Hermez

Security Assessment

December 23, 2020

Prepared For:
David Schwartz | *Hermez Network*
david@iden3.io

Jordi Baylina | *Hermez Network*
jordi@iden3.io

Prepared By:
Josselin Feist | *Trail of Bits*
josselin@trailofbits.com

Natalie Chin | *Trail of Bits*
natalie.chin@trailofbits.com

Jim Miller | *Trail of Bits*
james.miller@trailofbits.com

Changelog:

November 9, 2020:	Initial report delivered
November 24, 2020:	Improvements to TOB-HERMEZ-013
December 11, 2020:	Added Appendix F with retest results
December 23, 2020:	Updated Appendix F with additional retest results

Executive Summary

Project Dashboard

Code Maturity Evaluation

Engagement Goals

Coverage

Recommendations Summary
- Short term
- Long term

Findings Summary
- 1. Lack of a contract existence check allows token theft
- 2. No incentive for bidders to vote earlier
- 3. L1 transaction spam
- 4. Account creation spam
- 5. Lack of access control separation is risky
- 6. Lack of two-step procedure for critical operations leaves them error-prone
- 7. Lack of a contract existence check in TimeLock leads to incorrect assumption of code execution
- 8. Insufficient Logging
- 9. Lack of zero check on functions
- 10. Multiple contracts are missing inheritance
- 11. Using empty functions instead of interfaces leaves contract error-prone
- 12. Initialization functions can be front-run
- 13. Re-entrancy risks on TokenHez
- 14. ChainId usage can lead to collisions
- 15. Lack of overflow check on allocation ratio allows AuctionProtocol to be siphoned
- 16. Arithmetic rounding allows getMinBidBySlot to return the current bid value
- 17. _checkSig allows signature re-use
- 18. changeDefaultSlotSetBid allows the closed minimum bid of an open slot to be updated
- 19. cancelTransaction can be called on non-queued transaction
- 20. Contracts used as dependencies do not track upstream changes
- 21. Expected behavior regarding authorization for adding tokens is unclear
- 22. Contract name duplication leaves codebase error-prone

A. Vulnerability Classifications

B. Code Maturity Classifications

C. Code Quality Recommendations
- Smart contracts

D. Inconsistencies Between Code and Specification

E. Token Integration Checklist
- General security considerations
- ERC conformity
- Contract composition
- Owner privileges
- Token scarcity

F. Fix Log

Executive Summary

From October 26, 2020, through November 9, 2020, Iden3 engaged Trail of Bits to review the security of Hermez. Trail of Bits conducted this assessment over four person-weeks with three engineers working from `d52ed73` (`contracts`) and `a785328` (`circuits`).

In the first week, we focused on understanding the codebase. We reviewed the L1/L2 flow on the smart contracts against the most common Solidity flaws and the circuit codebase to ensure the code matched its specifications. In week two, we continued reviewing the contracts and the interactions between L1 and L2.

Our review identified 22 issues ranging from high to informational severity. The high-severity issues include:

- Missing contract existence check which allows attackers to steal tokens (TOB-HERMEZ-001)
- Lack of access control separation for frequently updated values (TOB-HERMEZ-005)
- Front-running of initialization functions (TOB-HERMEZ-012)

Appendix C makes additional code quality recommendations, Appendix D lists inconsistencies between the circuit specifications and the circuit code, and Appendix E contains a checklist for secured token integration.

Overall, the codebase follows Solidity best practices, and Iden3 avoids the most common smart contract pitfalls. However, Iden3 could improve the contract specification and identify the corner cases of functions. This documentation will help ensure the code matches the expected behavior. Additionally, the use of low-level manipulations and assembly increases the underlying risks. On the circuit side, we found the code mostly compliant with the specification, aside from some documentation errors noted in Appendix D. We do not report any findings related to the security of the circuit files.

We recommend that Iden3 address all the findings, improve the code specification, and use fuzzing or symbolic execution to check the contract invariants.

Update December 23, 2020: Trail of Bits reviewed fixes implemented for the issues present in this report. See the results from this fix review in Appendix F: Fix Log.

Project Dashboard

Application Summary

Name	Hermez
Version	`d52ed73` (`contracts`) and `a785328` (`circuits`)
Type	Solidity, Circom
Platforms	Ethereum

Engagement Summary

Dates	October 26–November 9, 2020
Method	Whitebox
Consultants Engaged	3
Level of Effort	4 person-weeks

Vulnerability Summary

Total High-Severity Issues	7	■ ■ ■ ■ ■ ■ ■
Total Medium-Severity Issues	4	■ ■ ■ ■
Total Low-Severity Issues	4	■ ■ ■ ■
Total Informational-Severity Issues	7	■ ■ ■ ■ ■ ■ ■
Total	22	

Category Breakdown

Access Controls	2	■ ■
Auditing and Logging	1	■
Configuration	1	■
Cryptography	1	■
Data Validation	11	■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■
Patching	1	■
Timing	2	■ ■
Undefined Behavior	3	■ ■ ■
Total	22	

Code Maturity Evaluation

Category Name	Description
Access Controls	**Satisfactory.** Privileged operations are clearly defined and the authorization is properly validated. However, the authorization required for adding tokens should be clarified and the expected behavior of the governance contract documented.
Arithmetic	**Moderate.** The codebase lacks overflow/underflow checks in many locations. This resulted in one high-severity issue.
Assembly Use	**Satisfactory.** The codebase relies heavily on optimized assembly code, which increases the likelihood of issues. However, the assembly code is well documented, and only minor issues were found.
Centralization	**Moderate.** While the system aims to be decentralized, it currently heavily relies on the deployer and privilege operators. As the system is at an early stage, this level of centralization is acceptable.
Upgradeability	**Moderate.** The use of the `delegatecall` proxy pattern leaves the codebase vulnerable to a front-running attack at deployment.
Function Composition	**Moderate.** The code is divided into folders with contracts grouped according to their functionality. Interfaces are used for Solidity inheritance, but some contracts fail to inherit their interfaces.
Front-Running	**Moderate.** The use of the `delegatecall` proxy pattern leaves the codebase vulnerable to a front-running attack at deployment.
Key Management	**Not Considered.**
Monitoring	**Moderate.** The code contains precautions taken in the event of an attack on the protocol. However, we were not provided with a clear incident response plan, and we found missing events in monitoring contracts for state variable changes (TOB-HERMEZ-008).
Specification	**Moderate.** While the code has significant documentation, it lacks function-level specifications. Many issues related to corner cases resulted from a lack of specification.
Testing & Verification	**Moderate.** The testing procedure includes unit tests which check for successful execution. However, the test cases are missing checks for failing cases and corner cases, and do not include automated tools such as fuzzers or static analyzers.

Engagement Goals

The engagement was scoped to provide a security assessment of the Hermez smart contracts in the contract and circuit repositories.

Specifically, we sought to answer the following questions:

- Is it possible to steal funds?
- Can L1 transactions block the protocol from forging new blocks?
- Is the upgrade strategy robust enough to adapt to new protocol changes?
- Is there any arithmetic overflow or underflow affecting the code?
- Can participants perform denial-of-service or spam attacks against any of the components?
- Are there gaps between the circuit specification and its implementation?

Coverage

The engagement was focused on the following components:

`HermezAuctionProtocol`. Facilitates the bidding process for coordinators, where the bidder with the highest bid price in HEZ tokens can forge new batches. We manually reviewed this contract and used automatic tools to verify that the arithmetic in this contract is correct. We looked for flaws in the bidding system that would allow an attacker to steal tokens, win without transferring tokens, or bid on a closed slot.

Hermez. Manages the L2 state by allowing coordinators to forge batches, users to submit additional L1 roll-up transactions, and allowing the use of the entry point for withdrawal of funds. We manually reviewed these contracts to ensure no one can spam them and that valid states can transition into the next state. We reviewed the addition of L1 transactions, and looked for ways for a coordinator could skip L1 transactions or forge malicious coordinator transactions. Finally, we reviewed the assembly code to ensure it matches the expected behavior.

`InstantWithdrawManager`, `WithdrawalDelayer`. This set of contracts allows a user to facilitate an `InstantWithdrawal` if the transaction value is below a credit amount. Otherwise, the contracts force the user to undergo a delayed withdrawal, i.e., wait until a dispute time period has passed before withdrawing their funds. We manually reviewed these contracts to ensure that no one can take advantage of the withdrawal functionality to steal funds. We also checked that the buckets could not be abused to withdraw more funds than expected. Finally, we looked for flaws that would allow an attacker to bypass the delay period.

Circom circuits. To leverage zero-knowledge proofs for Hermez's zk-rollups, the logic must be converted into circuits. Hermez wrote this logic in files that can be processed by their tool, circom, which then compiles this logic into circuits that can be used in zero-knowledge proofs. We reviewed these files to verify that they match the specification. We also verified that these files perform the proper checks to validate each transaction type, e.g., ensuring that accounts exist for transfers and ensuring the sender has a sufficient balance. We confirmed that the circuits comply with all the assumptions made from the smart contracts about their behavior (and vice-versa). Lastly, we reviewed the system for higher-level concerns, such as making sure the system is protected if the Poseidon hash function is insecure.

Recommendations Summary

This section aggregates all the recommendations made during the engagement. Short-term recommendations address the immediate causes of issues. Long-term recommendations pertain to the development process and long-term design goals.

