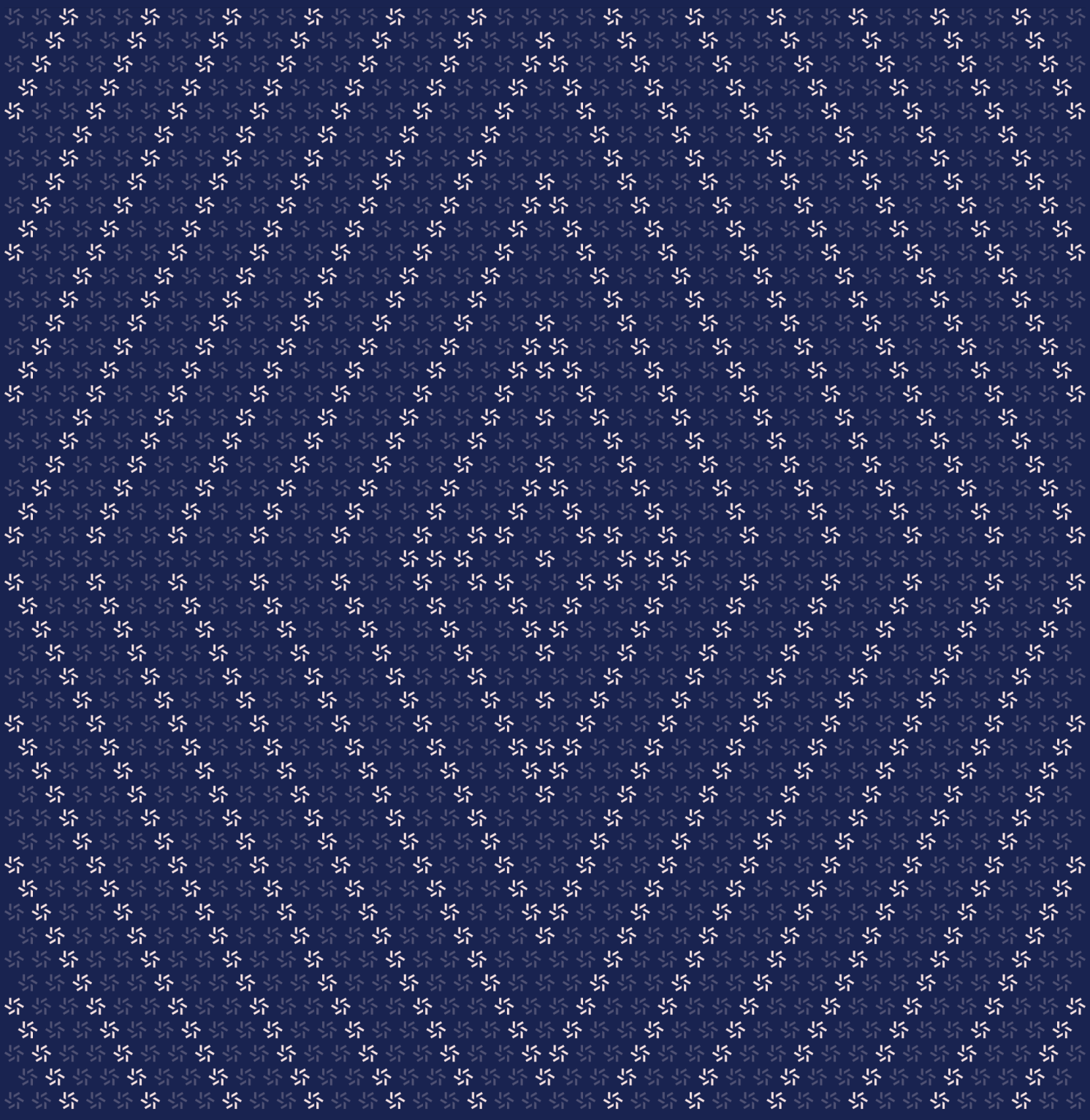
Zellic

Prepared for
The RabbitX team
RabbitX

Prepared by
Qingying Jie
Sylvain Pelissier
Zellic

November 13, 2024

Ethereum and Blast Exchanges
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 RabbitX from October 28th to October 30th, 2024. During this engagement, Zellic reviewed Ethereum and Blast exchanges' 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 the transfer calls correctly handled?
- Is the EIP-712 verification correctly implemented?
- Could an on-chain attacker drain the contracts?

