Zellic

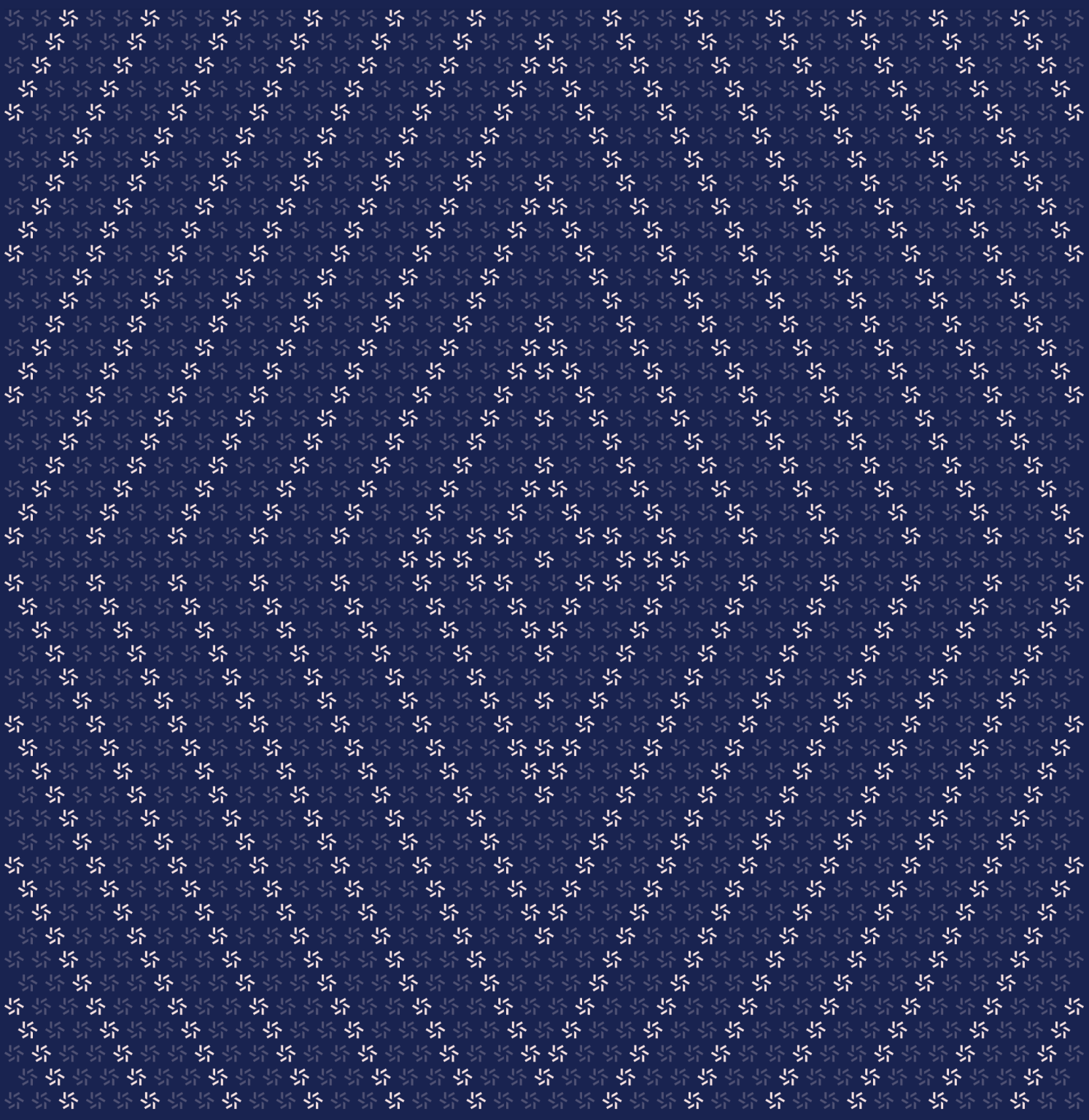
Prepared for
Jimmy Li
MegaETH

Prepared by
Jisub Kim
Jaeeu Kim
Zellic

November 20, 2025

MegaETH Predeposit

Smart Contract Security Assessment



Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for MegaETH from November 14th to November 17th, 2025. During this engagement, Zellic reviewed MegaETH Predeposit's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Is there any way to bypass or race the `entityID` <-> wallet one-to-one binding?
- Are time-window, pause, and cap states enforceable under boundary/race conditions?
- Can permits be replayed, cross-sale be reused, or signature/payload constraints be bypassed?
- Could admin operations accidentally or maliciously risk funds (e.g., withdrawals or misconfigurations)?
- Is `withdrawToTreasury` safe from misuse (e.g., token selection, balance checks, precision)?
- Are there any vulnerabilities that could result in the loss of user funds?
- Are access controls implemented effectively to prevent unauthorized operations?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Permit signer
- Off-chain components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

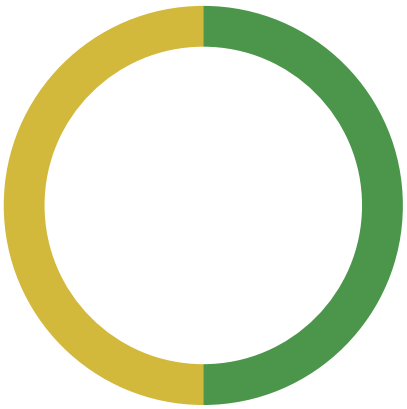
1.4. Results

During our assessment on the scoped MegaETH Predeposit contracts, we discovered two findings. No critical issues were found. One finding was of medium impact and one was of low impact.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of MegaETH in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	1
Low	1
Informational	0



2. Introduction

2.1. About MegaETH Predeposit

MegaETH contributed the following description of MegaETH Predeposit:

> Protocol `MegaUSDCDepositVault` is a secure pre-deposit vault for USDC on Ethereum mainnet as part of the MegaETH onboarding. It verifies eligibility via off-chain–issued purchase permits, enforces time-windowed deposits and a global cap, binds each KYC'ed `entityID` to a single wallet, and lets the owner manage pause/cap/timing and withdraw funds to treasury for future bridging and USDm distribution.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no

hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

MegaETH Predeposit Contracts

Type	Solidity
Platform	EVM-compatible
Target	pre-deposit-vault
Repository	https://github.com/megaeth-labs/pre-deposit-vault ↗
Version	`346955cb7236255f876de7b0faa4954229b9649e`

Programs	`MegaUSDmDistributorVault.sol`
	`MegaUSDCDepositVault.sol`
	`permits/PurchasePermit.sol`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of two person-days. The assessment was conducted by two consultants over the course of two calendar days.

Contact Information

The following project manager was associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Jisub Kim
Engineer
jisub@zellic.io ↗

Jaeeu Kim
Engineer
jaeeu@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

November 14, 2025	Start of primary review period
November 17, 2025	End of primary review period

3. Detailed Findings

3.1. Centralization risks

Target	MegaUSDCDepositVault		
Category	Protocol Risks	**Severity**	High
Likelihood	Low	**Impact**	Medium

Description

All critical configuration and fund-custody operations are guarded solely by `onlyOwner`. The owner can pause deposits, modify caps, rotate signers, and transfer any ERC-20 balance to `treasury` via `withdrawToTreasury`. There is no user-triggered withdrawal path, and no multi-signature or timelock enforcement, so custody fully depends on the owner's honesty and key custody.

```
function withdrawToTreasury(address token, uint256 amount)
    external onlyOwner {
    IERC20(token).safeTransfer(treasury, amount);
}
```

Impact

If the owner key is compromised or behaves maliciously, all deposited USDC can be redirected without user consent. This trust assumption may be acceptable for a custodial phase but should be clearly communicated to participants as it introduces a single point of failure.

Recommendations

Consider documenting the custodial model and using a multi-sig key custody.

Remediation

This issue has been acknowledged by MegaETH.