Short term

☐ **Check for contract existence in `_safeTransferFrom`. Add a similar check for any low-level calls, including in `WithdrawalDelayer`.** This will prevent an attacker from listing and depositing tokens in a contract that is not yet deployed. TOB-HERMEZ-001

☐ **Explore ways to incentivize users to vote earlier.** Consider a weighted bid, with a weight decreasing over time. While it won't prevent users with unlimited resources from manipulating the vote at the last minute, it will make the attack more expensive and reduce the chance of vote manipulation. TOB-HERMEZ-002

☐ **Add a fee for any L1 transaction, to be paid in Hermez tokens.** The fee will prevent Ethereum miners from spamming the contract for free. TOB-HERMEZ-003

☐ **Add a fee for account creation or ensure `MAX_NLEVELS` is at least 32. Also, monitor account creation and alert the community if a malicious coordinator spams the system.** This will prevent an attacker from spamming the system to prevent new accounts from being created. TOB-HERMEZ-004

☐ **Use a separate account to handle updating the tokens/USD ratio.** Using the same account for the critical operations and update the tokens/USD ratio increases underlying risks. TOB-HERMEZ-005

☐ **Use a two-step procedure for all non-recoverable critical operations to prevent irrecoverable mistakes.** TOB-HERMEZ-006

☐ **Check for contract existence in `Timelock.executeTransaction` if `callData` is not empty.** This will prevent the success of transactions that do not execute code. TOB-HERMEZ-007

☐ **Add events in `InstantWithdrawManager` so users can easily identify when system parameters change.** Without events, system monitoring is more difficult. TOB-HERMEZ-008

☐ **Add zero-value checks on all function arguments to ensure users can't accidentally set incorrect values.** This will prevent incorrect configuration of the system. TOB-HERMEZ-009

☐ **Ensure `WithdrawalDelayer` inherits from `WithdrawalDelayerInterface`, and `HermezAuctionProtocol` from `AuctionInterface`.** This will ensure the contracts stay up to date with their interfaces. TOB-HERMEZ-010

☐ **Use an interface instead of a contract in `WithdrawalDelayerInterface`.** This will make derived contracts follow the interface properly. TOB-HERMEZ-011

☐ **Prevent the front-running of initialization either by using a factory pattern or by ensuring that the deployment scripts are robust.** An attacker can front-run the system deployment to call the initialization functions with malicious parameters. TOB-HERMEZ-012

☐ **Ensure `TokenHez` does not have callback features, or follow the check-effects-interactions pattern.** The codebase has many re-entrancy patterns that could be exploited if `TokenHez` had a callback mechanism. TOB-HERMEZ-013

☐ **Document that Hermez must be deployed on a fork with a `chainID` below 65,535, or revert if `chainID` is above 65,535.** Collision on the circuits' input can happen if `chainId` is above 65,535. TOB-HERMEZ-014

☐ **Add checks to ensure that each allocation ratio index is < 10,000.** This will protect against overflow. TOB-HERMEZ-015

☐ **Add +1 to the value returned by `getMinBidBySlot`, or document that the bidding increase will not work for a low bid value.** Otherwise, an arithmetic rounding in `getMinBidBySlot` can make the function return the current bid value without the expected increase. TOB-HERMEZ-016

☐ **Add signature re-use mitigations in `_checkSig` against both contract re-deployment and forks.** This ensures that signatures can't be replayed across different versions. TOB-HERMEZ-017

☐ **Iterate over `i < current + _closedAuctionSlots` in `changeDefaultSlotSetBid`.** The loop currently iterates over one open bid. TOB-HERMEZ-018

☐ **Check that the transaction to be canceled exists in `cancelTransaction`.** This will ensure that monitoring tools can rely on emitted events. TOB-HERMEZ-019

☐ **Identify the areas in the code that are relying on external libraries and use an Ethereum development environment and NPM to manage packages as part of your project.** This will ensure that the code pulls vulnerability fixes. TOB-HERMEZ-020

☐ **Update either the implementation or the documentation to standardize the authorization specification for adding tokens.** Currently, the implementation and its documentation differ on who is authorized to add tokens. TOB-HERMEZ-021

☐ **Prevent the re-use of duplicate contract names or change the compilation framework.** Codebase with contracts name duplicate is not properly handled by `builder-waffle`. TOB-HERMEZ-022

Long term

☐ **Carefully review the Solidity documentation, especially the Warnings section. Carefully review the pitfalls of using `delegatecall` proxy pattern.** Three issues were present due to low-level Solidity manipulation TOB-HERMEZ-001, TOB-HERMEZ-007, TOB-HERMEZ-012

☐ **Stay up to date with the latest research on blockchain-based online voting and bidding.** Blockchain-based online voting is a known challenge. No perfect solution has been found yet. TOB-HERMEZ-002

☐ **When designing spam mitigation, consider that L1 gas cost can be avoided by Ethereum miners.** Otherwise, Ethereum miners can execute L1 transactions for free. TOB-HERMEZ-003, TOB-HERMEZ-004

☐ **Document the access controls and set up a proper authorization architecture.** Consider the risks associated with each access point and their frequency of usage to evaluate the proper design. TOB-HERMEZ-005

☐ **Identify and document all possible actions and their associated risks for privileged accounts.** Identifying the risks will assist codebase review and prevent future mistakes. TOB-HERMEZ-006

☐ **Always add sufficient logging to ensure users are aware of all state updates.** Events help to monitor the contracts. TOB-HERMEZ-008

☐ **Use Slither.** Slither catches several reported bugs. TOB-HERMEZ-009; TOB-HERMEZ-010, TOB-HERMEZ-013, TOB-HERMEZ-022

☐ **Properly document the inheritance schema of the contracts.** Use Slither's inheritance-graph printer to review the inheritance. TOB-HERMEZ-011

☐ **Carefully evaluate the value range of each variable stored, and ensure enough bits are conserved.** Collisions can happen if a variable has a large value but is stored with insufficient bits. TOB-HERMEZ-014

☐ **Write a specification of each function and thoroughly test it with unit tests and fuzzing. Use symbolic execution for arithmetic invariants.** TOB-HERMEZ-015, TOB-HERMEZ-016, TOB-HERMEZ-018, TOB-HERMEZ-019, TOB-HERMEZ-021

☐ **Always build on-chain signatures to be resilient in case of contract re-deployment or chain forks.** Attackers can re-use or front-run signatures to bypass authentication schema. TOB-HERMEZ-017

☐ **Identify the areas in the code that rely on external libraries and use an Ethereum development environment and NPM to manage packages as part of your project.** Relying on copy/paste for dependencies is error-prone. TOB-HERMEZ-020

Findings Summary

#	Title	Type	Severity
1	Lack of a contract existence check allows token theft	Data Validation	High
2	No incentive for bidders to vote earlier	Timing	Medium
3	L1 transactions spam	Timing	Low
4	Account creation spam	Data Validation	Low
5	Lack of access control separation is risky	Access Controls	High
6	Lack of two-step procedure for critical operations leaves them error-prone	Data Validation	High
7	Lack of a contract existence check in `TimeLock` allows incorrect assumption of code execution	Data Validation	Medium
8	Insufficient logging	Auditing and Logging	Low
9	No zero check on functions	Data Validation	Informational
10	Multiple contracts are missing inheritance	Undefined Behavior	Informational
11	Using empty functions instead of interfaces leave contract error-prone	Undefined Behavior	Informational
12	Initialization functions can be front-run	Configuration	High
13	Re-entrancy risks on `TokenHez`	Data Validation	High
14	`ChainId` usage can lead to collisions	Data Validation	Medium
15	Lack of overflow check on allocation ratio allows `AuctionProtocol` to be siphoned	Data Validation	High
16	Arithmetic rounding can lead `getMinBidBySlot` to return the current bid value	Data Validation	Low
17	`_checkSig` allows signature re-use	Cryptography	Medium

18	`changeDefaultSlotSetBid` allows the closed minimum bid of an open slot to be updated	Data Validation	Low
19	`cancelTransaction` can be called on non-queued transaction	Data Validation	Informational
20	Contracts used as dependencies do not track upstream changes	Patching	Low
21	Expected behavior regarding authorization for adding tokens is unclear	Access Controls	Informational
22	Contract name duplication leaves contracts error-prone	Undefined Behavior	Informational

1. Lack of a contract existence check allows token theft

Severity: High
Type: Data Validation
Target: `hermez/Hermez.sol`

Difficulty: Medium
Finding ID: TOB-HERMEZ-001

Description

Since there's no existence check for contracts that interact with external tokens, an attacker can steal funds by registering a token that's not yet deployed.

`_safeTransferFrom` transfers assets from a given token:

```
    /**
      * @dev transferFrom ERC20
      * Require approve tokens for this contract previously
      * @param token Token address
      * @param from Sender
      * @param to Reciever
      * @param value Quantity of tokens to send
      */
     function _safeTransferFrom(
         address token,
         address from,
         address to,
         uint256 value
     ) internal {
         (bool success, bytes memory data) = token.call(
             abi.encodeWithSelector(_TRANSFER_FROM_SIGNATURE, from, to, value)
         );
         require(
             success && (data.length == 0 || abi.decode(data, (bool))),
             "Hermez::_safeTransferFrom: ERC20_TRANSFERFROM_FAILED"
         );
     }
```

Figure 1.1: `hermez/Hermez.sol#L1059-L1080`.

The function does not check for the token code's existence. The Solidity documentation warns:

> *The low-level call, delegatecall and callcode will return success if the called account is non-existent, as part of the design of EVM. Existence must be checked prior to calling if desired.*

Figure 1.2: Solidity warning on existence check.

As a result, `_safeTransferFrom` will return success even if the token is not yet deployed, or was self-destructed.

An attacker that knows the address of a future token can register the token in Hermez, and deposit any amount prior to the token deployment. Once the contract is deployed and tokens have been deposited in Hermez, the attacker can steal the funds.

The address of a contract to be deployed can be determined by knowing the address of its deployer.

Exploit Scenario
Bob is about to deploy a new token. Eve knows the future address, lists the future token on Hermez, and deposits $1,000,000 worth of tokens. Bob deploys the token, uses Hermez, and deposits $1,000,000 worth of tokens. Eve steals all the funds.

Recommendations
Short term, check for contract existence in `_safeTransferFrom`. Add a similar check for any low-level calls, including in `WithdrawalDelayer`. This will prevent an attacker from listing and depositing tokens in a contract that is not yet deployed.

Long term, carefully review the Solidity documentation, especially the Warnings section.

2. No incentive for bidders to vote earlier

Severity: Medium
Difficulty: Low
Type: Timing
Finding ID: TOB-HERMEZ-002
Target: `auction/HermezAuctionProtocol.sol`

Description
Hermez relies on a voting system that allows anyone to vote with any weight at the last minute. As a result, anyone with a large fund can manipulate the vote.