1.3. Non-goals and Limitations

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

- Off-chain withdrawal signature generation
- Front-end 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.

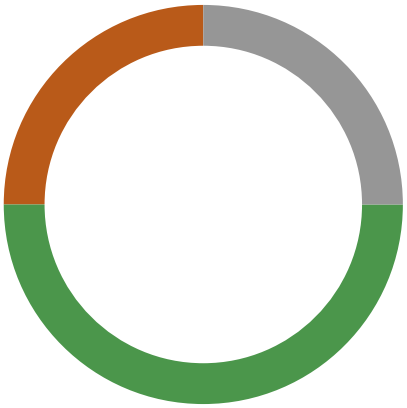
1.4. Results

During our assessment on the scoped Ethereum and Blast exchanges' contracts, we discovered eight findings. No critical issues were found. Two findings were of high impact, four were of low impact, and the remaining findings were informational in nature.

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

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	2
Medium	0
Low	4
Informational	2



All findings have been remediated by the RabbitX team, except 3.8, which was acknowledged.

2. Introduction

2.1. About Ethereum and Blast Exchanges

RabbitX contributed the following description of Ethereum and Blast exchanges:

> RabbitX is a multichain high performance orderbook leverage futures exchange. RabbitX combines cutting-edge technology with lightning speed, processing 10,000+ orders per second with transaction confirmations under 0.002 seconds.

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:

Ethereum and Blast Exchanges Contracts

Type	Solidity
Platform	EVM-compatible
Target	upgradeable-contracts
Repository	https://gitlab.com/stripsdev/upgradeable-contracts ↗
Version	`bec3e5235a9318492478d34d3883a230f3a1d7a3`

Programs	`BfxDepositU`
	`BfxU`
	`BfxVaultU`
	`EIP712VerifierU`
	`IPoolDeposit2`
	`IVault2`
	`ImportTimelockController`
	`Lock`
	`PoolDepositU`
	`RabbitU`
	`USDBFX`
	`VaultU`

2.4. Project Overview

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

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io ↗

Chad McDonald
Engagement Manager
chad@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Qingying Jie
Engineer
qingying@zellic.io ↗

Sylvain Pelissier
Engineer
sylvain@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

October 28, 2024	Start of primary review period
October 30, 2024	End of primary review period

3. Detailed Findings

3.1. Native tokens are not transferred to the exchange contracts

Target	VaultU, BfxVaultU		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

The vault contracts serve as a management layer for user permissions and staking operations. A user is able to stake tokens or ETH directly. For the first case of token staking, the `stakeToken` function is called, transferring the tokens to the exchange contracts — here, for example, in the contract BfxVaultU:

```
function stakeToken(uint256 amount, address token) public {
    require(supportedTokens[token], "UNSUPPORTED_TOKEN");
    require(amount >= minStakes[token], "AMOUNT_TOO_SMALL");
    string memory stakeId = allocateStakeId();
    emit Stake(stakeId, msg.sender, amount, token);
    uint256 prevBalance = IERC20(token).balanceOf(bfx);
    require(
        makeTransferFrom(msg.sender, bfx, amount, token),
        "TRANSFER_FAILED"
    );
    uint256 newBalance = IERC20(token).balanceOf(bfx);
    require(newBalance == amount + prevBalance, "NOT_ENOUGH_TRANSFERRED");
}
```

However, during a stake of ETH to the vault contract, the `handleReceivedNative` function is called:

```
function handleReceivedNative() internal {
    address native = address(0);
    require(supportedTokens[native], "UNSUPPORTED_TOKEN");
    uint256 minDeposit = minDeposits[native];
    require(msg.value >= minDeposit, "AMOUNT_TOO_SMALL");
    string memory depositId = allocateDepositId();
    emit Deposit(depositId, msg.sender, msg.value, native);
}
```

The ETH amount is received but not transferred to the exchange contracts. It stays in the vault contracts.

Impact

The user funds will not be transferred to the exchange contracts, and thus staking would not happen. The user funds will be locked in the vault contracts until withdrawn by the owner.

Recommendations

We recommend transferring the user ETH to the exchange contracts.

Remediation

This issue has been acknowledged by RabbitX, and a fix was implemented in commit `a7fa5c60`↗.

