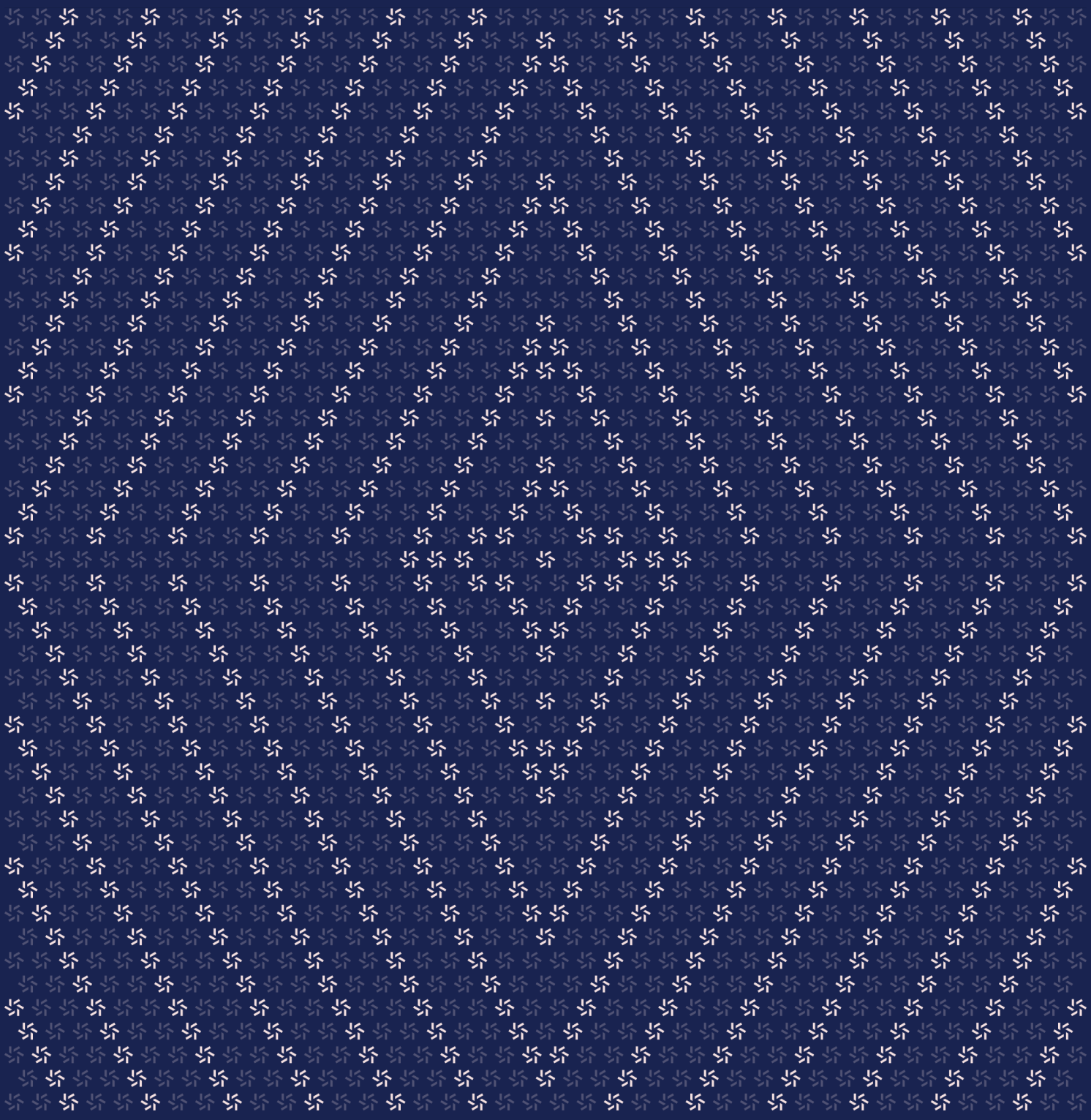
Zellic

Prepared for
Shushant Dahal
Blake Arnold
Definitive Finance, Inc

Prepared by
Jaeeu Kim
Ayaz Mammadov
Zellic

March 28, 2024

Definitive
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 Definitive Finance, Inc from March 28th to March 29th, 2024. During this engagement, Zellic reviewed Definitive'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:

- Are any of the initiable contracts improperly access controlled / lacking the proper access control?
- Assuming the definitive API signing key is not compromised, is a malicious actor able to hijack vault deploys for clients on chains they did not deploy on?
- By creating a minimal proxy instead of a new deploy for every client, is there anything that was expected to be set that will instead be the default value? For example, if we had a boolean value `TRADING_GUARDIAN_TRADING_ENABLED = true` in `TradingVaultImplementation`, creating a minimal proxy would set this value to false on the minimal proxy.
- If a Definitive performer is compromised, can funds be siphoned to wallets not controlled by Definitive or a client?
- If a Definitive performer is compromised and Definitive uses the GlobalGuardian to disable trading, can funds be siphoned to wallets not controlled by Definitive or a client?