Hermez’s voting mechanism relies on bidding. There is no incentive for users to bid tokens well before the voting ends. Users can bid a large amount of tokens just before voting ends, and anyone with a large fund can decide the outcome of the vote.

As all the votes are public, users bidding earlier will be penalized, because their bids will be known by the other participants. An attacker can know exactly how much currency will be necessary to change the outcome of the voting just before it ends.

Exploit Scenario
Alice bids 100 tokens. Eve bids 110 tokens one block before the end of the vote, so she decides the outcome of the vote without giving Alice a chance to win.

Recommendations
Short term, explore ways to incentivize users to vote earlier. Consider a weighted bid, with a weight decreasing over time. While it won’t prevent users with unlimited resources from manipulating the vote at the last minute, it will make the attack more expensive and reduce the chance of vote manipulation.

Long term, stay up to date with the latest research on blockchain-based online voting and bidding. Blockchain-based online voting is a known challenge. No perfect solution has been found yet.

3. L1 transaction spam

Severity: Low
Type: Timing
Target: `hermez/Hermez.sol`

Difficulty: High
Finding ID: TOB-HERMEZ-003

Description
The coordinator must be aware of all the L1 transactions to submit a rollup. An attacker can spam the network with invalidation transactions to prevent the submission of a rollup.

The documentation states:

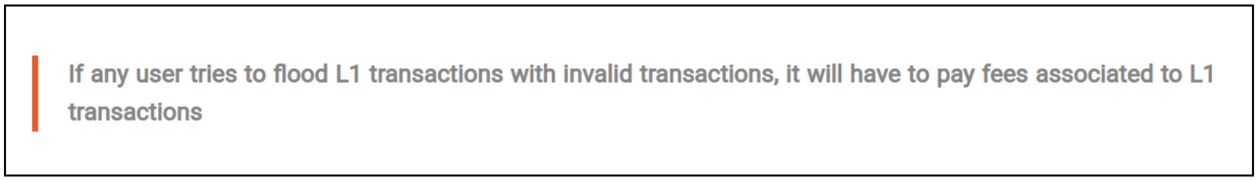

If any user tries to flood L1 transactions with invalid transactions, it will have to pay fees associated to L1 transactions

Figure 3.1: L1 transaction warning.

Ethereum miners don't have to pay for L1 transactions, so any miner can spam the Hermez contract to prevent the rollup from progressing.

Exploit Scenario
Bob is the coordinator. Eve is a miner. Eve sees Hermez as a threat that can reduce her earnings. Eve constantly spams the Hermez contract to prevent Bob's rollup from being accepted.

Recommendations
Short term, add a fee for any L1 transaction, to be paid in Hermez tokens. The fee will prevent Ethereum miners from spamming the contract for free.

Long term, when designing spam mitigation, consider that L1 gas cost can be avoided by Ethereum miners.

4. Account creation spam

Severity: Low
Type: Data Validation
Target: `hermez/Hermez.sol`

Difficulty: High
Finding ID: TOB-HERMEZ-004

Description
Hermez has a limit of 2**`MAX_NLEVELS` accounts. There is no fee on account creation, so an attacker can spam the network with account creation to fill the tree.

An account can be created with either an L1 user transaction or an L1 coordinator transaction.

The number of L1 user transactions that can be generated per second depends on the Ethereum network.

Multiple L1 coordinator transactions can be sent in one transaction by the coordinator. The number of L1 coordinator transactions that can be sent simultaneously depends on the Ethereum block number gas limit.

Similar to TOB-HERMEZ-003, Ethereum miners do not have to pay for account creation. Therefore, an Ethereum miner can spam the network with account creation by sending L1 user transactions (and L1 coordinator transactions, if the miner is also a coordinator).

If `MAX_NLEVELS` is below 32, an attacker can quickly reach the account limit. If `MAX_NLEVELS` is above or equal to 32, the time required to fill the tree will depend on the number of transactions accepted per second, but will take at least a couple of months.

Exploit Scenario
Eve is an Ethereum miner. She is constantly elected to be the Hermez coordinator, and always adds hundreds of L1 coordination transactions to create accounts. After a few months, no new accounts can be created.

Recommendations
Short term, add a fee for account creation or ensure `MAX_NLEVELS` is at least 32. Also, monitor account creation and alert the community if a malicious coordinator spams the system. This will prevent an attacker from spamming the system to prevent new accounts from being created.

Long term, when designing spam mitigation, consider that L1 gas cost can be avoided by Ethereum miners.

5. Lack of access control separation is risky

Severity: High
Difficulty: High
Type: Access Controls
Finding ID: TOB-HERMEZ-005
Target: `lib/InstantWithdrawManager.sol`

Description

The system uses the same account to change both frequently updated parameters and those that require less frequent updates. This architecture is error-prone and increases the severity of any privileged account compromises.

In `Hermez`, the governance DAO address can set multiple system parameters, including the:

- Fee for adding a token.
- L1L2 timeout.
- Bucket parameters.
- Tokens/USD value.

Additionally, if EMERGENCY_MODE is enabled in the protocol, the governance DAO can also withdraw all the funds that have been deposited in `WithdrawalDelayer`.

The tokens/USD value is updated through `updateTokenExchange`:

```
/**
 * @dev Update token USD value
 * @param addressArray Array of the token address
 * @param valueArray Array of USD values
 */
function updateTokenExchange(
    address[] memory addressArray,
    uint64[] memory valueArray
) external onlyGovernance {
    require(
        addressArray.length == valueArray.length,
        "InstantWithdrawManager::updateTokenExchange: INVALID_ARRAY_LENGTH"
    );
    for (uint256 i = 0; i < addressArray.length; i++) {
        tokenExchange[addressArray[i]] = valueArray[i];
    }
}
```

Figure 5.1: `hermez/lib/InstantWithdrawManager.sol#L177-L188`.

Among the functions callable by the governance DAO, only this function is meant to be called frequently, to keep the ratio tokens/USD up to date. This implies that the governance DAO will be in an environment that allows frequent interactions with the blockchain.

As a result, all the functions callable by the governance contract have a higher likelihood of compromise than they should.

Exploit Scenario
For Hermez' first deployment, the governance DAO is an EOA. Bob stores the key in its server. A set of scripts update `TokenExchange` every day. Eve compromises Bob's server and has access to the governance DAO account. Bob sees the compromise and calls the emergency mode. Eve uses the governance DAO account to drain all the funds from `WithdrawalDelayer`.

Recommendations
Short term, use a separate account to handle updating the tokens/USD ratio. Using the same account for the critical operations and update the tokens/USD ratio increases underlying risks.

Long term, document the access controls and set up a proper authorization architecture. Consider the risks associated with each access point and their frequency of usage to evaluate the proper design.

6. Lack of two-step procedure for critical operations leaves them error-prone

Severity: High
Difficulty: High
Type: Data Validation
Finding ID: TOB-HERMEZ-006
Target: `HermezAuctionProtocol.sol, WithdrawalDelayer.sol, Hermez.sol, Timelock.sol`

Description

Several critical operations are done in one function call. This schema is error-prone and can lead to irrevocable mistakes.

For example, the setter for the whitehack group address sets the address to the provided argument:

```
    /**
     * @notice Allows to change the `_whiteHackGroupAddress` if it's called by
 `_whiteHackGroupAddress`
     * @param newAddress new `_whiteHackGroupAddress`
     */
    function setWhiteHackGroupAddress(address payable newAddress) external {
        require(
            msg.sender == _whiteHackGroupAddress,
            "WithdrawalDelayer::setHermezGovernanceDAOAddress: ONLY_WHG"
        );
        _whiteHackGroupAddress = newAddress;
        emit NewWhiteHackGroupAddress(_whiteHackGroupAddress);
    }
```

Figure 6.1: `contracts/withdrawalDelayer/WithdrawalDelayer.sol#L134-L145`.

If the address is incorrect, the new address will take on the functionality of the new role immediately.

However, a two-step process is similar to the `approve-transferFrom` functionality: The contract approves the new address for a new role, and the new address acquires the role by calling the contract.

Functions that would benefit from a two-step procedure include:

- `WithdrawalDelayer`
    - `setHermezGovernanceDAOAddress`
    - `setHermezKeeperAddress`

    - `setWhiteHackGroupAddress`
- `HermezAuctionProtocol`
    - `setDonationAddress`
    - `setBootCoordinator`

Exploit Scenario
Alice deploys a new version of the whitehack group address. When she invokes the whitehack group address setter to replace the address, she accidentally enters the wrong address. The new address now has access to the role immediately and is too late to revert.

Recommendations
Short term, use a two-step procedure for all non-recoverable critical operations to prevent irrecoverable mistakes.

Long term, identify and document all possible actions and their associated risks for privileged accounts. Identifying the risks will assist codebase review and prevent future mistakes.

7. Lack of a contract existence check in `TimeLock` leads to incorrect assumption of code execution

Severity: Medium
Type: Data Validation
Target: `upgradability/Timelock.sol`

Difficulty: High
Finding ID: TOB-HERMEZ-007

Description
The lack of a contract existence check when executing transactions in `Timelock` might lead to incorrectly assuming that external code was executed.

`Timelock.executeTransaction` executes transactions that have passed the time lock:

```
(bool success, bytes memory returnData) = target.call.value(value)(
   callData
);
```

Figure 7.1: `upgradability/Timelock.sol#L188-L190`*.*

The function does not check for token code existence. The Solidity documentation warns:

The low-level call, delegatecall and callcode will return success if the called account is non-existent, as part of the design of EVM. Existence must be checked prior to calling if desired.

Figure 7.2: Solidity warning on existence check.

As a result, if a transaction is meant to execute code, and `callData` is non-empty, the transaction will succeed even if the destination has no code, or is destructed.

This behavior might lead the caller to assume the success of some operations, even though nothing was executed.

Exploit Scenario
Bob prepares a transaction to change a parameter in Hermez. He sets the destination incorrectly. After the time lock, Bob executes the transaction. The transaction is successful, and Bob does not notice that the parameter was not updated.