3.2. No test suite

Target	All scoped contracts		
Category	Code Maturity	**Severity**	High
Likelihood	N/A	**Impact**	High

Description

When building a project with multiple moving parts and dependencies, comprehensive testing is essential. This includes testing for both positive and negative scenarios. Positive tests should verify that each function's side effect is as expected, while negative tests should cover every revert, preferably in every logical branch.

There is currently no test suite for this project. It is important to test the invariants required for ensuring security.

Impact

This code has not undergone any testing, increasing the likelihood of potential bugs.

Recommendations

We recommend building a rigorous test suite to ensure that the system operates securely and as intended.

Good test coverage has multiple effects.

- It finds bugs and design flaws early (preaudit or prerelease).
- It gives insight into areas for optimization (e.g., gas cost).
- It displays code maturity.
- It bolsters customer trust in your product.
- It improves understanding of how the code functions, integrates, and operates — for developers and auditors alike.
- It increases development velocity long-term.

The last point seems contradictory, given the time investment to create and maintain tests. To expand upon that, tests help developers trust their own changes. It is difficult to know if a code refactor — or even just a small one-line fix — breaks something if there are no tests. This is especially true for new developers or those returning to the code after a prolonged absence. Tests have your back here. They are an indicator that the existing functionality *most likely* was not broken by your change to the code.

Remediation

This issue has been acknowledged by RabbitX, and a fix was implemented in commit a7fa5c60↗.

3.3. The function `supportToken` does not configure the yield mode

Target	BfxU		
Category	Business Logic	**Severity**	Medium
Likelihood	Low	**Impact**	Low

Description

During the initialization, the contract BfxU uses the mapping `rebasingTokens` to record whether a token is rebasing. If the token is rebasing, it then calls its function `configure` to set the yield mode.

```
function initialize(
    // [...]
    address _defaultToken,
    uint256 _minDeposit,
    bool _rebasing,
    address[] memory _otherTokens,
    uint256[] memory _minDeposits,
    bool[] memory _rebasingTokens
) public initializer {
    // [...]
    defaultToken = _defaultToken;
    IERC20Rebasing(_defaultToken).configure(YieldMode.CLAIMABLE);
    supportedTokens[_defaultToken] = true;
    rebasingTokens[_defaultToken] = _rebasing;
    if (_rebasing) {
        IERC20Rebasing(_defaultToken).configure(YieldMode.CLAIMABLE);
    }
    minDeposits[_defaultToken] = _minDeposit;
    for (uint256 i = 0; i < _otherTokens.length; i++) {
        address token = _otherTokens[i];
        supportedTokens[token] = true;
        minDeposits[token] = _minDeposits[i];
        rebasingTokens[token] = _rebasingTokens[i];
        if (_rebasingTokens[i]) {
            IERC20Rebasing(token).configure(YieldMode.CLAIMABLE);
        }
    }
    // [...]
}
```

However, when adding a token through the function `supportToken`, it does not record whether the token is rebasing, nor does it configure the yield mode.

```
function supportToken(
    address token,
    uint256 minDeposit
) external onlyOwner {
    supportedTokens[token] = true;
    minDeposits[token] = minDeposit;
    emit SupportToken(token, minDeposit);
}
```

Impact

For the native token on Blast, the yield will be lost if the yield mode is not configured.

For a stable token on Blast, if it is rebasing, the default yield mode will cause balance rebases. But the contract BfxU does not support rebasing tokens, which causes issues with internal accounting.

Recommendations

Consider adding a `rebasing` parameter to the function `supportToken`, recording its value to `rebasingTokens`, and configuring the yield mode when `rebasing` is true.

Remediation

This issue has been acknowledged by RabbitX, and a fix was implemented in commit `a7fa5c60` ↗.

3.4. Calling the function `configure` regardless of whether the token is rebasing

Target	BfxU		
Category	Coding Mistakes	**Severity**	Low
Likelihood	N/A	**Impact**	Low

Description

Some tokens on Blast are rebasing, and their yield mode can be configured by calling the function `configure`.

