CANTINA

Primev

Competition

December 18, 2024

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

A competition provides a broad evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While competitions endeavor to identify and disclose all potential security issues, they cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities, therefore, any changes made to the code would require an additional security review. Please be advised that competitions are not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Primev aims to solve coordination inefficiencies in decentralized domains additively to mev, driving network welfare and increase transparency and decentralization of the mev pipeline.

From Dec 3rd to Dec 8th Cantina hosted a competition based on primev-mev-commit. The participants identified a total of **8** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 2
- Medium Risk: 2
- Low Risk: 4
- Gas Optimizations: 0
- Informational: 0

The present report only outlines the **critical**, **high** and **medium** risk issues.

3 Findings

3.1 High Risk

3.1.1 `blockBuilderBLSKeyToAddress[]` can be overwritten in `ProviderRegistry.sol::_registerAndStake()`

Submitted by sammy, also found by amaron

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: `blockBuilderBLSKeyToAddress[]` can be overwritten in `ProviderRegistry.sol::_registerAndStake()`, allowing a malicious user to pass a block winner's `blsKey` during registration and prevent them from receiving rewards for successful commitments. Alternatively, a malicious provider can leverage this to avoid getting slashed for blocks they've built that have not included transactions from a commitment.

In the new version, a new mapping has been added to `ProviderRegistry.sol`, i.e, `blockBuilderBLSKeyToAddress[]`, this mapping is updated as follows:

```
function _registerAndStake(address provider, bytes[] calldata blsPublicKeys) internal {
    require(!providerRegistered[provider], ProviderAlreadyRegistered(provider));
    require(msg.value >= minStake, InsufficientStake(msg.value, minStake));
    require(blsPublicKeys.length != 0, AtLeastOneBLSKeyRequired());
    uint256 numKeys = blsPublicKeys.length;
    for (uint256 i = 0; i < numKeys; ++i) {
        bytes memory key = blsPublicKeys[i];
        require(key.length == 48, InvalidBLSPublicKeyLength(key.length, 48));
        blockBuilderBLSKeyToAddress[key] = provider; // <<<
    }
    eoaToBlsPubkeys[provider] = blsPublicKeys;
    providerStakes[provider] = msg.value;
    providerRegistered[provider] = true;
    emit ProviderRegistered(provider, msg.value, blsPublicKeys);
}
```

This means that any user can register as a provider and update the `blockBuilderBLSKeyToAddress[]` to their address. In `BlockTracker.sol`, the `recordL1Block()` function now records the winner of a particular block based on this mapping:

```
function recordL1Block(
    uint256 _blockNumber,
    bytes calldata _winnerBLSKey
) external onlyOracle whenNotPaused {
    address _winner = providerRegistry.getEoaFromBLSKey(_winnerBLSKey); // <<<
    _recordBlockWinner(_blockNumber, _winner);
    uint256 newWindow = (_blockNumber - 1) / WindowFromBlockNumber.BLOCKS_PER_WINDOW + 1;
    if (newWindow > currentWindow) {
        // We've entered a new window
        currentWindow = newWindow;
        emit NewWindow(currentWindow);
    }
    emit NewL1Block(_blockNumber, _winner, currentWindow);
}
```

```
function getEoaFromBLSKey(bytes calldata blsKey) external view returns (address) {
    return blockBuilderBLSKeyToAddress[blsKey];
}
```

This was likely done to avoid the scenario where an arbitrary block winner can open a commitment without even being registered in the provider registry. There are two attack paths that can exploit this vulnerability:

- Attack 1: A malicious bidder performs this attack to avoid paying `bidAmt` by registering as a provider and using the block winner's `blsKey`.
    1. Honest provider `A` registers in provider registry with the blsKey `key`.
    2. `A` accepts bidder `B`'s bid and stores it. Now, the malicious bidder registers as a provider with `key`.

3. L1 block is processed and `A` successfully fulfills the commitment.
4. Oracle calls `recordL1Block()`, with `key` as argument, however the `_winner` will be set to the malicious bidder's address.
5. `A` calls `openCommitment()` now that the oracle has done it's duty, however it reverts, as `A` is not `_winner`.
6. `B` gets away with not having to pay for the commitment, and `A` loses funds.
7. `B` can now unstake from the registry and get `minStake` back.

- Attack 2: A malicious provider performs this attack when they have built the L1 block but haven't included the bidder's transactions to avoid getting slashed.
  1. Malicious provider `A` registers in provider registry with the blsKey `key`.
  2. `A` accepts bidder `B`'s bid and stores it.
  3. `A` builds L1 block, but doesn't include `B`'s transactions.
  4. `A` quickly registers another address in the provider registry using `key`.
  5. Block winner is recorded as this new address instead of `A`.
  6. Bidder calls `openCommitment()` to slash `A` and get compensated, however it reverts as the commiter is not the block winner.
  7. `A` unstakes with the new address and gets `minStake` back.