1.3. Non-goals and Limitations

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

- Front-end components
- Infrastructure relating to the project
- Key custody
- Incorrectly supplied deployment parameters

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

During this assessment, uncertainties surrounding the API's functionality limited our ability to

completely ensure the security of aspects beyond the project's defined scope. Some examples include, whether malicious vault implementations are correctly filtered and, in the case of the EIP-1271 contract, doubts about whether the signing process will be properly executed.

1.4. Results

During our assessment on the scoped Definitive contracts, we discovered one finding, which was of low impact.

Additionally, Zellic recorded its notes and observations from the assessment for Definitive Finance, Inc's benefit in the Discussion section (4. ↗).

Breakdown of Finding Impacts

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

2. Introduction

2.1. About Definitive

Definitive Finance, Inc contributed the following description of Definitive:

> Definitive is a DeFi gateway for institutional clients, providing smart vaults for yield management, trading, and leverage.

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:

Definitive Contracts

Repository	https://github.com/DefinitiveCo/contracts ↗
Version	contracts: `0f833615ec8c37b7c9dbf2d37f05e5a50930fb06`
Programs	• contracts/strategies/Trading/v2/TradingVaultFactory.sol • contracts/strategies/Trading/v2/TradingVaultImplementation.sol • contracts/base/BaseSimpleSwapInitiable.sol • contracts/base/BaseRecoverSignerInitiable.sol • contracts/tools/GlobalGuardian/GlobalGuardian.sol
Type	Solidity
Platform	EVM-compatible

2.4. Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of 3.5 person-days. The assessment was conducted over the course of two calendar days.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald
Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Jaeeu Kim
Engineer
jaeeu@zellic.io

Ayaz Mammadov
Engineer
ayaz@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

March 28, 2024	Kick-off call
March 28, 2024	Start of primary review period
March 29, 2024	End of primary review period

3. Detailed Findings

3.1. Multiple Native deposits result in a revert

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

Description

The function `_depositNativeAndERC20` handles deposits in both ERC-20 tokens and Native tokens. These tokens are supplied as an array of tokens with an equivalent corresponding array of amounts; the Native token is supplied as a specific address. There is then a check if a token is the `NATIVE_ASSET_ADDRESS`, and if that is so, then it is marked as the `nativeAssetIndex`. Later, the corresponding amount in the `nativeAssetIndex` is compared against the `msg.value` sent. The issue arises when multiple Native token amounts are sent — only the latest index's amount is used when comparing against the `msg.value` instead of summing the total amount of value of the Native tokens.

```
  function _depositNativeAndERC20(uint256[] calldata amounts, address[]
    calldata assetAddresses) internal virtual {
      uint256 assetAddressesLength = assetAddresses.length;
...
      for (uint256 i; i < assetAddressesLength; ) {
          if (assetAddresses[i] == DefinitiveConstants.NATIVE_ASSET_ADDRESS) {
              nativeAssetIndex = i;
              hasNativeAsset = true;
              unchecked {
                  ++i;
              }
              continue;
          }
      }
...
      // Revert if depositing native asset and amount != msg.value
      if (hasNativeAsset && msg.value != amounts[nativeAssetIndex]) {
          revert InvalidMsgValue();
      }
  }
```

Impact

Valid deposits with multiple Native token deposits will always be rejected.

Recommendations

Sum the total Native token amount sent, then compare against the `msg.value` provided.

Remediation

This issue has been acknowledged by Definitive Finance, Inc, and fixes were implemented in the following commits:

- 28aeec8c ↗
- 47297e77 ↗

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. Access-control differences in `deposit` and fallback

The function `deposit` has the modifier `onlyClients`, allowing only clients to deposit ERC-20 tokens and Native token into the trading vaults; however, the Native fallback function also allows anyone to send Native token to the trading vault. The intent behind this access control is to ensure that the events emitted by the `deposit` function are those by the client and that unwanted coins are not sent to the trading vault, cluttering the UI and confusing the management of funds.

```
function _depositNativeAndERC20(uint256[] calldata amounts, address[]
    calldata assetAddresses) internal virtual {
...
}
```

```
receive() external payable virtual {
    emit NativeTransfer(_msgSender(), msg.value);
}
```