3.2. Zero-value `entityID` allows wallet rebinding

Target	MegaUSDCDepositVault		
Category	Business Logic	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The `walletToEntityID` treats `bytes16(0)` as unbound, yet `entityID = 0` is a valid permit. A wallet that first deposits with entity 0 gets stored as zero, but later checks still see it as unbound and allow another permit with a nonzero entity. The same wallet can therefore bind to multiple entity IDs, breaking the promised 1:1 mapping.

```
function deposit(uint256 amount, PurchasePermit calldata permit,
    bytes calldata permitSignature) external {
    bytes16 entityID = permit.entityID;
    ...
    bytes16 existingEntityID = walletToEntityID[msg.sender];
    if (existingEntityID != bytes16(0) && existingEntityID != entityID) {
        revert WalletAlreadyBound(msg.sender, existingEntityID);
    }
    ...
    walletToEntityID[msg.sender] = entityID;
}
```

Impact

An attacker can reuse a single wallet to deposit under multiple entity IDs, enabling allocation circumvention and bypassing the intended one-wallet-per-entity constraint.

Recommendations

Consider adding a check to block zero `entityID`.

Remediation

This issue has been acknowledged by MegaETH, and a fix was implemented in commit `776da3c9` ↗.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. There is no individual cap

The `deposit` function only checks the global `cap` against `totalDeposited`. There is no logic that limits how much a single entity can send, so a single participant could deposit up to the maximum cap, preventing others from taking part.

The MegaETH team confirmed this is on purpose; they only control per-entity limits when issuing permits off chain. As a result, one user may fill the whole remaining cap if their permit allows it.

4.2. The `totalDeposited` reflects all previous deposits

The `setPermitSigner` function swaps the key that checks new permits.

```
function setPermitSigner(address newPermitSigner) external onlyOwner {
    if (newPermitSigner == address(0)) {
        revert UnauthorizedSigner(newPermitSigner);
    }

    emit PermitSignerUpdated(permitSigner, newPermitSigner);
    permitSigner = newPermitSigner;
}
```

In this function, changing the signer only affects newly submitted `PurchasePermits`; existing deposits remain valid and continue contributing to `totalDeposited` and the cap. So if prior deposits are supposed to become invalid, the accounting could become inconsistent.

The MegaETH team responded that the permit signer validates eligibility at deposit time, so completed deposits remain valid historical records that continue contributing to `totalDeposited` and cap accounting.

The same perspective applies to the `setSaleUUID` function as well.

5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1. Module: MegaUSDCDepositVault.sol

Function: `deposit(uint256 amount, PurchasePermit permit, bytes permitSignature)`

This function is used to deposit USDC into the vault. This action needs a valid `PurchasePermit`, which is signed by `permitSigner`.

Inputs

- `amount`
  - **Control**: Arbitrary `uint256`.
  - **Constraints**: Must be greater than zero.
  - **Impact**: Amount of USDC to deposit.
- `permit`
  - **Control**: Arbitrary `PurchasePermit`.
  - **Constraints**: Should be signed by `permitSigner` and validated by `_validatePurchasePermit`.
  - **Impact**: The `PurchasePermit` to validate the eligibility of the user.
- `permitSignature`
  - **Control**: Arbitrary bytes.
  - **Constraints**: Should be a valid signature of the permit.
  - **Impact**: Signature of the permit.

Branches and code coverage

Intended branches

- Invoke `_validatePurchasePermit` to validate the provided permit.
  - ☑ Test coverage
- Calculate the actual amount to deposit based on the remaining cap.
  - ☑ Test coverage

- Update user and entity information if it is a new `entityID`.
  - ☑ Test coverage
- Update total deposited amount and the user's deposit amount.
  - ☑ Test coverage
- Transfer USDC from the user to the vault.
  - ☑ Test coverage
- Emit the `Deposited` event.
  - ☑ Test coverage

Negative behavior

- Revert if the amount is zero.
  - ☑ Negative test
- Revert if the total deposits have reached the cap.
  - ☑ Negative test
- Revert if the permit is not signed by `permitSigner`.
  - ☑ Negative test
- Revert if the permit is expired.
  - ☑ Negative test
- Revert if the permit is not valid (`forcedLockup`).
  - ☑ Negative test
- Revert if the permit is not valid (`isEligible`).
  - ☐ Negative test
- Revert if the sender is not the wallet of the permit.
  - ☑ Negative test
- Revert if the `entityID` is already bound to another wallet.
  - ☑ Negative test
- Revert if the wallet is already bound to another `entityID`.
  - ☑ Negative test
- Revert if the actual amount is zero.
  - ☑ Negative test

Function: `setCap(uint256 newCap)`

This function is used to set the new deposit cap. It is only callable by the owner.

