeroAddress();
         _move(_token, _to, _amt);
         return _amt;
     }

     function _move(address _token, address _to, uint256 _amt)
internal {
-        if (_token == address(0)) revert ZeroAddress();
         _safeTransfer(_token, _to, _amt);
         emit Move(_token, _to, _amt);
     }
```

Auditor Notes

During our invariant testing, we discovered this issue when testing the `move()` functions with zero addresses. The inconsistency in error handling between the two variants could lead to confusion and potential issues in production. The EVM errors from calling `balanceOf()` on `address(0)` are particularly problematic as they provide no meaningful error information to users or developers.

Client Response

Fixed.
We now provide uniform errors and checks against `address(0)` in both implementations of `move()`
https://github.com/Maseer-LTD/maseer-one/pull/4

11.5 ERC20 Transfers to Address(0) Allowed in MaseerOne

Context

MaseerToken.sol#L44-L73
MaseerToken.sol#L121-L137
MaseerGuard.sol#L26-L28

Description

MaseerOne token allows transfers to `address(0)`. While this is not a security vulnerability in itself, it could lead to several integration and user experience issues:

1. Transfer to Address(0) Risks
    - Users might accidentally send tokens to `address(0)` thinking it's a valid address
    - Unlike transfers to other addresses, transfers to `address(0)` are permanent and cannot be recovered
    - This could lead to user confusion and potential loss of funds
    - Many DeFi protocols and wallets don't properly handle transfers to `address(0)`, causing UI/UX issues

2. Mint to Address(0) Risks
    - If `mint()` is called with `address(0)` as the recipient, tokens would be permanently lost
    - This could lead to incorrect supply calculations and accounting issues
    - The loss of tokens through minting to `address(0)` would be permanent and unrecoverable

3. Burn from Address(0) Risks
    - While `address(0)` cannot hold tokens (as it cannot be a recipient), allowing burns from `address(0)` could lead to confusion
    - This could complicate token accounting and tracking
    - It may be unclear whether a burn from `address(0)` represents a legitimate operation

4. Integration and Tracking Complications
    - Transfers to `address(0)` can complicate token accounting and tracking
    - It may be unclear whether a transfer to `address(0)` represents a burn or a transfer to a non-existent address
    - This ambiguity can cause issues with balance tracking and supply calculations
    - Forensic analysis in case of issues becomes more complicated

Recommendation

The recommended solution is to modify `MaseerGuard` to include `address(0)` in its blacklist functionality. This approach:

1. Leverages Existing Infrastructure
    - Uses the existing blacklist mechanism in `MaseerGuard`
    - Maintains consistency with the protocol's security model
    - Requires minimal changes to the codebase
2. Provides Comprehensive Protection

    - Prevents transfers to `address(0)`
    - Prevents mints to `address(0)`
    - Note: Does not prevent burns from `address(0)` as the issuer can still smelt from that address
    - Works with all token operations that use the guard
3. Maintains Protocol Design
    - Aligns with the protocol's existing security model
    - Doesn't require changes to the token contract itself
    - Preserves the ability to burn tokens through explicit burn functions

Suggested Code Changes

Unset