4.2. Front-running is possible on certain deployments

When creating a vault through the TradingVaultFactory, specifically with the `createVaultWithPermission` call, the deployment parameters are validated against certain fields, then a signature check is performed, one of which being the `chain.id`. However, if a `chain.id` of `0xffffffff` is provided, the check is skipped.

```
function _verifyDeploymentParams(DeployParams calldata deployParams,
    bytes calldata signature) private view {
    ...
    if (deployParams.chainId != block.chainid && deployParams.chainId !=
    type(uint256).max) {
        revert MismatchedChainId();
    }

    SignatureVerifier.verifySignature(isSignatureVerifier,
    encodeDeploymentParams(deployParams), signature);
    }
```

As a result, a deployment could be front-run if an attacker were to retrieve the payload and the signature provided on a blockchain where the deployment has already happened. However, there are no security implications as all the initialization parameters are from the signed payload, which does not depend on who deploys the vault. It is still important to factor in such a possibility when deploying vaults.

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: BaseRecoverSignerInitiable.sol

Function: `isValidSignature(byte[32] _hash, bytes _encodedSignature)`

This function is used to verify that the signer of `_hash` is `clientAdminAddress`. It is expected to return `0x1626ba7e` when the signature is valid.

Inputs

- `_hash`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Value of the hash.
- `_encodedSignature`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Encoded signature, containing the `clientAdminAddress` and signature.

Branches and code coverage

Intended branches

- Decodes the `_encodedSignature` to get the `clientAdminAddress` and `signature`.
  - ☑ Test coverage
- Checks if the `clientAdminAddress` has the `DEFAULT_ADMIN_ROLE`.
  - ☑ Test coverage
- Invokes `isValidSignature` on the `clientAdminAddress` if it is a contract.
  - ☑ Test coverage
- Returns `0x1626ba7e` if the `clientAdminAddress` is not a contract and the signature is valid.
  - ☑ Test coverage

Negative behavior

- Reverts if the `clientAdminAddress` does not have the `DEFAULT_ADMIN_ROLE`.

☑ Negative test

- Reverts if `clientAdminAddress` is not a contract and the signature is invalid.
  ☑ Negative test

5.2. Module: BaseSimpleSwapInitiable.sol

Function: `disableSwapHandlers(address[] swapHandlers)`

This function is used to disable swap handlers. This function is only callable by the handler manager.

Inputs

- `swapHandlers`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Array of swap handlers address to disable.

Branches and code coverage

Intended branches

- Invokes `_updateSwapHandlers` with `swapHandlers` and `false`, updating mapping of `_swapHandlers` to false.
  ☑ Test coverage

Negative behavior

- Reverts if the caller does not have the admin role.
  ☑ Negative test

Function: `enableSwapHandlers(address[] swapHandlers)`

This function is used to enable swap handlers. This function is only callable by the handler manager.

Inputs

- `swapHandlers`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Array of swap handlers address to enable.

Branches and code coverage

Intended branches

- Invokes `_updateSwapHandlers` with `swapHandlers` and `true`, updating mapping of `_swapHandlers` to true.
  - ☑ Test coverage

Negative behavior

- Reverts if the caller is not the handler manager.
  - ☑ Negative test
- Reverts if the `STOP_GUARDIAN_ENABLED` is true.
  - ☑ Negative test

Function: `swap(SwapPayload[] payloads, address outputToken, uint256 amountOutMin, uint256 feePct)`

This function is used to swap tokens. This function is only callable by the definitive role. This function is guarded by the non-reentrant `stopGuard` and `tradingEnabled` modifiers. This function invokes `_swap` with `payloads` and `outputToken`, and it invokes `_processSwap` for each payload and handler's swap function.

Inputs

- `payloads`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Array of swap payloads, containing the swap token, amount, amount out min, handler calldata, and signature.
- `outputToken`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the output token.
- `amountOutMin`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Amount out min.
- `feePct`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Value of the fee percentage.

