Quantstamp

API3 Merkle Funder

Executive Summary

This audit report was prepared by Quantstamp, the leader in blockchain security.

Type	Merkle Tree Protected Vault
Timeline	2023-09-18 through 2023-09-22
Language	Solidity
Methods	Architecture Review, Unit Testing, Functional Testing, Computer-Aided Verification, Manual Review
Specification	None
Source Code	• api3dao/merkle-funder #bcbfb33
Auditors	• Danny Aksenov Senior Auditing Engineer • Julio Aguliar Auditing Engineer • Mostafa Yassin Auditing Engineer

Documentation quality	Medium
Test quality	High
Total Findings	7 Acknowledged: 7
High severity findings	1 Acknowledged: 1
Medium severity findings	1 Acknowledged: 1
Low severity findings	1 Acknowledged: 1
Undetermined severity findings	1 Acknowledged: 1
Informational findings	3 Acknowledged: 3

Summary of Findings

The Merkle Funder contracts were designed to provide a way for shared funding to be distributed amongst several accounts without having to store funds separately in several hot wallets. The funder contract deploys a deterministic address for a depository contract, that acts as a type of "vault" for the recipients and their corresponding limits that have been defined in a Merkle tree. To create one of these depositories, an owner (which can be non-zero to avoid withdrawals) and the appropriate Merkle root have to be provided. The funding contract is then able to make the appropriate transfer and withdrawal requests based on the limitations provided by the respective Merkle tree.

Overall, the code is well documented and well-written. Quantstamp was not able to identify a large amount of security issues due to the simplicity of the contract implementation, however, we do have some reservations regarding how the low-level call is being used by the depository contract.

Update: The API3 DAO team has reviewed and acknowledged all the issues raised by the Quantstamp team. The primary concern of the auditors still remains the issue of re-entrancy via the low-level call being made by the `MerkleFunderDepository` contract, however this is an acceptable risk for the API3 DAO team, as the recipients participating in the funding are trusted EOA's and smart contracts. We strongly encourage the API3 DAO to document these risks and caution the end-user of these possibilities.

ID	DESCRIPTION	SEVERITY	STATUS
API-1	Draining of Deposits via Malicious Recipients	● High	Acknowledged
API-2	Malicious Bots Can Grief `Multicall` Operations via Gas Consumption	● Medium	Acknowledged
API-3	Funds Could Get Stuck in the `MerkleFunderDepository`	● Low	Acknowledged
API-4	Risk of Phishing	● Informational	Acknowledged
API-5	Block Production May Not Be Constant Across Different Chains	● Informational	Acknowledged
API-6	`zkSync Era` Incompatibilities	● Informational	Acknowledged

ID	DESCRIPTION	SEVERITY	STATUS
API-7	No Access Control on the `Fund()` Function	● Undetermined ⓘ	Acknowledged

Assessment Breakdown

Quantstamp's objective was to evaluate the repository for security-related issues, code quality, and adherence to specification and best practices.

Disclaimer
Only features that are contained within the repositories at the commit hashes specified on the front page of the report are within the scope of the audit and fix review. All features added in future revisions of the code are excluded from consideration in this report.

Possible issues we looked for included (but are not limited to):

- Transaction-ordering dependence
- Timestamp dependence
- Mishandled exceptions and call stack limits
- Unsafe external calls
- Integer overflow / underflow
- Number rounding errors
- Reentrancy and cross-function vulnerabilities
- Denial of service / logical oversights
- Access control
- Centralization of power
- Business logic contradicting the specification
- Code clones, functionality duplication
- Gas usage
- Arbitrary token minting

Methodology

1. Code review that includes the following
   1. Review of the specifications, sources, and instructions provided to Quantstamp to make sure we understand the size, scope, and functionality of the smart contract.
   2. Manual review of code, which is the process of reading source code line-by-line in an attempt to identify potential vulnerabilities.
   3. Comparison to specification, which is the process of checking whether the code does what the specifications, sources, and instructions provided to Quantstamp describe.
2. Testing and automated analysis that includes the following:
   1. Test coverage analysis, which is the process of determining whether the test cases are actually covering the code and how much code is exercised when we run those test cases.
   2. Symbolic execution, which is analyzing a program to determine what inputs cause each part of a program to execute.
3. Best practices review, which is a review of the smart contracts to improve efficiency, effectiveness, clarity, maintainability, security, and control based on the established industry and academic practices, recommendations, and research.
4. Specific, itemized, and actionable recommendations to help you take steps to secure your smart contracts.