Recommendations
Short term, check for contract existence in `Timelock.executeTransaction` if `callData` is not empty. This will prevent the success of transactions that do not execute code.

Long term, carefully review the Solidity documentation, especially the Warnings section.

8. Insufficient Logging

Severity: Low
Type: Auditing and Logging
Target: `lib/InstantWithdrawManager.sol`

Difficulty: High
Finding ID: TOB-HERMEZ-008

Description
When system parameters such as withdrawal delays and bucket parameters are updated, these functions can be updated silently and do not alert users to updated values.

When the governance DAO updates the `withdrawlDelay` timeframe, it saves it to a local state variable:

```
    /**
     * @dev Update WithdrawalDelay
     * @param newWithdrawalDelay New WithdrawalDelay
     * Events: `UpdateWithdrawalDelay`
     */
   function updateWithdrawalDelay(uint64 newWithdrawalDelay)
       external
       onlyGovernance
   {
       require(
           newWithdrawalDelay <= _MAX_WITHDRAWAL_DELAY,
           "InstantWithdrawManager::updateWithdrawalDelay: EXCEED_MAX_WITHDRAWAL_DELAY"
       );
       withdrawalDelay = newWithdrawalDelay;
   }
```

Figure 8.1: `lib/InstantWithdrawManager.sol#L195-L204`.

As there is no event emitted, users are unaware of state changes that affect the Hermez Protocol.

This issue is currently present in:

- `InstantWithdrawManager`
  - `_initializeWithdraw`
  - `updateWithdrawDelay`
  - `updateBucketParameters`
  - `updateTokenExchange`
  - `safeMode`
- `HermezAuctionProtocol`
  - `hermezAuctionProtocolInitializer`

- `Hermez`
    - `initializeHermez`
    - `_initializeVerifiers`
- `WithdrawalDelayer`
    - `withdrawalDelayerInitializer`

Exploit Scenario
After submitting a withdrawal request, Alice tries to withdraw her funds once the withdrawal delay has passed. However, as the governance DAO has silently changed the parameter, she can't withdraw her funds yet.

Recommendations
Short term, add events in `InstantWithdrawManager` so users can easily identify when system parameters change. Without events, system monitoring is more difficult.

Long term, always add sufficient logging to ensure users are aware of all state updates.

9. Lack of zero check on functions

Severity: Informational
Difficulty: High
Type: Data Validation
Finding ID: TOB-HERMEZ-009
Target: `HermezAuctionProtocol.sol, WithdrawalDelayer.sol, Hermez.sol, Timelock.sol`

Description
Certain setter functions fail to validate incoming arguments, so callers can accidentally set important state variables to the zero address.

For example, the `setWhiteHackGroupAddress` function in the `WithdrawalDelayer` sets the address allowed to invoke an escape hatch withdrawal once the system has entered emergency mode:

```
    /**
    * @notice Allows to change the `_whiteHackGroupAddress` if it's called by
`_whiteHackGroupAddress`
    * @param newAddress new `_whiteHackGroupAddress`
    */
    function setWhiteHackGroupAddress(address payable newAddress) external {
        require(
            msg.sender == _whiteHackGroupAddress,
            "WithdrawalDelayer::setHermezGovernanceDAOAddress: ONLY_WHG"
        );
        _whiteHackGroupAddress = newAddress;
        emit NewWhiteHackGroupAddress(_whiteHackGroupAddress);
    }
```

Figure 9.1: `contracts/withdrawalDelayer/WithdrawalDelayer.sol#L134-L145`.

Once this address is set to `address(0)`, the `_whiteHackGroupAddress` cannot be reclaimed, as the address cannot call the setter function.

This issue is also prevalent in other contracts:

- `HermezAuctionProtocol.sol`
  - `hermezAuctionProtocolInitializer`
  - `setBootCoordinator`
- `WithdrawalDelayer.sol`
  - `withdrawalDelayerInitializer`
  - `setHermezGovernanceDAOAddress`
  - `setHermezKeeperAddress`
  - `setWhiteHackGroupAddress`

- `Hermez`
  - `initializeHermez`
- `Timelock`
  - `constructor`
  - `setPendingAdmin`
  - `executeTransaction`

Exploit Scenario
Alice deploys a new version of the whitehack group address. When she invokes the whitehack group address setter to replace the address, she accidentally enters the zero address, so the whitehack group address no longer exists.

Recommendations
Short term, add zero-value checks on all function arguments to ensure users can't accidentally set incorrect values. This will prevent incorrect configuration of the system.

Long term, use Slither, which will catch functions that do not have zero-checks.

10. Multiple contracts are missing inheritance

Severity: Informational
Difficulty: Low
Type: Undefined Behavior
Finding ID: TOB-HERMEZ-010
Target: `interfaces/WithdrawalDelayerInterface.sol`, `interfaces/AuctionInterface.sol`

Description
Several contracts implement interfaces but do not inherit from them. This behavior is error-prone and might prevent the implementation from following the interface if the code is updated.

The missing inheritances are:

- `WithdrawalDelayer` from `WithdrawalDelayerInterface`
- `HermezAuctionProtocol` from `AuctionInterface`

Exploit Scenario
`WithdrawalDelayerInterface` is updated, and one of its functions has a new signature. `WithdrawalDelayer` is not updated, so any external calls using `WithdrawalDelayerInterface` will fail.

Recommendations
Short term, ensure `WithdrawalDelayer` inherits from `WithdrawalDelayerInterface`, and `HermezAuctionProtocol` from `AuctionInterface`. This will ensure the contracts stay up to date with their interfaces.

Long term, use Slither, which will catch contracts missing inheritance.

11. Using empty functions instead of interfaces leaves contract error-prone

Severity: Informational
Difficulty: Low
Type: Undefined Behavior
Finding ID: TOB-HERMEZ-011
Target: `interfaces/WithdrawalDelayerInterface.sol`

Description

`WithdrawalDelayerInterface` is a contract meant to be an interface. It contains functions with empty bodies instead of function signatures, which might lead to unexpected behavior.

```
contract WithdrawalDelayerInterface {
     /**
      * @dev function to register a deposit in this smart contract, only the hermez
smart contract can do it
      * @param owner can claim the deposit once the delay time has expired
      * @param token address of the token deposited (0x0 in case of Ether)
      * @param amount deposit amount
      */
     function deposit(
         address owner,
         address token,
         uint192 amount
     ) public payable {}

     /**
      * @notice This function allows the HermezKeeperAddress to change the
withdrawal delay time, this is the time that
      * anyone needs to wait until a withdrawal of the funds is allowed. Since this
time is calculated at the time of
      * withdrawal, this change affects existing deposits. Can never exceed
`MAX_WITHDRAWAL_DELAY`
      * @dev It changes `_withdrawalDelay` if `_newWithdrawalDelay` it is less than
or equal to MAX_WITHDRAWAL_DELAY
      * @param _newWithdrawalDelay new delay time in seconds
      * Events: `NewWithdrawalDelay` event.
      */
     function changeWithdrawalDelay(uint64 _newWithdrawalDelay) external {}
```

Figure 11.1: `interfaces/WithdrawalDelayerInterface.sol#L5-L26`.

A contract inheriting from `WithdrawalDelayerInterface` will not require an override of these functions and will not benefit from the compiler checks on its correct interface.

Exploit Scenario

Bob creates a contract that inherits from `WithdrawalDelayerInterface.` He then creates a deposit that uses a `uint256` instead of `uint192`. The compiler does not flag the incorrect override, so Bob deploys the contract with the incorrect functions.

Recommendations
Short term, use an interface instead of a contract in `WithdrawalDelayerInterface`. This will make derived contracts follow the interface properly.

Long term, properly document the inheritance schema of the contracts. Use Slither's inheritance-graph printer to review the inheritance.

12. Initialization functions can be front-run

Severity: High
Difficulty: High
Type: Configuration
Finding ID: TOB-HERMEZ-012
Target: `contracts`

Description

`Hermez`, `HermezAuctionProtocol`, and `WithdrawalDelayer` have initialization functions that can be front-run, allowing an attacker to incorrectly initialize the contracts.

Due to the use of the `delegatecall` proxy pattern, `Hermez`, `HermezAuctionProtocol`, and `WithdrawalDelayer` cannot be initialized with a constructor, and have initializer functions:

```
    function initializeHermez(
         address[] memory _verifiers,
         uint256[] memory _verifiersParams,
         address _withdrawVerifier,
         address _hermezAuctionContract,
         address _tokenHEZ,
         uint8 _forgeL1L2BatchTimeout,
         uint256 _feeAddToken,
         address _poseidon2Elements,
         address _poseidon3Elements,
         address _poseidon4Elements,
         address _hermezGovernanceDAOAddress,
         address _safetyAddress,
         uint64 _withdrawalDelay,
         address _withdrawDelayerContract
     ) external initializer {
         // set state variables
         _initializeVerifiers(_verifiers, _verifiersParams);
         withdrawVerifier = VerifierWithdrawInterface(_withdrawVerifier);
         hermezAuctionContract = AuctionInterface(_hermezAuctionContract);
         tokenHEZ = _tokenHEZ;
         forgeL1L2BatchTimeout = _forgeL1L2BatchTimeout;
         feeAddToken = _feeAddToken;

         // set default state variables
         lastIdx = _RESERVED_IDX;
         // lastL1L2Batch = 0 --> first batch forced to be L1Batch
         // nextL1ToForgeQueue = 0 --> First queue will be forged
         nextL1FillingQueue = 1;
         // stateRootMap[0] = 0 --> genesis batch will have root = 0
         tokenList.push(address(0)); // Token 0 is ETH

         // initialize libs
         _initializeHelpers(
             _poseidon2Elements,
             _poseidon3Elements,
             _poseidon4Elements
         );
         _initializeWithdraw(
             _hermezGovernanceDAOAddress,
             _safetyAddress,
             _withdrawalDelay,
             _withdrawDelayerContract
         );
     }
```