Branches and code coverage

Intended branches

- Checks if the fee percentage is greater than the maximum fee percentage.
  - ☑ Test coverage
- Invokes `_swap` with `payloads` and `outputToken`.
  - ☑ Test coverage
- Checks if the output amount is less than the minimum output amount.
  - ☑ Test coverage
- Creates and updates an array of swap tokens to use in emitting the `SwapHandled` event.
  - ☑ Test coverage
- Invokes `_handleFeesOnAmount` with `outputToken`, `outputAmount`, and `feePct` if the fee account is not zero and the output amount and fee percent are greater than zero.
  - ☑ Test coverage
- Returns the output amount.
  - ☑ Test coverage

Negative behavior

- Reverts if the caller is not the definitive role.
  - ☑ Negative test
- Reverts if this function is reentrant.
  - ☑ Negative test
- Reverts if the `STOP_GUARDIAN_ENABLED` is true.
  - ☑ Negative test
- Reverts if the trading is not enabled.
  - ☑ Negative test
- Reverts if the fee percentage is greater than the maximum fee percentage.
  - ☑ Negative test
- Reverts if the output amount is less than the minimum output amount.
  - ☑ Negative test

Function call analysis

- `_swap(SwapPayload[] memory payloads, address expectedOutputToken)`
  - Swaps tokens using the swap payloads.
  - Calls `_processSwap` for each payload and handler's swap function.
  - Returns the input amounts and output amount.
- `_handleFeesOnAmount(address token, uint256 amount, uint256 feePct)`
  - Checks if the fee percentage is greater than the maximum fee percentage.
  - Gets the fee amount.
  - Transfers the fee amount to the fee account.
  - Returns the fee amount.

5.3. Module: GlobalGuardian.sol

Function: `disable(byte[32] keyHash)`

This function is used to disable a functionality by updating the `functionalityIsDisabled` mapping with the `keyHash`. It is used in the `tradingEnabled` modifier.

Inputs

- `keyHash`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Value of the `keyHash`.

Branches and code coverage

Intended branches

- Updates the `functionalityIsDisabled` mapping with the `keyHash`.
  - ☑ Test coverage

Negative behavior

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

Function: `enable(byte[32] keyHash)`

This function is used to enable a functionality by deleting the `keyHash` from the `functionalityIsDisabled` mapping. It is used in the `tradingEnabled` modifier.

Inputs

- `keyHash`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Value of the `keyHash`.

Branches and code coverage

Intended branches

- Deletes the `keyHash` from the `functionalityIsDisabled` mapping.

☑ Test coverage

Negative behavior

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

5.4. Module: TradingVaultFactory.sol

Function: `addSignatureVerifier(address _signatureVerifier)`

This function is used to add a new signature verifier to the factory. Only the owner of the factory can add a new signature verifier.

Inputs

- `_signatureVerifier`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the signature verifier to be added.

Branches and code coverage

Intended branches

- Invokes `_addSignatureVerifier` to add the signature verifier.
  ☑ Test coverage

Negative behavior

- Reverts if the caller is not the owner of the factory.
  ☑ Negative test

Function: `createVaultWithPermission(DeployParams deployParams, bytes authorizedSignature)`

This function is used to create a new trading vault of the user. It deploys a new trading-vault contract using the provided deployment parameters and authorized signature. Minimal proxy pattern is used to deploy the vault contract.

Inputs

- `deployParams`
  - **Control**: Arbitrary.

- **Constraints**: Verified in `_verifyDeploymentParams`.
- **Impact**: Deployment parameters for the trading vault.

- `authorizedSignature`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Bytes of the authorized signature, which is used to verify the deployment parameters.

Branches and code coverage

Intended branches

- Invokes `_verifyDeploymentParams` to verify the deployment parameters and signature.
  - ☑ Test coverage
- Deploys a new trading-vault contract using the provided deployment parameters.
  - ☑ Test coverage
- Invokes the initialize function of the deployed trading-vault contract using the provided configuration.
  - ☑ Test coverage

Function: `createVault(address tradingVaultImplementation, byte[32] salt, BaseNativeWrapperConfig baseNativeWrapperConfig, CoreAccessControlConfig coreAccessControlConfig, CoreSimpleSwapConfig coreSimpleSwapConfig, CoreFeesConfig coreFeesConfig)`

This function is used to create a new trading vault of the user. It deploys a new trading-vault contract using the provided implementation and configuration. Minimal proxy pattern is used to deploy the vault contract.

Inputs

- `tradingVaultImplementation`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the trading-vault–implementation contract.
- `salt`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Value to be used as salt for deterministic address calculation.
- `baseNativeWrapperConfig`
  - **Control**: Arbitrary.
  - **Constraints**: None.

    - **Impact**: Address of `wrappedNativeAssetAddress`.
- `coreAccessControlConfig`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Addresses of admin, `definitiveAdmin`, definitive, and client roles.
- `coreSimpleSwapConfig`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Addresses of `swapHandlers`, which are used for swapping assets.
- `coreFeesConfig`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Addresses of `feeAccount`, which is used for collecting fees.