Impact: High -- Loss of funds in both scenarios.

Likelihood: High -- There's no cost for this attack except gas fees.

Recommendation: Revert if `blockBuilderBLSKeyToAddress[key]` is already populated in `_registerAndStake()`.

Note: This may still allow an attack vector where a user can front-run `registerAndStake()` transactions by copying the arguments and DoSing honest providers from registering. Extended remediation is up for discussion.

3.1.2 Overpayment to Bidder in Slash Function Due to Incorrect Amount Transfer

Submitted by 0xNirix, also found by Nexarion and sashik-eth

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: The `slash` function in ProviderRegistry contains a critical bug where it sends the full `amt` to bidders instead of the calculated `residualAmt` (amt after decay). While the contract correctly calculates `residualAmt = (amt * residualBidPercentAfterDecay) / ONE_HUNDRED_PERCENT`, it erroneously calls `bidder.send(amt)` instead of `bidder.send(residualAmt)`. This results in bidders receiving more funds than intended, potentially draining the contract's balance. For example, with a 50% decay rate, a bidder would receive 1 ETH instead of the intended 0.5 ETH.

Proof of Concept: Add this test case in `ProviderRegistryTest.sol`:

```
function test_SlashResidualAmountBug() public {
    // Setup: Register provider with initial stake
    vm.deal(provider, 3 ether);
    vm.prank(provider);
    providerRegistry.registerAndStake{value: 2 ether}(validBLSPubkeys);
    providerRegistry.setPreconfManager(address(this));

    // Setup: Prepare bidder and track initial balances
    address bidder = vm.addr(4);
    uint256 initialBidderBalance = bidder.balance;
    uint256 initialProviderStake = providerRegistry.getProviderStake(provider);

    // Calculate expected values
    uint256 slashAmount = 1 ether;
    uint256 residualPercent = 50 * providerRegistry.PRECISION(); // 50% decay
    uint256 expectedResidualAmount = (slashAmount * residualPercent) / providerRegistry.ONE_HUNDRED_PERCENT();
    uint256 expectedPenaltyFee = (expectedResidualAmount * providerRegistry.feePercent()) /
↪   providerRegistry.ONE_HUNDRED_PERCENT();

    // Execute slash with 50% decay
    providerRegistry.slash(slashAmount, provider, payable(bidder), residualPercent);

    // Verify the actual transfer amount vs expected
    uint256 actualBidderReceived = bidder.balance - initialBidderBalance;
    uint256 actualProviderLost = initialProviderStake - providerRegistry.getProviderStake(provider);

    // The bug manifests here - bidder receives full amount instead of residual
    assertEq(actualBidderReceived, slashAmount, "Bidder incorrectly received full amount instead of residual");
    assertGt(actualBidderReceived, expectedResidualAmount, "Bidder received more than residual amount");

    // Calculate the excess payment
    uint256 excessPayment = actualBidderReceived - expectedResidualAmount;
    assertGt(excessPayment, 0, "Excess payment should be greater than 0");

    // Total deducted from provider matches residual + fee
    assertEq(actualProviderLost, expectedResidualAmount + expectedPenaltyFee,
        "Provider lost correct amount despite overpayment to bidder");
}
```

3.2 Medium Risk

3.2.1 Lack of `processedTxnHashes[]` check in `storeUnopenedCommitment()`

Submitted by sammy, also found by bbl4de, amaron, amaron and amaron

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: A bidder can avoid paying for successful commitments due to lack of `processedTxnHashes[]` check in `storeUnopenedCommitment()`.

In the new implementation of `openCommitment()`, there is a new mapping `processedTxnHashes[]`, which prevents a bidder from creating two different bids with the same `txnHash`. However, this check is not present in `storeUnopenedCommitment()`, which makes it possible for a bidder to not pay for a successful commitment, after an unsuccessful one.

- Attack vector:
    1. Bidder `A` places a bid at block `x` with provider `Z`.
    2. Provider `Z` builds the block but doesn't involve the bidder's transaction(s) in the block.
    3. Bidder calls `openCommitment()` and `processedTxnHashes[]` for the `txnHash + bidder` hash is set to `true`.
    4. Provider `Z` is slashed.
    5. Bidder `A`, desperate to get their transaction included, creates a new bid with Provider `Y` for the same `txnHash` for block `x + 10`.
    6. Provider `Y` accepts the bid, calls `storeUnopenedCommitment()`, since there's no check for `processedTxnHashes[]`, the provider doesn't suspect anything.