Scope

Files Included

- `contracts/MerkleFunder.sol`
- `contracts/MerkleFunderDepository.sol`

Findings

API-1 Draining of Deposits via Malicious Recipients

● High ⓘ Acknowledged

Update
Marked as "Acknowledged" by the client. The client provided the following explanation:

> This is a design decision and the assumption is that the deployer of a MerkleFunderDepository contract has already verified the recipients (both EOA and smart contracts) and trust each recipient not to drain all the funds (accidentally or on purpose). If for some reason a recipient can not be trusted or it is not possible to pre-determine the amount funds it would consume then this recipient could just be funded by a dedicated MerkleFunderDepository contract

File(s) affected: `MerkleFunder.sol`, `MerkleFunderDepository.sol`

Description: It is currently possible for a malicious `recipient` to drain the depository contract via re-entrancy.

Exploit Scenario:
1. Alice creates a `MerkleFunderDepository` via `MerkleFunder` with Bob's smart contract being one of the "trusted" recipients and with a low threshold of 0.5 ETH and a high threshold of 1 ETH. Alice then funds the depository contract with 3 ETH.
2. Bob decides to be malicious, and taking advantage of the `selfDestruct()` that he has included in his contract, he redeploys a malicious contract, with a `recieve()` fallback function that transfers ETH to his EOA and calls back into `MerkleFunder.fund()` after the transfer.
3. Bob then calls into `MerkleFunder.fund()` to recieve 1 ETH from the depository contract, after which the transfer of funds will call back into `MerkleFunder.fund()` via re-entrancy, passing the requirements checks and receiving an additional 1 ETH. This will continue until the contract is drained of funds.

Recommendation: Due to the multi-call feature that the client plans to utilize, the use of re-entrancy guards may impede that. The recommendation instead would be to hard code the gas limit on the low-level call being made via `MerkleFunderDepository.transfer()` to prevent any complex operations from executing upon transfer, effectively preventing re-entrancy.

API-2
Malicious Bots Can Grief `Multicall` Operations via Gas Consumption

• Medium ⓘ Acknowledged

Update
Marked as "Acknowledged" by the client. The client provided the following explanation:

> Same explanation from API-1 applies

File(s) affected: `MerkleFunder.sol`, `MerkleFunderDepository.sol`

Description: Due to the lack of gas limit restrictions, it is possible for a malicious `recepient` to grief other `recepients` during multi-call transactions that call multiple `fund()` and/or purposefully make the `multi-call` transactions expensive. This can be even more prevalent on other L2's, where the cost of gas is significantly cheaper, making an attack like this more economically viable.

Exploit Scenario:
1. Alice includes Bob's malicious contract as one of the recipients in the `MerkleFunderDepository` contract.
2. Bob's contract purposefully consumes large amounts of gas during the transfer operation.
3. During a multicall transaction with several calls to `fund()`, one of them being Bob's contract, any additional calls to `MerkleFunder.fund()` will fail due to running out of gas.

Recommendation: Introduce a hard-coded gas limit on the low-level call being used in `MerkleFunderDepository.transfer()`.

API-3 Funds Could Get Stuck in the `MerkleFunderDepository`

• Low ⓘ Acknowledged

Update
Marked as "Acknowledged" by the client. The client provided the following explanation:

> The user that called `deployMerkleFunderDepository()` is the one expected to call `fund()`, therefore it already knows the non-hashed values and the order for them when generating merkle tree proofs (https://github.com/api3dao/merkle-funder/blob/main/config/config.example.json#L38). Getting `Invalid Proof` revert messages is preferable over making the `deployMerkleFunderDepository()` more expensive to execute

File(s) affected: `MerkleFunder.sol`

Description: There is no clearly documented way of creating the leaves of the Merkle tree without looking at the `MerkleFunder` contract's function `fund()`, where the leaf is hashed as shown below:

```
bytes32 leaf = keccak256(
    bytes.concat(
        keccak256(abi.encode(recipient, lowThreshold, highThreshold))
    )
);
```

If the `owner` created the leaves in a different order, the `fund()` function will fail with an `InvalidProof()` error. Additionally, the funds that may already be transferred to the deployed `MerkleFunderDepository` would get stuck if the contract has no owner.

Recommendation: Consider adding two additional function arguments to `deployMerkleFunderDepository()` so that it can accept a leaf hash and the parameters to recreate the hash. The function can then create the hash and validate it against the provided leaf hash. Furthermore, we recommend providing user-facing documentation describing the proper leaf hashing process.

API-4 Risk of Phishing

● Informational ⓘ Acknowledged

Update
Marked as "Acknowledged" by the client. The client provided the following explanation:

> Deploying on behalf of other users is an intended feature. The owner is expected to confirm the deployment merkle tree

File(s) affected: `MerkleFunder.sol`

Description: Since any user is able to create a depository without owning the address of the `owner`, there is a risk that uninformed users can be victims of phishing attacks. Where an attacker creates a depository with the victim's address, and while controlling all the Merkle root information and passing off the newly created depository contract as legitimate.