Figure 12.1: hermez/Hermez.sol#L169-L213.

```
    function hermezAuctionProtocolInitializer(
        address token,
        uint128 genesis,
        address hermezRollupAddress,
        address governanceAddress,
        address donationAddress,
        address bootCoordinatorAddress
    ) public initializer {
        __ReentrancyGuard_init_unchained();
        _outbidding = 1000;
        _slotDeadline = 20;
        _closedAuctionSlots = 2;
        _openAuctionSlots = 4320;
        _allocationRatio = [4000, 4000, 2000];
        _defaultSlotSetBid = [
            INITIAL_MINIMAL_BIDDING,
            INITIAL_MINIMAL_BIDDING,
            INITIAL_MINIMAL_BIDDING,
            INITIAL_MINIMAL_BIDDING,
            INITIAL_MINIMAL_BIDDING,
            INITIAL_MINIMAL_BIDDING
        ];

        tokenHEZ = IHEZToken(token);
        require(
            genesis >= block.number + (BLOCKS_PER_SLOT * _closedAuctionSlots),
            "HermezAuctionProtocol::hermezAuctionProtocolInitializer
GENESIS_BELOW_MINIMAL"
        );
        genesisBlock = genesis;
        hermezRollup = hermezRollupAddress;
        _governanceAddress = governanceAddress;
        _donationAddress = donationAddress;
        _bootCoordinator = bootCoordinatorAddress;
    }
```

Figure 12.2: `auction/HermezAuctionProtocol.sol#L122-L155`.

```
function _initializeWithdraw(
    address _hermezGovernanceDAOAddress,
    address _safetyAddress,
    uint64 _withdrawalDelay,
    address _withdrawDelayerContract
 ) internal initializer {
    hermezGovernanceDAOAddress = _hermezGovernanceDAOAddress;
    safetyAddress = _safetyAddress;
    withdrawalDelay = _withdrawalDelay;
    withdrawDelayerContract = WithdrawalDelayerInterface(
        _withdrawDelayerContract
    );
 }
```
Figure 12.3: `lib/InstantWithdrawManager.sol#L56-L68`*.*

All these functions can be front-run by an attacker, allowing them to initialize the contracts with malicious values.

Exploit Scenario
Bob deploys the Hermez contracts. Eve front-runs the `HermezAuctionProtocol`'s initialization and sets her own address for the donation address. As a result, she receives the tokens sent as a donation.

Recommendations
Short term, either:

- Use a factory pattern that will prevent front-running of the initialization, or
- Ensure the deployment scripts are robust in case of a front-running attack.

Carefully review the Solidity documentation, especially the Warnings section. Carefully review the pitfalls of using `delegatecall` proxy pattern.

13. Re-entrancy risks on TokenHez

Severity: High
Type: Data Validation
Target: `Hermez.sol`

Difficulty: High
Finding ID: TOB-HERMEZ-013

Description

The Hermez codebase has many re-entrancy patterns that could be exploited if `TokenHez` had a callback mechanism.

Many interactions with `TokenHez` do not follow the check-effects-interactions pattern. If `TokenHez` had a callback mechanism, this would lead to exploitable re-entrancies.

For example, `addToken` allows the same token to be added twice through re-entrancy on `_safeTransferFrom`:

```
function addToken(address tokenAddress, bytes calldata permit) public {
     uint256 currentTokens = tokenList.length;
     require(
         currentTokens < _LIMIT_TOKENS,
         "Hermez::addToken: TOKEN_LIST_FULL"
     );
     require(
         tokenAddress != address(0),
         "Hermez::addToken: ADDRESS_0_INVALID"
     );
     require(tokenMap[tokenAddress] == 0, "Hermez::addToken: ALREADY_ADDED");

     // permit and transfer HEZ tokens
     if (permit.length != 0) {
         _permit(tokenHEZ, feeAddToken, permit);
     }
     _safeTransferFrom(tokenHEZ, msg.sender, address(this), feeAddToken);

     tokenList.push(tokenAddress);
     tokenMap[tokenAddress] = currentTokens;

     emit AddToken(tokenAddress, uint32(currentTokens));
```

Figure 13.1: `hermez/Hermez.sol#L640-L661`.

Exploit Scenario

`TokenHez` is a ERC-20 token with a callback mechanism. `TokenHez` is deployed. Eve exploits the re-entrancy in `addToken` to add the same token twice.

Recommendations

Short term, either ensure `TokenHez` does not have callback features, or follow the check-effects-interactions pattern.

Long term, use Slither, which will detect the potential re-entrancies in the codebase.

References:

- imBTC re-entrancy attack

14. ChainId usage can lead to collisions

Severity: Medium
Type: Data Validation
Target: Hermez.sol

Difficulty: High
Finding ID: TOB-HERMEZ-014

Description
The circuit stores only two bytes from chainID. This can lead to collisions if a chainId is above 65,535.

The _constructCircuitInput function stores chainID at the end of the inputBytes, and only keeps two bytes:

```
// store 2 bytes of chainID at the end of the inputBytes
assembly {
   mstore(ptr, shl(240, chainid())) // 256 - 16 = 240
}
```

Figure 14.1: Hermez.sol#L949-L952.

Many existing forks have a chainID value above 65,535, so the chainId used in Hermez can lead to collisions between forks.

Exploit Scenario
Eve deploys Hermez on a fork with a chainID of 65,536. As a result, the circuit's inputs in the mainnet and on the fork are equivalent.

Recommendations
Short term, either document that Hermez must be deployed on a fork with a chainID below 65,535, or revert if chainID is above 65,535.

Long term, carefully evaluate the value range of each variable stored, and ensure enough bits are conserved.

15. Lack of overflow check on allocation ratio allows `AuctionProtocol` to be siphoned

Severity: High
Difficulty: High
Type: Data Validation
Finding ID: TOB-HERMEZ-015
Target: `auction/HermezAuctionProtocol.sol`

Description
Failing to check the allocation ratio overflow allows an attacker with access to the governance/donation address to siphon all HEZ out of the `AuctionProtocol`.

The `AuctionProtocol` defines an allocation ratio that sets the percentage of funds that will be burnt, sent to the donation address, and recirculated into the Hermez Protocol. The allocation ratio has the following setter function:

```
     /**
      * @notice Allows to change the `_allocationRatio` array if it's called by the
owner
      * @param newAllocationRatio new `_allocationRatio` uint8[3] array
      * Events: `NewAllocationRatio`
      */
    function setAllocationRatio(uint16[3] memory newAllocationRatio)
         external
         onlyGovernance
     {
         require(
             (newAllocationRatio[0] +
                 newAllocationRatio[1] +
                 newAllocationRatio[2]) == 10000,
             "HermezAuctionProtocol::setAllocationRatio:
ALLOCATION_RATIO_NOT_VALID"
         );
         _allocationRatio = newAllocationRatio;
         emit NewAllocationRatio(_allocationRatio);
     }
```

Figure 15.1: `auction/HermezAuctionProtocol.sol#L262-L279`.

The `require` statement uses integer addition and fails to account for integer overflow. Later, these ratios are used to distribute tokens when a new bid is forged:

```
     // We save the minBid that this block has had
      slots[slotToForge].closedMinBid = slots[slotToForge].bidAmount;

      // calculation of token distribution
     uint128 burnAmount = slots[slotToForge]
```

```
            .bidAmount
            .mul(_allocationRatio[0])
            .div(uint128(10000)); // Two decimal precision
        uint128 donationAmount = slots[slotToForge]
            .bidAmount
            .mul(_allocationRatio[1])
            .div(uint128(10000)); // Two decimal precision
        uint128 governanceAmount = slots[slotToForge]
            .bidAmount
            .mul(_allocationRatio[2])
            .div(uint128(10000)); // Two decimal precision
        // Tokens to burn
        tokenHEZ.burn(burnAmount);
        // Tokens to donate
        pendingBalances[_donationAddress] = pendingBalances[_donationAddress]
           .add(donationAmount);
        // Tokens for the governace address
        pendingBalances[_governanceAddress] = pendingBalances[_governanceAddress]
            .add(governanceAmount);
```

Figure 15.2: `auction/HermezAuctionProtocol.sol#L755-L787`.

The calculations above change the `pendingBalances` mapping, which keeps track of funds per user. The value in this mapping is then used to allow a user to withdraw their HEZ:

```
        /**
        * @notice function to know how much HEZ tokens are pending to be claimed for
an address
        * @param bidder address to query
        * @return the total claimable HEZ by an address
        */
        function getClaimableHEZ(address bidder) public view returns (uint128) {
            return pendingBalances[bidder];
        }

        /**
         * @notice distributes the tokens to msg.sender address
         * Events: `HEZClaimed`
```

```
    */
    function claimHEZ() public nonReentrant {
       uint128 pending = getClaimableHEZ(msg.sender);
       require(
          pending > 0,
          "HermezAuctionProtocol::claimHEZ: NOT_ENOUGH_BALANCE"
      );
      pendingBalances[msg.sender] = 0;
      require(
          tokenHEZ.transfer(msg.sender, pending),
          "HermezAuctionProtocol::claimHEZ: TOKEN_TRANSFER_FAILED"
      );
       emit HEZClaimed(msg.sender, pending);
     }
```
Figure 15.3: `auction/HermezAuctionProtocol.sol#L792-L817`*.*

The failure to account for the allocation ratio's overflow causes the pending balance to be incorrectly assigned for addresses. This allows the donation/governance address to withdraw all funds from the `AuctionProtocol` contract.

Exploit Scenario
The first version of Hermez is deployed with a governance DAO address set to an EOA account. Eve, an attacker, gains access to the governance DAO address and replaces the governance address with her own address. She also sets the allocation ratio to [1, 10,000, 65,535], which passes the `require` statement due to integer overflow. Eve submits and wins a bid with a bid amount of 10,000. 1 HEZ gets burnt, the pending balance of the `donationAmount` increases by 10,000, and the `governanceAmount` increases by 65,535. Eve can then steal 65,535 HEZ held by the `AuctionProtocol`.

Recommendations
Short term, add checks to ensure that each allocation ratio index is < 10,000. This will protect against overflow.