If the default token is rebasing, the contract BfxU will configure its yield mode during the initialization. However, there is an extra `configure` function call that ignores the value of `_rebasing` in the function `initialize`.

```
function initialize(
    // [...]
    address _defaultToken,
    uint256 _minDeposit,
    bool _rebasing,
    // [...]
) public initializer {
    // [...]
    defaultToken = _defaultToken;
    IERC20Rebasing(_defaultToken).configure(YieldMode.CLAIMABLE);
    supportedTokens[_defaultToken] = true;
    rebasingTokens[_defaultToken] = _rebasing;
    if (_rebasing) {
        IERC20Rebasing(_defaultToken).configure(YieldMode.CLAIMABLE);
    }
    // [...]
}
```

Impact

If the token is not rebasing, the call to the function `configure` may fail, causing the transaction to revert.

Recommendations

Consider removing the call to the function `configure` that is not within the if condition.

Remediation

This issue has been acknowledged by RabbitX, and a fix was implemented in commit `a7fa5c60`↗.

3.5. Use of identical domain separators

Target	BfxU, RabbitU		
Category	Coding Mistakes	**Severity**	Low
Likelihood	N/A	**Impact**	Low

Description

BfxU and RabbitU contracts both use the EIP712VerifierU contract for signature verification. During the initialization, the contracts call the function `__EIP712VerifierU_init`, which takes as input a domain separator to prevent replaying the signature between the Blast and Ethereum chains:

```
function __EIP712VerifierU_init(string memory domainName,
    string memory version, address signer) internal initializer {
    require(signer != address(0), "ZERO_SIGNER");
    __EIP712_init(domainName, version);
    external_signer = signer;
}
```

This domain separator is hashed together with a chain ID and contract address by `_buildDomainSeparator` to build the message digest during signature verification:

```
function _buildDomainSeparator() private view returns (bytes32) {
    return keccak256(abi.encode(TYPE_HASH, _EIP712NameHash(),
    _EIP712VersionHash(), block.chainid, address(this)));
}
```

However, both BfxU and RabbitU contracts use the same domain separator at initialization:

```
function initialize(
    // [...]
) public initializer {
    __Ownable_init(_owner);
    __UUPSUpgradeable_init();

    EIP712VerifierU.__EIP712VerifierU_init(
        "RabbitXWithdrawal",
        "1",
        _signer
    );
```

```
    // [...]
}
```

This prevents the purpose of using a domain separator to thwart replay attacks.

Impact

In the current implementation, the chain IDs and the contract addresses are different between contracts and, thus, prevent the replay of the signature, but using the same domain separator for the two different contracts is not best practice.

Recommendations

We recommend to change the BfxU domain separator at initialization according to the name of the application.

Remediation

This issue has been acknowledged by RabbitX, and a fix was implemented in commit a7fa5c60↗.

3.6. Lack of storage gap in the contract EIP712VerifierU

Target	EIP712VerifierU		
Category	Business Logic	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

When using upgradable contracts, storage gaps are used for reserving storage slots in a base contract, allowing upgrade of that contract to use up those slots without affecting the storage layout. However, the upgradable contract EIP712VerifierU does not have a storage-gap variable, and it is inherited by other contracts.

Impact

If new storage variables are added to the contract EIP712VerifierU in the future, it will affect the storage variables in the child contract.

Recommendations

Consider adding a gap variable to be safe against storage collisions.

Remediation

This issue has been acknowledged by RabbitX, and a fix was implemented in commit `a7fa5c60` ↗.

3.7. Incorrect event-parameter order

Target	BfxDepositU		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

In the contract BfxDepositU, the parameters passed to the event `Deposit` are in a different order than the order defined in the interface IPoolDeposit2. The last two parameters of the event are defined as `token` and `poolId`, but BfxDepositU passes them as `poolId` and `token`.

```
event Deposit(
    string id,
    address indexed trader,
    uint256 amount,
    address indexed token,
    uint256 indexed poolId
);
```

```
function individualDeposit(address contributor, uint256 amount) external {
    // [...]
    emit Deposit(depositId, contributor, amount, 0, defaultToken);
    // [...]
}
```

Impact

This will cause the contract BfxDepositU to fail to compile.

Recommendations

Consider modifying the parameter order to match the definition.

Remediation

This issue has been acknowledged by RabbitX, and a fix was implemented in commit a7fa5c60↗.

3.8. Missing zero-address check

Target	BfxU, BfxVaultU, BfxDepositU, RabbitU, VaultU, PoolDepositU		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `timelock` address is allowed to upgrade the contracts, and its address is set during contract initialization. However, the address value is not checked to be nonzero. This address cannot be changed later.