Inputs

- `newCap`
  - **Control**: Arbitrary `uint256`.
  - **Constraints**: Must be greater than zero and greater than the current total deposited amount.
  - **Impact**: New deposit cap.

Branches and code coverage

Intended branches

- Emit the `CapUpdated` event.
  - ☑ Test coverage
- Update the cap.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the owner.
  - ☑ Negative test
- Revert if the new cap is zero.
  - ☑ Negative test
- Revert if the new cap is less than the current total deposited amount.
  - ☑ Negative test

Function: `setEndTime(uint48 newEndTime)`

This function is used to set the end time for deposits. It is only callable by the owner.

Inputs

- `newEndTime`
  - **Control**: Arbitrary `uint48`.
  - **Constraints**: Must be greater than the current block timestamp.
  - **Impact**: New end time for deposits.

Branches and code coverage

Intended branches

- Emit the `EndTimeUpdated` event.
  - ☑ Test coverage
- Update `endTime`.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the owner.
  - ☑ Negative test
- Revert if the new end time is less than the current block timestamp.
  - ☑ Negative test
- Revert if the new end time is less than the start time.
  - ☑ Negative test

Function: `setPause(bool pause)`

This function is used to set the pause status for deposits. It is only callable by the owner.

Inputs

- `pause`
  - **Control**: True to pause — false to unpause.
  - **Constraints**: None.
  - **Impact**: Pause status for deposits.

Branches and code coverage

Intended branches

- Pause/unpause the contract.
  - ☑ Test coverage
- Emit the `PauseStatusUpdated` event.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the owner.
  - ☑ Negative test

Function: `setPermitSigner(address newPermitSigner)`

This function is used to set the new permit signer. It is only callable by the owner.

Inputs

- `newPermitSigner`
  - **Control**: Arbitrary address.
  - **Constraints**: Must not be zero.
  - **Impact**: New permit signer.

Branches and code coverage

Intended branches

- Emit the `PermitSignerUpdated` event.
  - ☑ Test coverage
- Update `permitSigner`.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the owner.
  - ☐ Negative test
- Revert if the new permit signer is zero.
  - ☐ Negative test

Function: `setSaleUUID(byte[16] newSaleUUID)`

This function is used to set the new sale UUID. It is only callable by the owner.

Inputs

- `newSaleUUID`
  - **Control**: Arbitrary `bytes16`.

- **Constraints**: Must not be zero.
- **Impact**: New sale UUID.

Branches and code coverage

Intended branches

- Emit the `SaleUUIDUpdated` event.
  - ☑ Test coverage
- Update `saleUUID`.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the owner.
  - ☐ Negative test
- Revert if the new sale UUID is zero.
  - ☐ Negative test

Function: `setStartTime(uint48 newStartTime, uint48 activeDuration)`

This function is used to set the start time and active duration for deposits. It is only callable by the owner.

Inputs

- `newStartTime`
  - **Control**: Arbitrary `uint48`.
  - **Constraints**: Must be greater than the current block timestamp.
  - **Impact**: New start time for deposits.
- `activeDuration`
  - **Control**: Arbitrary `uint48`.
  - **Constraints**: Must be greater than zero.
  - **Impact**: New active duration for deposits.

Branches and code coverage

Intended branches

- Emit the `StartTimeUpdated` and `EndTimeUpdated` events.

☑ Test coverage

- Update `startTime` and `endTime`.

☑ Test coverage

Negative behavior

- Revert if the caller is not the owner.

☑ Negative test

- Revert if the new start time is less than the current block timestamp.

☑ Negative test

- Revert if the active duration is zero.

☑ Negative test

Function: `setTreasury(address newTreasury)`

This function is used to set the new treasury address. It is only callable by the owner.