Long term, write a specification of each function and thoroughly test it with unit tests and fuzzing. Use symbolic execution for arithmetic invariants.

16. Arithmetic rounding allows `getMinBidBySlot` to return the current bid value

Severity: Low
Type: Data Validation
Target: `auction/HermezAuctionProtocol.sol`

Difficulty: Low
Finding ID: TOB-HERMEZ-016

Description
An arithmetic rounding error in `getMinBidBySlot` allows the function to return the current bid value without the expected increase.

`getMinBidBySlot` returns the added value for a slot, plus an minimal increase:

```
        (slots[slot].bidAmount == 0)
             ? _defaultSlotSetBid[slotSet].add(
                 _defaultSlotSetBid[slotSet].mul(_outbidding).div(
                     uint128(10000) // two decimal precision
                 )
             )
             : slots[slot].bidAmount.add(
                 slots[slot].bidAmount.mul(_outbidding).div(uint128(10000)) // two
decimal precision
             );
```

Figure 16.1: `auction/HermezAuctionProtocol.sol#L432-L440`.

The formula is:

```
bid_amount + bid_amount * outbidding / 10000
```

...where `bid_amount` is either the current bid amount or the default amount for the slot. `outbidding` is set by the contract's deployer and the governance—its default value is 1,000.

If `bid_amount * outbidding` is below 10,000, the increase will be zero. So, `getMinBidBySlot` will return the current value, and allow anyone to outbid without paying the additional increase.

We classified this issue as low severity because it can only impact bids with really low value, and outbid users will have the opportunity to continue the bid process.

Exploit Scenario
Bob bids 9 on a slot. Eve bids 9 on the same slot and becomes the winning bidder.

Recommendations

Short term, either add +1 to the value returned by `getMinBidBySlot`, or document that the bidding increase will not work for a low bid value. Otherwise, an arithmetic rounding in `getMinBidBySlot` can make the function return the current bid value without the expected increase.

Long term, write a specification of each function and thoroughly test it with unit tests and fuzzing. Use symbolic execution for arithmetic invariants.

17. `_checkSig` allows signature re-use

Severity: Medium
Type: Cryptography
Target: `lib/HermezHelpers.sol`

Difficulty: High
Finding ID: TOB-HERMEZ-017

Description

`_checkSig` allows signature re-use across multiple contract deployments and forks. This means an attacker can re-use an association of `babyjub` with an Ethereum address in multiple Hermez instances.

`_checkSig` checks that a `babyjub` address was signed by an Ethereum address:

```
     bytes32 messageDigest = keccak256(
          abi.encodePacked(
              "\x19Ethereum Signed Message:\n98", // 98 bytes --> 66 bytes (string
message) + 32 bytes (babyjub)
              "I authorize this babyjubjub key for hermez rollup account creation",
              babyjub
          )
      );
      address ethAddress = ecrecover(messageDigest, v, r, s);
```

Figure 17.1: `lib/HermezHelpers.sol#L226-L233`.

The signature schema lacks signatures re-use protection, including:

- The contract address, to prevent re-use across deployments.
- The `chainId`, to prevent re-use across forks.

This means an attacker can re-use a signature in different contract instances.

Exploit Scenario

- Eve forks Hermez on a new contract.
- Eve asks Bob to test Hermez on the newly deployed contract. She creates the private key for Bob's `babyjub` account; since it's a test contract, Bob doesn't worry.
- Eve creates the same account on the original Hermez contract
- Eve tricks Alice into sending funds to the `babyjub` account—Alice believes she's sending the asset to Bob.

Recommendations

Short term, add signature re-use mitigations in `_checkSig` against both contract re-deployment and forks. This ensures that signatures can't be replayed across different versions.

Long term, always build on-chain signatures to be resilient in case of contract re-deployment or chain forks. Attackers can re-use or front-run signatures to bypass authentication schema.

18. `changeDefaultSlotSetBid` allows the closed minimum bid of an open slot to be updated

Severity: Low
Difficulty: High
Type: Data Validation
Finding ID: TOB-HERMEZ-018
Target: `auction/HermezAuctionProtocol.sol`

Description

`changeDefaultSlotSetBid` updates the minimum bid value for a slot but prevents the change for closed bids. The function's loop incorrectly iterates over one additional slot, preventing the update of one open bid.

`changeDefaultSlotSetBid` takes a `slotSet`, a new minimum bid value. To preserve the expected minimum bid value of closed bids, their current default minimum bid is saved:

```
        uint128 current = getCurrentSlotNumber();
         // This prevents closed bids from being modified
         for (uint128 i = current; i <= current + _closedAuctionSlots; i++) {
             // Save the minbid in case it has not been previously set
             if (slots[i].closedMinBid == 0) {
                 slots[i].closedMinBid = _defaultSlotSetBid[getSlotSet(i)];
             }
```

Figure 18.1: `auction/HermezAuctionProtocol.sol#L356-L362`.

The loop iterates on `current + closedAuctionSlots`, where the last element is an open bid:

```
        require(
             slot >= (getCurrentSlotNumber() + _closedAuctionSlots),
             "HermezAuctionProtocol::processBid: AUCTION_CLOSED"
         );
```

Figure 18.2: `auction/HermezAuctionProtocol.sol#L462-L465`.

As a result, `changeDefaultSlotSetBid` incorrectly sets the closed minimum bid value of an open bid.

Exploit Scenario

Bob wants to update the default minimum value of the bid that is about to be closed. Bob calls `changeDefaultSlotSetBid`, but the function does not update the value correctly.

Recommendations

Short term, iterate over `i < current + _closedAuctionSlots` in `changeDefaultSlotSetBid`. The loop currently iterates over one open bid.

Long term, write a specification of each function and thoroughly test it with unit tests and fuzzing. Use symbolic execution for arithmetic invariants.

19. `cancelTransaction` can be called on non-queued transaction

Severity: Informational
Type: Data Validation
Target: `upgradability/Timelock.sol`

Difficulty: Low
Finding ID: TOB-HERMEZ-019

Description
Without a transaction existence check in `cancelTransaction`, an attacker can confuse monitoring systems.

`cancelTransaction` emits an event without checking that the transaction to be canceled exists:

```
function cancelTransaction(
     address target,
     uint256 value,
     string memory signature,
     bytes memory data,
     uint256 eta
 ) public {
     require(
         msg.sender == admin,
         "Timelock::cancelTransaction: Call must come from admin."
     );

     bytes32 txHash = keccak256(
         abi.encode(target, value, signature, data, eta)
     );
     queuedTransactions[txHash] = false;

     emit CancelTransaction(txHash, target, value, signature, data, eta);
 }
```

Figure 19.1: `upgradability/Timelock.sol#L126-L144`*.*

This allows a malicious admin to confuse monitoring systems by generating malicious events.

Recommendations
Short term, check that the transaction to be canceled exists in `cancelTransaction`. This will ensure that monitoring tools can rely on emitted events.

Long term, write a specification of each function and thoroughly test it with unit tests and fuzzing. Use symbolic execution for arithmetic invariants.

20. Contracts used as dependencies do not track upstream changes

Severity: Low
Difficulty: High
Type: Patching
Finding ID: TOB-HERMEZ-020
Target: `lib/HermezHelpers.sol`

Description
Third-party contracts like `_concatStorage` are pasted into the Hermez repository. Moreover, the code documentation does not specify the exact revision used, or if it is modified. This makes updates and security fixes on these dependencies unreliable since they must be updated manually.

`_concatStorage` is borrowed from the `solidity-bytes-utils` library, which provides helper functions for byte-related operations. Recently, a critical vulnerability was discovered in the library's slice function which allows arbitrary writes for user-supplied inputs:

```
There was a critical bug in the slice method, reported on an audit to a DXDao
codebase.

Previously, no checks were being made on overflows of the _start and _length
parameters since previous reviews of the codebase deemed this overflow
"unexploitable" because of an inordinate expansion of memory (i.e., reading an
immensely large memory offset causing huge memory expansion) resulting in an
out-of-gas exception.

However, as noted in the review mentioned above, this is not the case. The slice
method in versions <=0.9.0 actually allows for arbitrary kind of (i.e., it allows
memory writes to very specific values) arbitrary memory writes _in the specific
case where these parameters are user-supplied inputs and not hardcoded values
(which is uncommon).
```

Figure 20.1: `solidity-bytes-utils changelog`.

Exploit Scenario
A third-party contract used in Hermez receives an update with a critical fix for a vulnerability. An attacker detects the use of a vulnerable contract and exploits the vulnerability against Hermez.

Recommendations

Short term, review the codebase and document each dependency’s source and version. Include the third-party sources as submodules in your Git repository so internal path consistency can be maintained and dependencies are updated periodically.

Long term, identify the areas in the code that are relying on external libraries and use an Ethereum development environment and NPM to manage packages as part of your project.

21. Expected behavior regarding authorization for adding tokens is unclear

Severity: Informational
Type: Access Controls
Target: `hermez/Hermez.sol`

Difficulty: Low
Finding ID: TOB-HERMEZ-021

Description

addToken allows anyone to list a new token on Hermez. This contradicts the online documentation, which implies that only the governance should have this authorization.