```
diff --git a/src/MaseerGuard.sol b/src/MaseerGuard.sol
index 7ea5089..ba7b306 100644
--- a/src/MaseerGuard.sol
+++ b/src/MaseerGuard.sol
@@ -24,6 +24,7 @@ contract MaseerGuard is MaseerImplementation {
     * @return bool Returns `true` if the user can pass, otherwise `false`.
     */
    function pass(address usr) external view returns (bool) {
+       if (usr == address(0)) return false;
        return !GuardSource(source).getBlackListStatus(usr);
    }
}
```

Auditor Notes

This finding was discovered during our review of the token implementation. While not a security vulnerability, it's important to consider the implications for integrations and user experience. The current implementation's approach to burning through transfers to `address(0)` is a common pattern in ERC20 tokens, but it may be worth considering more explicit alternatives to improve clarity and prevent accidental token loss. The recommended solution using `MaseerGuard` provides a clean way to prevent most operations involving `address(0)` while maintaining the protocol's existing security model. Note that this solution does not prevent burns from `address(0)` as the issuer can still smelt from that address.

Client Response

Acknowledged.

`address(0)` is already a blacklisted user via the guard. Additional checks are redundant. The token is an unopinionated abstract and this case is handled in the implementation. Test cases cover this at https://github.com/Maseer-LTD/maseer-one/blob/5d84bce236371794997a9909ad0f20deda1730cc/test/MaseerOne.token.t.sol#L89-L116

Follow-up Auditor Notes

The only followup comment we would make here after a discussion with Maseer is that there is the theoretical possibility that Tether removes `address(0)` from their blacklist, which would realize this risk. We think this is highly unlikely, and if it did happen, a contract upgrade could be performed to add this functionality. For now, it's defense-in-depth to add it, but it would come at a socialized gas cost for an extremely unlikely scenario. We agree with Maseer's decision not to add this functionality, but encourage a monitoring regime to ensure removal of `address(0)` is discovered and eventually remediated.

12. Gas Optimization

12.1 Inconsistent Error Handling Styles Between MaseerToken and MaseerOne

Context

MaseerToken.sol#L84-L113
MaseerOne.sol#L110-L128

Description

The codebase exhibits inconsistent error handling patterns across related contracts. `MaseerToken.sol` primarily relies on `require` statements for validation and error handling, while `MaseerOne.sol` (which inherits from `MaseerToken`) uses custom errors with `revert` statements.

For example, in `MaseerToken.sol`:

```
Unset
require(block.chainid == CHAIN_ID,   "INVALID_CHAIN");
require(deadline >= block.timestamp, "PERMIT_DEADLINE_EXPIRED");
```

```
require(recoveredAddress != address(0) && recoveredAddress ==
owner, "INVALID_SIGNER");
```

While in `MaseerOne.sol`:

```
Unset
if (!_canPass(usr_)) revert UnauthorizedUser(usr_);
if (!mintable()) revert MarketClosed();
if (_unit == 0) revert InvalidPrice();
```

This inconsistency makes error handling harder to audit, test, and less gas efficient in the parent contract. It also creates a confusing mixed pattern within the same inheritance chain.

Our testing found this inconsistency during fuzzing, where we needed to handle multiple error patterns in catch blocks. This can lead to more complex error handling in interacting systems and potential missed error states.

Recommendation

Standardize the error handling approach across the entire codebase. Consider migrating all error handling to custom errors with `revert` (as used in `MaseerOne.sol`), which are more gas efficient and provide more detailed information.

The use of custom errors carries several advantages:

1. Lower gas costs (no need to store error strings on-chain)
2. Ability to pass dynamic data in errors for better debugging
3. Stronger type safety and easier error identification
4. Better ABI interface generation for frontends and integrations

Suggested Code Changes

```
Unset
// In MaseerToken.sol
+ error InvalidChain(uint256 expected, uint256 actual);
```

```
+ error PermitDeadlineExpired(uint256 deadline, uint256
timestamp);
+ error InvalidSigner(address recovered, address expected);

function permit(...) public virtual {
-   require(block.chainid == CHAIN_ID, "INVALID_CHAIN");
+   if (block.chainid != CHAIN_ID) revert InvalidChain(CHAIN_ID,
block.chainid);

-   require(deadline >= block.timestamp,
"PERMIT_DEADLINE_EXPIRED");
+   if (deadline < block.timestamp) revert
PermitDeadlineExpired(deadline, block.timestamp);

    // Later in function:
-   require(recoveredAddress != address(0) && recoveredAddress ==
owner, "INVALID_SIGNER");
+   if (recoveredAddress == address(0) || recoveredAddress !=
owner) revert InvalidSigner(recoveredAddress, owner);
}
```

Client Response

Acknowledged.

The MaseerToken was adapted from the solmate implementation and these errors were left untouched. These have been streamlined to include custom errors at https://github.com/Maseer-LTD/maseer-one/pull/3

12.2 MaseerGate mixes paradigms for setting state

Context

src/MaseerGate.sol#L94-L150

Description

The `MaseerGate` contract mixes paradigms for allowing authed users to set contract variables. This contract uses both a set of setters:

```
Unset
    function setOpenMint(uint256 open_) external auth {
        if (open_ > block.timestamp + 365 days) revert
UnrecognizedParam(bytes32(open_));
        _setOpenMint(open_);
    }

    function setHaltMint(uint256 halt_) external auth {
        if (halt_ > block.timestamp + 365 days) revert
UnrecognizedParam(bytes32(halt_));
        _setHaltMint(halt_);
    }

    function setOpenBurn(uint256 open_) external auth {
        if (open_ > block.timestamp + 365 days) revert
UnrecognizedParam(bytes32(open_));
        _setOpenBurn(open_);
    }

    function setHaltBurn(uint256 halt_) external auth {
        if (halt_ > block.timestamp + 365 days) revert
UnrecognizedParam(bytes32(halt_));
        _setHaltBurn(halt_);
    }

    function pauseMarket() external auth {
        _setOpenMint(0);
        _setHaltMint(0);
        _setOpenBurn(0);
        _setHaltBurn(0);
    }

    function setBpsin(uint256 bpsin_) external auth {
        if (bpsin_ > 10000) revert
UnrecognizedParam(bytes32(bpsin_));
```

```
        _setBpsin(bpsin_);
    }

    function setBpsout(uint256 bpsout_) external auth {
        if (bpsout_ > 10000) revert
UnrecognizedParam(bytes32(bpsout_));
        _setBpsout(bpsout_);
    }

    function setCooldown(uint256 cool_) external auth {
        if (cool_ > 365 days) revert
UnrecognizedParam(bytes32(cool_));
        _setCooldown(cool_);
    }

    function setCapacity(uint256 capacity_) external auth {
        _setCapacity(capacity_);
    }
```

And a Maker style `file()` interface:

Unset

```
    function file(bytes32 what, uint256 data) external auth {
        if      (what == "openmint") _setOpenMint(data);
        else if (what == "haltmint") _setHaltMint(data);
        else if (what == "openburn") _setOpenBurn(data);
        else if (what == "haltburn") _setHaltBurn(data);
        else if (what == "bpsin")    _setBpsin(data);
        else if (what == "bpsout")   _setBpsout(data);
        else if (what == "cooldown") _setCooldown(data);
        else if (what == "capacity") _setCapacity(data);
        else    revert UnrecognizedParam(what);
    }
```

These two interfaces add gas cost to the deploy of `MaseerGate` and unnecessary cognitive overhead and duplicate effort to integrations. What's more, there is a minimal chance on upgrades to introduce bugs by having to pattern the code this way.

Recommendation

We would not recommend keeping both interface patterns. The setter pattern is more expensive for deployment, but also the most common pattern for doing updates like this. It follows with `rely()` and `deny()` and other contracts in the project. There is one exception. The `MaseerPrice`, or `pip`, has the old Maker style `file()` interface. This was probably done to make oracle integrations easier. We think this interface should be updated to whatever makes the most sense for the oracle and gas costs.

For the rest of the contracts, we would recommend simple setters.

Suggested Code Changes

Unset

```
--- a/src/MaseerGate.sol
+++ b/src/MaseerGate.sol
@@ -137,18 +137,6 @@ contract MaseerGate is MaseerImplementation
{
          _setCapacity(capacity_);
      }

-    function file(bytes32 what, uint256 data) external auth {
-        if      (what == "openmint") _setOpenMint(data);
-        else if (what == "haltmint") _setHaltMint(data);
-        else if (what == "openburn") _setOpenBurn(data);
-        else if (what == "haltburn") _setHaltBurn(data);
-        else if (what == "bpsin")    _setBpsin(data);
-        else if (what == "bpsout")   _setBpsout(data);
-        else if (what == "cooldown") _setCooldown(data);
-        else if (what == "capacity") _setCapacity(data);
-        else    revert UnrecognizedParam(what);
-    }
-
      function _setOpenMint(uint256 open_) internal {
          _setVal(_OPEN_MINT_SLOT, bytes32(open_));
```

```
    }
```

Client Response

Acknowledged.

The named external functions offer some guardrails that prevent accepting values outside of expected ranges. The generalized `file` function here enables direct setting of values in this contract by the authorized user outside of these conditions in the case of unanticipated market conditions.