This remark applies also for other addresses like `owner`, `_defaultToken`, or `_signer`. However, some of them may be changed later by the `timelock` address.

Impact

If by accident the `timelock` is initialized to zero, the contracts will not be upgradable and the owner cannot be changed.

Recommendations

We recommend to implement zero-address checks in the `initialize` functions.

Remediation

This issue has been acknowledged by RabbitX.

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. Lack of documentation

There are certain areas in the code where some mechanisms are not documented. For example, the role `onlyTimelock` or the modification of the signature verification would benefit from documentation about the rationales behind the design choices and the interactions with other parts of the system.

Code maturity is very important in high-assurance projects. Undocumented code may result in developer confusion, potentially leading to future bugs should the code be modified later on. In general, a lack of documentation impedes the auditors' and external developers' ability to read, understand, and extend the code. The problem is also carried over if the code is ever forked or reused.

We recommend adding more comments to the code — especially comments that tie operations in code to locations in the documentation and brief comments to reaffirm developers' understanding.

4.2. Centralization risks

There are four types of privileged accounts for the contracts:

1. The timelock
2. The claimer
3. The signer
4. The owner

The timelock contract can authorize an upgrade of the contracts according to the Universal Upgradeable Proxy Standard (UUPS) scheme but have no other special powers.

The claimer is able to claim gas from the Blast contracts.

The signer contract can sign withdrawal requests. The signature-generation part is done off chain and was not reviewed. However, it allows the signer to withdraw all assets from Blast and Rabbit contracts.

The owner is able to change the supported tokens.

The above introduces centralization risks that users should be aware of, as it grants a single point of control over the system.

We recommend that these centralization risks be clearly documented for users so that they are aware of the extent of the owner's control over the contract. This can help users make informed decisions about their participation in the project. Additionally, clear communication about the circumstances in which the owner may exercise these powers can help build trust and transparency with users. Therefore, it is recommended to implement additional measures to mitigate these risks, such as implementing a multi-signature requirement for owner access.

4.3. Enforce a more recent Solidity version

Some contracts specify Solidity version ^0.8.0 to be used, which allows compilation with any version of Solidity from 0.8.0 up to the latest release. If possible, enforcing the use of the latest Solidity version, ^0.8.28, would prevent introducing bugs present in the previous versions and may add the latest optimizations to the contract.

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: BfxDepositU.sol

Function: `depositNative(address contributor)`

This function emits the `Deposit` event and transfers the received native tokens to the `bfx` on behalf of the `contributor`.

Inputs

- `contributor`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: The contributor of the deposit.

Branches and code coverage

Intended branches

- Transfer the received native tokens to the `bfx`.
  - ☐ Test coverage
- Increase the `nextDepositNum` by one when the transfer succeeds.
  - ☐ Test coverage

Negative behavior

- Revert when the native token is not supported.
  - ☐ Negative test
- Revert when the amount is smaller than `minDeposits[native]`.
  - ☐ Negative test
- Revert when the transfer fails.
  - ☐ Negative test

Function: `depositToken(address contributor, uint256 amount, address token)`

This function emits the `Deposit` event and transfers the supported tokens from the caller to the `bfx` on behalf of the `contributor`.

Inputs

- `contributor`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: The contributor of the deposit.
- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The value must not be smaller than `minDeposits[token]`, and the caller must have enough tokens to transfer.
  - **Impact**: The amount to transfer.
- `token`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The token must be supported by this contract.
  - **Impact**: The token to transfer.

Branches and code coverage

Intended branches

- Transfer the supported tokens from the caller to the `bfx`.
  - ☐ Test coverage
- Increase the `nextDepositNum` by one when the transfer succeeds.
  - ☐ Test coverage

Negative behavior

- Revert when the token is not supported.
  - ☐ Negative test
- Revert when the amount is smaller than `minDeposits[token]`.
  - ☐ Negative test
- Revert when the transfer fails.
  - ☐ Negative test

Function: `individualDeposit(address contributor, uint256 amount)`

This function emits the `Deposit` event and transfers the default tokens from the caller to the `bfx` on behalf of the `contributor`.

Inputs

- `contributor`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: The contributor of the deposit.
- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The value must not be smaller than `minDeposits[defaultToken]`, and the caller must have enough tokens to transfer.
  - **Impact**: The amount to transfer.