Branches and code coverage

Intended branches

- Generates a new salt by encoding the sender and provided salt.
    - ☑ Test coverage
- Deploys a new trading-vault contract using the generated salt.
    - ☑ Test coverage
- Invokes the initialize function of the deployed trading-vault contract using the provided configuration.
    - ☑ Test coverage

Function: `encodeDeploymentParams(DeployParams deployParams)`

This function is used to encode the deployment parameters to generate a hash. It is used to verify the signature of the deployment parameters.

Inputs

- `deployParams`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Deployment parameters for the trading vault.

Branches and code coverage

Intended branches

- Returns the `keccak256` hash of the deployment parameters.
  - ☑ Test coverage

Function: `encodeSalt(address sender, byte[32] deploymentSalt)`

This function is used to encode the sender and deployment salt to generate a new salt. It is used to calculate the deterministic address of the trading-vault contract.

Inputs

- `sender`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the sender.
- `deploymentSalt`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Value to be used as salt for deterministic address calculation.

Branches and code coverage

Intended branches

- Returns the `keccak256` hash of the deployment salt and sender.
  - ☑ Test coverage

Function: `getVaultAddress(address tradingVaultImplementation, byte[32] deploymentSalt)`

This function is used to get the address of the trading vault that would be deployed using the provided implementation and deployment salt. It calculates the deterministic address of the trading-vault contract.

Inputs

- `tradingVaultImplementation`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the trading-vault–implementation contract.
- `deploymentSalt`
  - **Control**: Arbitrary.

- **Constraints**: None.
- **Impact**: Value to be used as salt for deterministic address calculation.

Branches and code coverage

Intended branches

- Returns the deterministic address of the trading vault contract using the provided implementation and deployment salt.
  - ☑ Test coverage

Negative behavior

- Reverts if the address of the trading-vault implementation is zero.
  - ☑ Negative test

Function: `removeSignatureVerifier(address _signatureVerifier)`

This function is used to remove a signature verifier from the factory. Only the owner of the factory can remove a signature verifier.

Inputs

- `_signatureVerifier`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the signature verifier to be removed.

Branches and code coverage

Intended branches

- Invokes `_removeSignatureVerifier` to remove the signature verifier.
  - ☑ Test coverage

Negative behavior

- Reverts if the caller is not the owner of the factory.
  - ☑ Negative test

Function: `_addSignatureVerifier(address _signatureVerifier)`

This function is used to add a new signature verifier to the factory. It is an internal function and called by the `addSignatureVerifier` function.

Inputs

- `_signatureVerifier`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the signature verifier to be added.

Branches and code coverage

Intended branches

- Updates the mapping of signature verifiers as true.
  - ☑ Test coverage

Negative behavior

- Reverts if the address of the signature verifier is zero.
  - ☑ Negative test

Function: `_removeSignatureVerifier(address _signatureVerifier)`

This function is used to remove a signature verifier from the factory. It is an internal function and called by the `removeSignatureVerifier` function.

Inputs

- `_signatureVerifier`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the signature verifier to be removed.

Branches and code coverage

Intended branches

- Updates the mapping of signature verifiers as false.
  - ☑ Test coverage

Negative behavior

- Reverts if the address of the signature verifier is zero.
  - ☑ Negative test

Function: `_verifyDeploymentParams(DeployParams deployParams, bytes signature)`

This function is used to verify the deployment parameters and signature. It checks if the trading-vault implementation address is valid, the deadline is not exceeded, and the chain ID matches the current chain ID. It also verifies the signature using the provided deployment parameters.

Inputs

- `deployParams`
  - **Control**: Arbitrary.
  - **Constraints**: Checked in the function and reverted if not met.
  - **Impact**: Deployment parameters for the trading vault.
- `signature`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Bytes of the signature to be verified.

Branches and code coverage

Intended branches

- Invokes `SignatureVerifier.verifySignature` to verify the signature, checking that the signer is a signature verifier and the signature is valid.
  - ☑ Test coverage

Negative behavior

- Reverts if the trading-vault–implementation address is zero.
  - ☑ Negative test
- Reverts if the deadline is exceeded.
  - ☑ Negative test
- Reverts if the chain ID does not match the current chain ID or the provided chain ID is not the maximum value.
  - ☑ Negative test

6. Assessment Results

At the time of our assessment, the reviewed code was deployed to the Ethereum Mainnet, Arbitrum, Avalanche, Base, Optimism, and Polygon.

During our assessment on the scoped Definitive contracts, we discovered one finding, which was of low impact. Definitive Finance, Inc acknowledged the finding and implemented a fix.

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.