Recommendation: Either consider clearly outlining, in user-facing docs, that the `owner` variable does not mean that the depository belongs to that address, or consider using `msg.sender` instead of `owner` in `deployMerkleFunderDepository()`.

API-5 Block Production May Not Be Constant Across Different Chains

● Informational ⓘ Acknowledged

Update
Marked as "Acknowledged" by the client. The client provided the following explanation:

> We will refrain from using the function that uses `block.number`

File(s) affected: `ExtendedSelfMulticall.sol`

Description: Since the plan is to deploy these contracts across several chains, including L2's, it is important to note that `block.number` is NOT a reliable source of timing information for short terms.

For example: On Arbitrum it reflects the L1 block number, which is updated once per minute.

Recommendation: Consider how this information may affect API3 DAO's business logic.

API-6 `zkSync Era` Incompatibilities

● Informational ⓘ Acknowledged

Update
Marked as "Acknowledged" by the client. The client provided the following explanation:

> We have a ticket in the repository to handle these type of incompatibilities (https://github.com/api3dao/merkle-funder/issues/20)

Description: zkSync Era has many differences from Ethereum on EVM instructions like `CREATE`, `CREATE2`, `CALL`, `STATICCALL`, `DELEGATECALL`, `MSTORE`, `MLOAD`, `CALLDATALOAD`, `CALLDATACOPY`, etc. The full list can be checked here as well as other differences.

Recommendation: If the plan is to deploy to zkSync Era, double-check the compatibility of the contracts, as well as any differences between Ethereum and other L2's.

API-7 No Access Control on the `Fund()` Function

● Undetermined ⓘ Acknowledged

Update
Marked as "Acknowledged" by the client. The client provided the following explanation:

> The intended use-case is to always have enough funds in the MerkleFunderDepository contracts to fund all recipients up to their high threshold at all times so that kind of a preferential funding is not an issue

File(s) affected: `MerkleFunder.sol`

Description: The `fund` function transfers any amount that is bounded by a `lowThreshold` and `highThreshold` and sent to a `recipient`. These Three values are embedded in the Merkle proof. However, an arbitrary user can invoke a transfer to a `recipient` at any time, given there are enough funds in the depository.

While it is a use case for the protocol to allow bots to handle these transfers, it might not always be desired by the `owner` to do this. Especially, when there are multiple possible recipients.

Exploit Scenario:
1. Alice has 5 different recipients
2. Alice's bots want to transfer funds to recipient A
3. A griefer sees that, and front-run Alice's transaction and transfers funds to recipient B

Recommendation: Determine whether this behavior coincides with the business logic of the team, if it does not, here are two possible recommendations to consider:
- Adding restrictions on whom can call the `fund` function.
- Adding a pausing functionality to allow an owner to pause transfers.

Definitions

- **High severity** – High-severity issues usually put a large number of users' sensitive information at risk, or are reasonably likely to lead to catastrophic impact for client's reputation or serious financial implications for client and users.
- **Medium severity** – Medium-severity issues tend to put a subset of users' sensitive information at risk, would be detrimental for the client's reputation if exploited, or are reasonably likely to lead to moderate financial impact.
- **Low severity** – The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated is low impact in view of the client's business circumstances.
- **Informational** – The issue does not post an immediate risk, but is relevant to security best practices or Defence in Depth.
- **Undetermined** – The impact of the issue is uncertain.
- **Fixed** – Adjusted program implementation, requirements or constraints to eliminate the risk.
- **Mitigated** – Implemented actions to minimize the impact or likelihood of the risk.
- **Acknowledged** – The issue remains in the code but is a result of an intentional business or design decision. As such, it is supposed to be addressed outside the programmatic means, such as: 1) comments, documentation, README, FAQ; 2) business processes; 3) analyses showing that the issue shall have no negative consequences in practice (e.g., gas analysis, deployment settings).

Appendix

File Signatures

The following are the SHA-256 hashes of the reviewed files. A file with a different SHA-256 hash has been modified, intentionally or otherwise, after the security review. You are cautioned that a different SHA-256 hash could be (but is not necessarily) an indication of a changed condition or potential vulnerability that was not within the scope of the review.

Contracts

- `31b...52c ./contracts/MerkleFunderDepository.sol`
- `c54...c92 ./contracts/MerkleFunder.sol`
- `7da...39a ./contracts/interfaces/IMerkleFunderDepository.sol`
- `782...bb3 ./contracts/interfaces/IMerkleFunder.sol`

Tests

- `853...09a ./test/test-utils.ts`
- `9bf...e20 ./test/MerkleFunder.sol.ts`
- `db2...22f ./test/MerkleFunderDepository.sol.ts`

Toolset

The notes below outline the setup and steps performed in the process of this audit.