Branches and code coverage

Intended branches

- Transfer the default tokens from the caller to the `bfx`.
  - ☐ Test coverage
- Increase the `nextDepositNum` by one when the transfer succeeds.
  - ☐ Test coverage

Negative behavior

- Revert when the amount is smaller than `minDeposits[defaultToken]`.
  - ☐ Negative test
- Revert when the transfer fails.
  - ☐ Negative test

Function: `pooledDepositCommon(Contribution[] contributions, address token)`

This function emits the `Deposit` event for each contributor and the `PooledDeposit` event for this deposit. It returns the total amount that needs to be transferred.

Inputs

- `contributions`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The array length must not be greater than `MAX_CONTRIBUTIONS`. Each contribution must not have an amount smaller than `minDeposits[token]`.
  - **Impact**: The contributors and amounts of each deposit.
- `token`
  - **Control**: Fully controlled by the caller.

- **Constraints**: The token must be supported by this contract.
- **Impact**: The token to deposit.

Branches and code coverage

Intended branches

- Increase the `nextStakeNum` by one for each deposit.
  - ☐ Test coverage
- Increase the `nextPoolId` by one for the pooled deposit.
  - ☐ Test coverage

Negative behavior

- Revert when the token is not supported.
  - ☐ Negative test
- Revert when the length of the `contributions` array is greater than `MAX_CONTRIBUTIONS`.
  - ☐ Negative test
- Revert when a deposit amount is smaller than `minDeposits[token]`.
  - ☐ Negative test

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

This function transfers the native tokens from this contract to the given address. The caller must be the owner.

Inputs

- `to`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The value must not be zero.
  - **Impact**: The address to transfer the tokens to.
- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: This contract must have enough native tokens, and this value must be greater than zero.
  - **Impact**: The amount of native tokens to transfer.

Branches and code coverage

Intended branches

- Transfer the native tokens to the given address.

- ☐ Test coverage

Negative behavior

- Revert when the caller is not the owner.
  - ☐ Negative test

Function: `withdrawTokensTo(address to, uint256 amount, address token)`

This function transfers the supported tokens from this contract to the given address. The caller must be the owner.

Inputs

- `to`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The value must not be zero.
  - **Impact**: The address to transfer the tokens to.
- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: This contract must have enough tokens, and this value must be greater than zero.
  - **Impact**: The amount of tokens to transfer.
- `token`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The token must be supported by this contract.
  - **Impact**: The token to transfer.

Branches and code coverage

Intended branches

- Transfer the supported tokens to the given address.
  - ☐ Test coverage

Negative behavior

- Revert when the token is not supported.
  - ☐ Negative test
- Revert when the caller is not the owner.
  - ☐ Negative test

5.2. Module: BfxU.sol

Function: `depositToken()`

The function handles the deposit of tokens to the Blast exchange. This is a call to the `handleDeposit` function. Internally, the token passed as a parameter is checked to be supported.

The `deposit` function calls the same `handleDeposit` function with the `defaultToken` address passed as parameter. The same threat model applies to it.

Inputs

- `amount`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: The value must not be smaller than the minimum deposit, and the transfer must be successful. The final contract balance must be correct after the transfer.
  - **Impact**: The amount to deposit.
- `token`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: The token address is checked to be supported.
  - **Impact**: The token to deposit.

Branches and code coverage (including function calls)

Intended branches

- Transfer some `defaultToken` from the sender to the contract.
  - ☐ Test coverage
- The contract balance is correctly updated when the transfer succeeds.
  - ☐ Test coverage

Negative behavior

- Revert when the amount is smaller than the minimum deposit.
  - ☐ Negative test
- Revert when the transfer fails.
  - ☐ Negative test
- Revert when the token is not supported.
  - ☐ Negative test

Function call analysis

- `depositToken -> handleDeposit(uint256 amount, address token)`

- **External/Internal?** Internal.
- **Argument control?** Both arguments are controlled.
- **Impact**: Handle underlying checks on arguments and perform the transfer. Ensure the balance is updated correctly.

Function: `function withdraw()`

This function withdraws an amount of the `defaultToken` if the ECDSA signature verifies the given `id`, `trader`, and `amount` on the Blast chain.