Inputs

- `newTreasury`
  - **Control**: Arbitrary address.
  - **Constraints**: Must not be zero.
  - **Impact**: New treasury address.

Branches and code coverage

Intended branches

- Emit the `TreasuryUpdated` event.

☑ Test coverage

- Update treasury.

☑ Test coverage

Negative behavior

- Revert if the caller is not the owner.

☑ Negative test

- Revert if the new treasury is zero.

☐ Negative test

Function: `withdrawToTreasury(address token, uint256 amount)`

This function is used to withdraw tokens from the contract to the treasury address. It is only callable by the owner.

Inputs

- `token`
  - **Control**: Arbitrary address.
  - **Constraints**: Must not be zero.
  - **Impact**: Token address to withdraw.
- `amount`
  - **Control**: Arbitrary `uint256`.
  - **Constraints**: Must be greater than zero and less than or equal to the balance of the token.
  - **Impact**: Amount to withdraw.

Branches and code coverage

Intended branches

- Emit the `FundsWithdrawn` event.
  - ☑ Test coverage
- Transfer tokens from the contract to the treasury address.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the owner.
  - ☑ Negative test
- Revert if the token address is zero.
  - ☐ Negative test
- Revert if the treasury address is zero.
  - ☐ Negative test
- Revert if the amount is zero.
  - ☑ Negative test
- Revert if the amount is greater than the balance of the token.
  - ☑ Negative test

5.2. Module: MegaUSDmDistributorVault.sol

Function: `claim(address user, uint256 tokenAmount, byte[32][] merkleProof)`

This function is used to claim tokens and ETH for a user. It is only callable by the user themselves if `selfClaim` is true; otherwise, it is only callable by the owner.

Inputs

- `user`
  - **Control**: Arbitrary address.
  - **Constraints**: Must be `msg.sender` if `selfClaim` is true and must be in the Merkle tree.
  - **Impact**: Address of the user to claim for.
- `tokenAmount`
  - **Control**: Arbitrary `uint256`.
  - **Constraints**: Must be in the Merkle tree.
  - **Impact**: Amount of tokens to claim.
- `merkleProof`
  - **Control**: Arbitrary `bytes32[]`.
  - **Constraints**: Must be a valid Merkle proof for the user and token amount.
  - **Impact**: Merkle proof for verification.

Branches and code coverage

Intended branches

- Invoke `MerkleProof.verify` to verify the Merkle proof.
  - ☑ Test coverage
- Update the claim record.
  - ☑ Test coverage
- Transfer ERC-20 tokens.
  - ☑ Test coverage
- Transfer ETH.
  - ☑ Test coverage
- Emit the `Claimed` event.
  - ☑ Test coverage

Negative behavior

- Revert if claiming is paused.
  - ☑ Negative test
- Revert if the caller is not the user themselves, when `selfClaim` is true.
  - ☐ Negative test
- Revert if the user has already claimed.
  - ☑ Negative test
- Revert if the Merkle proof is invalid.
  - ☑ Negative test
- Revert if the token amount is zero and the ETH amount per user is zero.
  - ☐ Negative test
- Revert if the token balance is insufficient.
  - ☐ Negative test
- Revert if the ETH balance is insufficient.
  - ☐ Negative test

Function: `setEthPerUser(uint256 _ethPerUser)`

This function is used to set Ether (that is claimable by the user) per user. It is only callable by the owner.

Inputs

- `_ethPerUser`
  - **Control**: Arbitrary `uint256`.
  - **Constraints**: None.
  - **Impact**: New ETH amount per user.

Branches and code coverage

Intended branches

- Update `ethPerUser`.
  - ☑ Test coverage
- Emit the `EthPerUserUpdated` event.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the owner.
  - ☑ Negative test

Function: `setMerkleRoot(byte[32] _merkleRoot)`

This function is used to set the new Merkle root. It is only callable by the owner.

Inputs

- `_merkleRoot`
  - **Control**: Arbitrary `bytes32`.
  - **Constraints**: None.
  - **Impact**: New Merkle root.