7. Provider `Y` builds block `x+10` with the bidder's transaction included.
8. Provider `Y`, seeking compensation for it's honest behaviour, calls `openCommitment()`, however the transaction reverts due to the check:

```
require(
    processedTxnHashes[txnHashAndBidder] == false,
    TxnHashAlreadyProcessed(txnHash, bidderAddress)
);
```

9. Provider `Y` will not get rewarded, and hence loses `bidAmt`, `bidder` avoids paying `bidAmt`.

Impact: Loss of funds for an honest provider, bidders can take advantage of this to avoid paying for successful commitments after a previous unsuccessful commitment.

Likelihood: The likelihood of this scenario is med-high as:

1. Block builders can fail to include transactions, and that's why there exists an entire infrastructure to penalize them, in the scenario of a failed commitment.
2. A bidder will try to get their transactions included through a different provider after a previous one fails.
3. It is very unlikely that a provider will check the `processedTxnHashes[]` mapping off-chain, and this is something that isn't specified anywhere in the code, the docs or the contest readme.
4. Neither party needs to be particularly "malicious" for this scenario to take place.

Recommendation: The fix for this is tricky, including `txnHash` as an argument in `storeUnopenedCommitment()` might be a breach of the bidder's privacy, but currently this is the only viable remidiation.

3.2.2 Users funds can get stuck forever

Submitted by mxuse, also found by bbl4de, amaron, amaron and pepoc3

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: The `mev-commit` team introduced two new parameters: `counterpartyFee` and `finalizationFee`. These fees are interdependent, meaning that any change to one requires a corresponding update to the other to maintain consistency. For example:

If `counterpartyFee` is updated to 0.1 ETH, then `finalizationFee` must also be updated to 0.1 ETH. The problem is that this will not prevent a case where a user's transaction might be stuck forever.

Consider the following:

- `counterpartyFee` = 0.1 eth `finalizationFee` = 0.1 ETH.
- Alice calls `initiateTransfer` with 0.1 ETH.
- `counterpartyFee` is updated to 0.12 ETH.
- `finalizationFee` is then also updated to 0.12 ETH to ensure consistency.
- `finalizeTransfer` is called.
- This will never go through since the function will revert here:

```
function finalizeTransfer(address _recipient, uint256 _amount, uint256 _counterpartyIdx)
    external onlyRelayer whenNotPaused nonReentrant {
    require(_amount >= finalizationFee, // ... <<<
}
```

- Funds will be stuck.

It is worth noting that while the update functions are owner controlled it is impossible to ensure that no one is mid-flight while updating these values, also, there is no functionality to rescue these funds meaning they will be stuck forever. lastly, `initiateTransfer` is a user called function.

Impact: User funds are stuck and essentially lost.

Proof of Concept:

- Modify the `counterpartyfee` to be equal to the `finalizationfee` in 'L1GatewayTest.sol:

```
contract L1GatewayTest is Test {
    L1Gateway l1Gateway;
    address owner;
    address relayer;
    address bridgeUser;
    uint256 finalizationFee;
    uint256 counterpartyFee;

    function setUp() public {
        owner = address(this); // Original contract deployer as owner
        relayer = address(0x78);
        bridgeUser = address(0x101);
        finalizationFee = 0.1 ether;
        counterpartyFee = 0.1 ether;
```

- Paste the following test-function inside `L1GatewayTest.sol`:

```
function test_InitiateTransferWithFeeUpdate() public {

    vm.deal(bridgeUser, 0.1 ether); // give user equal amount of eth as the fee
    uint256 amount = 0.1 ether;

    vm.prank(bridgeUser);
    l1Gateway.initiateTransfer{value: amount}(bridgeUser, amount);

    // Update the counterparty fee and finalization fee
    l1Gateway.setCounterpartyFee(0.12 ether);
    l1Gateway.setFinalizationFee(0.12 ether);

    // Attempt to finalize the transfer
    vm.expectRevert(abi.encodeWithSelector(IGateway.AmountTooSmall.selector, 0.1 ether, 0.12 ether));
    vm.prank(relayer);
    l1Gateway.finalizeTransfer(bridgeUser, amount, 1);
}
```

- Run `forge test --match-test test_InitiateTransferWithFeeUpdate -vvv`.
- Logs:

```
[PASS] test_InitiateTransferWithFeeUpdate() (gas: 80666)
```

Recommendation: We would recommend introducing a rescue function in case this happens to a user, alternatively, come up with some other mitigation.