Setup

Tool Setup:

- Slither v0.9.3

Steps taken to run the tools:

1. Install the Slither tool: `pip3 install slither-analyzer`
2. Run Slither from the project directory: `slither .`

Automated Analysis

Slither

Slither found 29 results all of which were either deemed as false positives or covered by the report.

Test Suite Results

At the time of this report, all tests are passing.

```
MerkleFunderDepository
  constructor
    Owner is not zero address
      Root is not zero
        ✔ constructs (1423ms)
      Root is zero
        ✔ constructs
    Owner is zero address
      ✔ constructs
  receive
    ✔ receives
  transfer
    Sender is MerkleFunder
      Transfer is successful
        ✔ transfers
      Transfer is not successful
        ✔ reverts
    Sender is not MerkleFunder
      ✔ reverts


MerkleFunder
  deployMerkleFunderDepository
    Root is not zero
      MerkleFunderDepository has not been deployed before
        ✔ deploys MerkleFunderDepository (1238ms)
      MerkleFunderDepository has been deployed before
        ✔ reverts
    Root is zero
      ✔ reverts
  fund
    Recipient address is not zero
      Low threshold is not higher than high
        High threshold is not zero
          Proof is valid
            Balance is low enough
              Amount is not zero
                Respective MerkleFunderDepository is deployed
                  Transfer is successful
                    ✔ funds (89ms)
                  Transfer is not successful
                    ✔ reverts
                Respective MerkleFunderDepository is not deployed
                  ✔ reverts
              Amount is zero
                ✔ reverts
            Balance is not low enough
              ✔ reverts (40ms)
          Proof is not valid
            ✔ reverts
        High threshold is zero
          ✔ reverts
      Low threshold is higher than high
        ✔ reverts
    Recipient address is zero
      ✔ reverts
  withdraw
    Recipient address is not zero
      Amount is not zero
        MerkleFunderDepository is deployed
          Balance is sufficient
            Transfer is successful
              ✔ withdraws (42ms)
            Transfer is not successful
              ✔ reverts
          Balance is insufficient
            ✔ reverts
        MerkleFunderDepository is not deployed
          ✔ reverts
      Amount is zero
        ✔ reverts
    Recipient address is zero
      ✔ reverts
  withdrawAll
    ✔ withdraws all
  computeMerkleFunderDepositoryAddress
    Root is not zero
      ✔ computes
    Root is zero
      ✔ reverts


28 passing (3s)
```

Code Coverage

Tests are yielding 100% coverage.

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Lines
contracts/	100	100	100	100	
MerkleFunder.sol	100	100	100	100	
MerkleFunderDepository.sol	100	100	100	100	
contracts/interfaces/	100	100	100	100	
IMerkleFunder.sol	100	100	100	100	
IMerkleFunderDepository.sol	100	100	100	100	
All files	100	100	100	100	

Changelog

- 2023-09-25 - Initial report
- 2023-10-02 - Final report

About Quantstamp

Quantstamp is a global leader in blockchain security. Founded in 2017, Quantstamp's mission is to securely onboard the next billion users to Web3 through its best-in-class Web3 security products and services.

Quantstamp's team consists of cybersecurity experts hailing from globally recognized organizations including Microsoft, AWS, BMW, Meta, and the Ethereum Foundation. Quantstamp engineers hold PhDs or advanced computer science degrees, with decades of combined experience in formal verification, static analysis, blockchain audits, penetration testing, and original leading-edge research.

To date, Quantstamp has performed more than 500 audits and secured over $200 billion in digital asset risk from hackers. Quantstamp has worked with a diverse range of customers, including startups, category leaders and financial institutions. Brands that Quantstamp has worked with include Ethereum 2.0, Binance, Visa, PayPal, Polygon, Avalanche, Curve, Solana, Compound, Lido, MakerDAO, Arbitrum, OpenSea and the World Economic Forum.

Quantstamp's collaborations and partnerships showcase our commitment to world-class research, development and security. We're honored to work with some of the top names in the industry and proud to secure the future of web3.

Notable Collaborations & Customers:

- Blockchains: Ethereum 2.0, Near, Flow, Avalanche, Solana, Cardano, Binance Smart Chain, Hedera Hashgraph, Tezos
- DeFi: Curve, Compound, Maker, Lido, Polygon, Arbitrum, SushiSwap
- NFT: OpenSea, Parallel, Dapper Labs, Decentraland, Sandbox, Axie Infinity, Illuvium, NBA Top Shot, Zora
- Academic institutions: National University of Singapore, MIT