Branches and code coverage

Intended branches

- Update `merkleRoot`.
  - ☑ Test coverage
- Emit the `MerkleRootUpdated` event.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the owner.
  - ☑ Negative test

Function: `setPause(bool _pause)`

This function is used to set the pause status for claiming. It is only callable by the owner.

Inputs

- `_pause`
  - **Control**: True to pause — false to unpause.
  - **Constraints**: None.

- **Impact**: Pause status for claiming.

Branches and code coverage

Intended branches

- Update pause.
  - ☑ Test coverage
- Emit the `PauseUpdated` event.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the owner.
  - ☑ Negative test

Function: `setSelfClaim(bool _selfClaim)`

This function is used to set the self-claim property for claiming. It is only callable by the owner.

Inputs

- `_selfClaim`
  - **Control**: Arbitrary bool.
  - **Constraints**: None.
  - **Impact**: Self-claim property for claiming.

Branches and code coverage

Intended branches

- Update `selfClaim`.
  - ☑ Test coverage
- Emit the `SelfClaimUpdated` event.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the owner.
  - ☑ Negative test

Function: `setToken(address _token)`

This function is used to set the new token address. It is only callable by the owner.

Inputs

- `_token`
  - **Control**: Arbitrary address.
  - **Constraints**: Must not be zero.
  - **Impact**: New token address.

Branches and code coverage

Intended branches

- Update the token.
  - ☑ Test coverage
- Emit the `TokenUpdated` event.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the owner.
  - ☑ Negative test
- Revert if the new token address is zero.
  - ☐ Negative test

Function: `withdrawAll(address to)`

This function is used to withdraw all remaining tokens and ETH from the contract. It is only callable by the owner.

Inputs

- `to`
  - **Control**: Arbitrary address.
  - **Constraints**: None.
  - **Impact**: Address to receive the funds.

Branches and code coverage

Intended branches

- Withdraw all remaining base tokens.
  - ☑ Test coverage
- Withdraw all remaining ETH.
  - ☑ Test coverage
- Emit the `TokensWithdrawn` event.
  - ☑ Test coverage
- Emit the `ETHWithdrawn` event.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the owner.
  - ☑ Negative test

Function: `withdrawETH(uint256 amount, address to)`

This function is used to withdraw ETH from the contract. It is only callable by the owner.

Inputs

- `amount`
  - **Control**: Arbitrary `uint256`.
  - **Constraints**: Must be less than or equal to the balance of the contract.
  - **Impact**: Amount to withdraw.
- `to`
  - **Control**: Arbitrary address.
  - **Constraints**: Must not be zero.
  - **Impact**: Address to receive the funds.

Branches and code coverage

Intended branches

- Calculate the withdrawal amount.
  - ☑ Test coverage

- Determine the recipient address.
  - ☑ Test coverage
- Revert if the withdrawal amount is zero.
  - ☑ Test coverage
- Transfer the ETH to the recipient address.
  - ☑ Test coverage
- Emit the `ETHWithdrawn` event.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the owner.
  - ☑ Negative test
- Revert if the amount is zero.
  - ☑ Negative test

Function: `withdrawToken(address _token, uint256 amount, address to)`

This function is used to withdraw ERC-20 tokens from the contract. It is only callable by the owner.

Inputs

- `_token`
  - **Control**: Arbitrary address.
  - **Constraints**: None.
  - **Impact**: Address of the token to withdraw.
- `amount`
  - **Control**: Arbitrary `uint256`.
  - **Constraints**: Must be less than or equal to the balance of the token.
  - **Impact**: Amount to withdraw.
- `to`
  - **Control**: Arbitrary address.
  - **Constraints**: None.
  - **Impact**: Address to receive the funds.

Branches and code coverage

Intended branches

- Calculate the withdrawal amount.
  - ☑ Test coverage
- Determine the recipient address.
  - ☑ Test coverage
- Transfer the tokens to the recipient address.
  - ☑ Test coverage
- Emit the `TokensWithdrawn` event.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the owner.
  - ☑ Negative test
- Revert if the amount is zero.
  - ☑ Negative test

6. Assessment Results

During our assessment on the scoped MegaETH Predeposit contracts, we discovered two findings. No critical issues were found. One finding was of medium impact and one was of low impact.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.