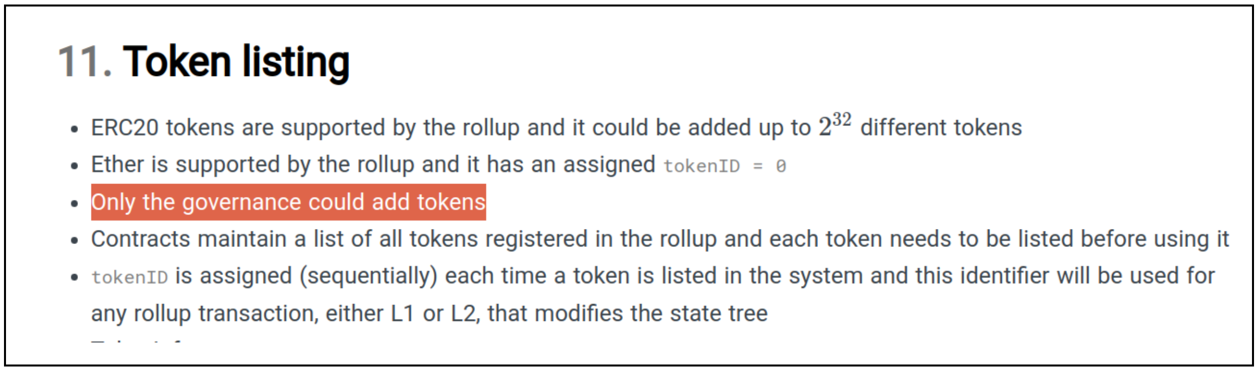
```
    /**
      * @dev Inclusion of a new token to the rollup
      * @param tokenAddress Smart contract token address
      * Events: `AddToken`
      */
     function addToken(address tokenAddress, bytes calldata permit) public {
         uint256 currentTokens = tokenList.length;
         require(
             currentTokens < _LIMIT_TOKENS,
             "Hermez::addToken: TOKEN_LIST_FULL"
         );
         require(
             tokenAddress != address(0),
             "Hermez::addToken: ADDRESS_0_INVALID"
         );
         require(tokenMap[tokenAddress] == 0, "Hermez::addToken: ALREADY_ADDED");

         // permit and transfer HEZ tokens
         if (permit.length != 0) {
             _permit(tokenHEZ, feeAddToken, permit);
         }
         _safeTransferFrom(tokenHEZ, msg.sender, address(this), feeAddToken);

         tokenList.push(tokenAddress);
         tokenMap[tokenAddress] = currentTokens;

         emit AddToken(tokenAddress, uint32(currentTokens));
     }
```

Figure 21.1: `hermez/Hermez.sol#L635-L662`.



11. **Token listing**

- ERC20 tokens are supported by the rollup and it could be added up to 2^{32} different tokens
- Ether is supported by the rollup and it has an assigned `tokenID = 0`
- Only the governance could add tokens
- Contracts maintain a list of all tokens registered in the rollup and each token needs to be listed before using it
- `tokenID` is assigned (sequentially) each time a token is listed in the system and this identifier will be used for any rollup transaction, either L1 or L2, that modifies the state tree

Figure 21.2: `token-listing`.

It is unclear whether the implementation or the documentation is correct.

Recommendations
Short term, update either the implementation or the documentation to standardize the authorization specification for adding tokens.

Long term, write a specification of each function and thoroughly test it with unit tests and fuzzing. Use symbolic execution for arithmetic invariants.

22. Contract name duplication leaves codebase error-prone

Severity: Informational
Difficulty: Low
Type: Undefined Behavior
Finding ID: TOB-HERMEZ-022
Target: `contracts`

Description
The codebase has multiple contracts that share the same name. This allows `buidler-waffle` to generate incorrect `json` artifacts, preventing third parties from using their tools.

`Buidler-waffle` does not correctly support a codebase with duplicate contract names. The compilation overwrites compilation artifacts and prevents the use of third-party tools, such as Slither.

The contracts that have duplicate names are:

- `SafeMath`
- `IERC20`
- `Initializable`

Recommendations
Short term, prevent the re-use of duplicate contract names or change the compilation framework.

Long term, use Slither, which will help detect duplicate contract names.

References
- Similar issue in truffle: `https://github.com/trufflesuite/truffle/issues/3124`

A. Vulnerability Classifications

Vulnerability Classes	
Class	**Description**
Access Controls	Related to the authorization of users and assessment of rights
Auditing and Logging	Related to auditing of actions or logging of problems
Authentication	Related to the identification of users
Configuration	Related to security configurations of servers, devices, or software
Cryptography	Related to protecting the privacy or integrity of data
Data Exposure	Related to unintended exposure of sensitive information
Data Validation	Related to improper reliance on the structure or values of data
Denial of Service	Related to causing system failure
Error Reporting	Related to the reporting of error conditions in a secure fashion
Patching	Related to keeping software up to date
Session Management	Related to the identification of authenticated users
Testing	Related to test methodology or test coverage
Timing	Related to race conditions, locking, or order of operations
Undefined Behavior	Related to undefined behavior triggered by the program

Severity Categories	
Severity	**Description**
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or Defense in Depth
Undetermined	The extent of the risk was not determined during this engagement
Low	The risk is relatively small or is not a risk the customer has indicated is important

Medium	Individual user’s information is at risk, exploitation would be bad for client’s reputation, moderate financial impact, possible legal implications for client
High	Large numbers of users, very bad for client’s reputation, or serious legal or financial implications

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploit was not determined during this engagement
Low	Commonly exploited, public tools exist or can be scripted that exploit this flaw
Medium	Attackers must write an exploit, or need an in-depth knowledge of a complex system
High	The attacker must have privileged insider access to the system, may need to know extremely complex technical details, or must discover other weaknesses in order to exploit this issue

B. Code Maturity Classifications

Code Maturity Classes	
Category Name	**Description**
Access Controls	Related to the authentication and authorization of components.
Arithmetic	Related to the proper use of mathematical operations and semantics.
Assembly Use	Related to the use of inline assembly.
Centralization	Related to the existence of a single point of failure.
Upgradeability	Related to contract upgradeability.
Function Composition	Related to separation of the logic into functions with clear purpose.
Front-Running	Related to resilience against front-running.
Key Management	Related to the existence of proper procedures for key generation, distribution, and access.
Monitoring	Related to use of events and monitoring procedures.
Specification	Related to the expected codebase documentation.
Testing & Verification	Related to the use of testing techniques (unit tests, fuzzing, symbolic execution, etc.).

Rating Criteria	
Rating	**Description**
Strong	The component was reviewed and no concerns were found.
Satisfactory	The component had only minor issues.
Moderate	The component had some issues.
Weak	The component led to multiple issues; more issues might be present.
Missing	The component was missing.

Not Applicable	The component is not applicable.
Not Considered	The component was not reviewed.
Further Investigation Required	The component requires further investigation.

C. Code Quality Recommendations

The following recommendations are not associated with specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future.

Smart contracts

- **Ensure getter return variables are the same as actual variables to prevent type mismatch on off-chain components.** `WithdrawalDelayer`'s `_withdrawalDelay` and `_emergencyModeStartingTime` are `uint64,` and the getter returns a `uint128`.
- **Uniformly declare and identify function signatures.** Function hashes in `Hermez.sol` define its function calls in byte form, whereas hashes in `WithdrawalDelayer` define them by hashing bytes. For readability and maintainability, these should be made uniform.
- **Wrap all external calls in a `require`.** In `AuctionProtocol`, #L812-815 wraps a HEZ token transfer in a `require` to ensure the call is successful. However, in #L770-771, the operation to burn tokens is not successful. Ensure uniformness in retrieving the result of all external calls.
- **Implement a minimal amount for each rate.** Adding minimum amounts on setter functions for allocation ratios, token prices, or amounts protects the system from accidentally providing incorrect arguments.
- **Remove the call to `_fillZeros(ptr, feeIdxCoordinatorLength - dLen)` `(Hermez.sol#L946)` and the pointer increase `ptr += feeIdxCoordinatorLength - dLen (Hermez.sol#L947)`.** `dLen` is always equal to `feeIdxCoordinatorLength`, so these operations can be removed to save gas.

D. Inconsistencies Between Code and Specification

When reviewing the circom files, we discovered a number of instances in which the specification does not reflect the behavior of the code. We believe these are largely mistakes or typos in the specification. They should not have an immediate impact on the system's security, but we encourage you to fix them to ensure the specification accurately reflects the code.

- Padding checks are performed on various input signals, such as IDs, but these checks aren't described in the specification.
- The schematic in `balance-updater` has an error in the calculation of `effectiveAmount2`. The `nullifyAmount` should be replaced with `!nullifyAmount`.
- The schematic in `rollup-tx-states` has the following errors:
    - There are two figures describing the computation of `checkToEthAddr` and `checkToBjj`. The top figures should be removed.
    - The `nop` value should be replaced with `!nop` in the computation of both `checkToEthAddr` and `checkToBjj`.
    - The mux for `key2` should include one `finalFromIdx` and one `finalToIdx`, instead of two `finalFromIdx` values.
    - Both `isAmount` and `isLoadAmount` are included as intermediate signals in the schematic, but the schematic does not indicate how these values are derived.
- The schematic in `rollup-tx` has the following errors:
    - In part B, the second `futureToBjj` should be `pastToBjj`.
    - In part C, the "equal if enabled" diagram for `toEthAddr` and `ethAddr2` should also include `checkToBjj`.
    - In part E, the second `s1Nonce` value should be `s1OldValue`.
    - In part K, the `P1_newRoot` and `P2_newRoot` inputs should be flipped.
- The schematic in `rollup-main` is missing `toIdx` in the diagram for part D.

E. Token Integration Checklist

The following checklist provides recommendations for interacting with arbitrary tokens. Every unchecked item should be justified and its associated risks understood. An up-to-date version of the checklist can be found in `crytic/building-secure-contracts`.

For convenience, all Slither utilities can be run directly on a token address, such as:

```
slither-check-erc 0xdac17f958d2ee523a2206206994597c13d831ec7 TetherToken
```

To follow this checklist, you'll want to have this output from Slither for the token:

```
- slither-check-erc [target] [contractName] [optional: --erc ERC_NUMBER]
- slither [target] --print human-summary
- slither [target] --print contract-summary
- slither-prop . --contract ContractName # requires configuration, and use of
Echidna and Manticore
```

General security considerations

- ❏ **The contract has a security review.** Avoid interacting with contracts that lack a security review. Check the length of the assessment (aka "level of effort"), the reputation of the security firm, and the number and severity of the findings.
- ❏ **You have contacted the developers.** You may need to alert their team to an incident. Look for appropriate contacts on blockchain-security-contacts.
- ❏ **They have a security mailing list for critical announcements.** Their team should advise users (like you!) when critical issues are found or when upgrades occur.

ERC conformity

Slither includes a utility, `slither-check-erc`, that reviews the conformance of a token to many related ERC standards. Use slither-check-erc to review that:

- ❏ **`Transfer` and `transferFrom` return a boolean.** Several tokens do not return a boolean on these functions, so their calls in the contract might fail.
- ❏ **The `name`, `decimals`, and `symbol` functions are present if used.** These functions are optional in the ERC20 standard and might not be present.
- ❏ **`Decimals` returns a `uint8`.** Several tokens incorrectly return a `uint256`. If this is the case, ensure the value returned is below 255.

- ❏ **The token mitigates the known ERC20 race condition.** The ERC20 standard has a known ERC20 race condition that must be mitigated to prevent attackers from stealing tokens.
- ❏ **The token is not an ERC777 token and has no external function call in `transfer` and `transferFrom`.** External calls in the transfer functions can lead to re-entrancies.

Slither includes a utility, `slither-prop`, which generates unit tests and security properties that can discover many common ERC flaws. Use `slither-prop` to confirm that:

- ❏ **The contract passes all unit tests and security properties from `slither-prop`.** Run the generated unit tests, then check the properties with Echidna and Manticore.

Finally, there are certain characteristics that are difficult to identify automatically. Search for these conditions manually:

- ❏ **`Transfer` and `transferFrom` should not take a fee.** Deflationary tokens can lead to unexpected behavior.
- ❏ **Potential interest earned from the token is taken into account.** Some tokens distribute interest to token holders. This interest might be trapped in the contract if not taken into account.

Contract composition

- ❏ **The contract avoids unnecessary complexity.** The token should be a simple contract; a token with complex code requires a higher standard of review. Use Slither's `human-summary` printer to identify complex code.
- ❏ **The contract uses `SafeMath`.** Contracts that do not use `SafeMath` require a higher standard of review. Inspect the contract manually for `SafeMath` usage.
- ❏ **The contract has only a few non–token-related functions.** Non–token-related functions increase the likelihood of an issue in the contract. Use Slither's `contract-summary` printer to broadly review the code used in the contract.
- ❏ **The token only has one address.** Tokens with multiple entry points for balance updates can break internal bookkeeping based on the address (e.g., `balances[token_address][msg.sender]` might not reflect the actual balance).

Owner privileges

- ❏ **The token is not upgradeable.** Upgradeable contracts might change their rules over time. Use Slither's `human-summary` printer to determine if the contract is upgradeable.
- ❏ **The owner has limited minting capabilities.** Malicious or compromised owners can abuse minting capabilities. Use Slither's `human-summary` printer to review minting capabilities, and consider manually reviewing the code.

- ❏ **The token is not pausable.** Malicious or compromised owners can trap contracts relying on pausable tokens. Identify pauseable code manually.
- ❏ **The owner cannot blacklist the contract.** Malicious or compromised owners can trap contracts relying on tokens with a blacklist. Identify blacklisting features manually.
- ❏ **The team behind the token is known and can be held responsible for abuse.** Contracts with anonymous development teams, or teams that reside in legal shelters, should require a higher standard of review.

Token scarcity

Finding token scarcity issues requires manual review. Check for these conditions:

- ❏ **No user owns most of the supply.** If a few users own most of the tokens, they can influence operations based on the token's repartition.
- ❏ **The total supply is sufficient.** Tokens with a low total supply can be easily manipulated.
- ❏ **The tokens are located in more than a few exchanges.** If all the tokens are in one exchange, a compromise of the exchange can compromise the contract relying on the token.
- ❏ **Users understand the associated risks of large funds or flash loans.** Contracts relying on the token balance must carefully take into consideration attackers with large funds or flash loan attacks.
- ❏ **The token does not allow flash minting.** Flash minting can lead to substantial swings in the balance and the total supply, which necessitate strict and comprehensive overflow checks in the operation of the token.

F. Fix Log

Hermez Network asked us to retest the codebase after addressing issues described in this report. Each of the fixes provided by Hermez Network was checked by Trail of Bits on the week of December 7th.

#	Title	Type	Severity	Difficulty	Status
1	Lack of a contract existence check allows token theft	Data Validation	High	Medium	Fixed (10)
2	No incentive for bidders to vote earlier	Timing	Medium	Low	Fixed (24)
3	L1 transactions spam	Timing	Low	High	Not fixed
4	Account creation spam	Data Validation	Low	High	Not fixed
5	Lack of access control separation is risky	Access Controls	High	High	Fixed
6	Lack of two-step procedure for critical operations leaves them error-prone	Data Validation	High	High	Fixed
7	Lack of a contract existence check in `TimeLock` allows incorrect assumption of code execution	Data Validation	Medium	High	Not fixed
8	Insufficient logging	Auditing and Logging	Low	High	Fixed (14)
9	No zero check on functions	Data Validation	Informational	High	Not fixed
10	Multiple contracts are missing inheritance	Undefined Behavior	Informational	Low	Fixed (15)
11	Using empty functions instead of interfaces leave contract error-prone	Undefined Behavior	Informational	Low	Fixed (15)

12	Initialization functions can be front-run	Configuration	High	High	Partially fixed
13	Re-entrancy risks on `TokenHez`	Data Validation	High	High	Not fixed
14	`ChainId` usage can lead to collisions	Data Validation	Medium	High	Not fixed
15	Lack of overflow check on allocation ratio allows `AuctionProtocol` to be siphoned	Data Validation	High	High	Fixed (19)
16	Arithmetic rounding can lead `getMinBidBySlot` to return the current bid value	Data Validation	Low	Low	Fixed (23)
17	`_checkSig` allows signature re-use	Cryptography	Medium	High	Fixed (20)
18	`changeDefaultSlotSetBid` allows the closed minimum bid of an open slot to be updated	Data Validation	Low	High	Fixed (21)
19	`cancelTransaction` can be called on non-queued transaction	Data Validation	Informational	Low	Not fixed
20	Contracts used as dependencies do not track upstream changes	Patching	Low	High	Not fixed
21	Expected behavior regarding authorization for adding tokens is unclear	Access Controls	Informational	Low	Fixed
22	Contract name duplication leaves contracts error-prone	Undefined Behavior	Informational	Low	Fixed (41, 42)

Detailed fix log

3. L1 transactions spam - Not fixed
Hermez team stated:

> *Spam the network with L1 transactions will have a gas cost for the spamer. In case of a Miner, the miner will have to pay an opportunity cost.*
>
> *In any case the spamer will only delay the legit L1 Txs some blocks, but sooner or later those transactions will be forged.*
> *Important to note that the rollup will never stop because once the queue is filled with 128TX, the next TX will be stored in the next queue.*

4. Account creation spam - Not fixed
Hermez team stated:

> *As specified in the recommendation of the issue, there will be more than 32 levels*

5. Lack of access control separation is risky - Fixed
Hermez will use a governance contract to control the system. Hermez team stated:

> *We will create an external smart contract that will handle the access control of the governance.*
> *This smart contract may be updated by the same governance in order to update the operation needs.*
> *Fix: https://github.com/hermeznetwork/contracts/pull/28*
> *Acces control smart contract:*
> *https://github.com/hermeznetwork/contracts/pull/28/files#diff-e7b9b76560b7a55d03eb1c6f6f7c3295355d6a89f0aa1aa96fb5caa696d360dd*

Trail of Bits did not review the governance and access control contacts.

6. Lack of two-step procedure for critical operations leaves them error-prone - Fixed
The access control is now done through the contracts added in the fix of issue 5.

7. Lack of a contract existence check in TimeLock leads to incorrect assumption of code execution - Not fixed
Hermez team stated:

> *This smart contract is only used for smart contract upgrading.*
> *This may confuse some monitors, so monitors will have to check that the upgrade is done correctly (It's not enough to check just the event).*
>
> *We prefer not to touch this code as it's an external contract that has already been audited.*

9. Lack of zero check on functions - Not fixed
Hermez implemented the protection on some parameters in 31, but not all, including:

- `HermezAuctionProtocol.sol.setBootCoordinator`
- The `initializer` functions
- The `Timelock` functions

Hermez team stated:

> *HermezAuctionProtocol.sol.setBootCoordinator*
> *Boot coordinator is intended to be 0 in a future, also governance could update it if needed.*
>
> *The initializer functions*
> *We check the addresses that we think are required, but some of them might be meant to be 0*
>
> *The Timelock functions*
> *Admin address is claimable, therefore this verification is not required*

12. Initialization functions can be front-run - Partially fixed
Hermez team stated:

> *The deployment script is not finished yet, but we added some checks to ensure that the initialize functions are working as expected, in case of a front-run attack we can easily be aware of that and re-deploy the contracts.*

Checks were added to the deployment script to expect an event. These deployment scripts are still a work in progress.

13. Re-entrancy risks on TokenHez - Not fixed
Hermez team stated:

> *Hermez token was launched before the start of the audit and it's not an ERC-777 so it does not have re-entrancy issues:*
> *https://etherscan.io/token/0xEEF9f339514298C6A857EfCfC1A762aF84438dEE*

14. `ChainId` usage can lead to collisions - Not fixed
Hermez team stated:

> *We use only 2 bytes since Hermez is only expected to be deployed in the Ethereum mainnet or one of its tesnets, so only 2 bytes are needed*

Documentation regarding valid chainID was updated here.

18. `changeDefaultSlotSetBid` allows the closed minimum bid of an open slot to be updated - Fixed

Note: the fix (21) changes the behavior of the last slot at `getCurrentSlotNumber() + _closedAuctionSlots`, which is now considered closed.

19. `cancelTransaction` can be called on non-queued transaction - Not fixed
Hermez team stated:

> *This smart contract is only for Updating the smart contract.*
> *The monitoring system should take in account this logic.*
> *We prefer not to touch this code as it's an external contract that has already been audited*

Trail of Bits did not review the monitoring system.

20. Contracts used as dependencies do not track upstream changes - Not fixed
Hermez team stated:

> *We will freeze the dependencies and check for bugs just before the deployment.*

21. Expected behavior regarding authorization for adding tokens is unclear - Fixed
Hermez team stated:

> *We will clarify the documentation.*

Documentation regarding token listing was updated here.