The `withdrawToken` and `withdrawNative` functions work similarly, except that the token address is included in the signature. The same threat model applies to them.

Inputs

- `id`
  - **Control**: Controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Preventing replay attacks.
- `trader`
  - **Control**: Controlled by the caller.
  - **Constraints**: The `trader` in the given signature must match.
  - **Impact**: Ensuring the address of the recipient was authorized.
- `amount`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: The `amount` in the given signature must match, and the value must be bigger than zero.
  - **Impact**: The amount to transfer.
- `v`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: The `v` argument of the ECDSA signature can be either 27 or 28, arbitrary to be compatible with the AWS KMS.
  - **Impact**: A part of the signature to verify.
- `r`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: This must be a part of the valid ECDSA signature for the given `id`, `trader`, and `amount`.
  - **Impact**: The first part of the signature to verify.
- `s`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: This must be a part of the valid ECDSA signature with the given `id`, `trader`, and `amount`.

- **Impact**: The second part of the signature to verify.

Branches and code coverage (including function calls)

Intended branches

- Verify the signature.
  - ☐ Test coverage
- Verify the signature with another `v` value.
  - ☐ Test coverage
- Send the `defaultToken` to the given `trader` and verify the balance is correct.
  - ☐ Test coverage

Negative behavior

- Revert when it fails to verify an incorrect signature.
  - ☐ Negative test
- Revert when the amount is zero.
  - ☐ Negative test
- Revert when the `id` is already withdrawn.
  - ☐ Negative test
- Revert when the signature is replayed from the Ethereum chain.
  - ☐ Negative test
- Revert when the transfer fails.
  - ☐ Negative test

Function call analysis

- `withdraw -> getDigest(id, trader, amount, defaultToken, false)`
  - **External/Internal?** Internal.
  - **Argument control?** `id`, `trader`, and `amount` are controlled.
  - **Impact**: Compute the hash of the parameters before the signature.
- `withdraw -> verify(digest, v, r, s)`
  - **External/Internal?** Internal.
  - **Argument control?** `v`, `r`, and `s` are controlled.
  - **Impact**: Verify the ECDSA signature.
- `withdraw -> makeTransfer(trader, amount, defaultToken)`
  - **External/Internal?** Internal.
  - **Argument control?** `trader` and `amount` are controlled but must be signed correctly.
  - **Impact**: Transfer the token to the trader.

5.3. Module: BfxVaultU.sol

Function: `stakeNative()`

This function calls the `handleReceivedNative` function, which emits the `Stake` event.

Branches and code coverage

Intended branches

- Increase the `nextStakeNum` by one.
  - ☐ Test coverage

Negative behavior

- Revert when the native token is not supported.
  - ☐ Negative test
- Revert when the `msg.value` is smaller than `minStakes[native]`.
  - ☐ Negative test

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

This function emits the `Stake` event and transfers supported tokens from the caller to the `bfx`.

Inputs

- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The value must not be smaller than `minStakes[token]`, and the caller must have enough tokens to transfer.
  - **Impact**: The amount to transfer.
- `token`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The token must be supported by this contract.
  - **Impact**: The token to transfer.

Branches and code coverage

Intended branches

- Transfer the supported tokens from the caller to the `bfx`.
  - ☐ Test coverage
- Increase the `nextStakeNum` by one when the transfer succeeds.
  - ☐ Test coverage

Negative behavior

- Revert when the token is not supported.
  - ☐ Negative test
- Revert when the amount is smaller than `minStakes[token]`.
  - ☐ Negative test
- Revert when the transfer fails.
  - ☐ Negative test

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

This function transfers the native tokens from this contract to the given address. The caller must be the owner.

Inputs

- `to`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The value must not be zero.
  - **Impact**: The address to transfer the tokens to.
- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: This contract must have enough native tokens, and this value must be greater than zero.
  - **Impact**: The amount of native tokens to transfer.

Branches and code coverage

Intended branches

- Transfer the native tokens to the given address.
  - ☐ Test coverage

Negative behavior

- Revert when the caller is not the owner.
  - ☐ Negative test

Function: `withdrawTokensTo(address to, uint256 amount, address token)`

This function transfers the supported tokens from this contract to the given address. The caller must be the owner.

Inputs

- `to`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The value must not be zero.
  - **Impact**: The address to transfer the tokens to.
- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: This contract must have enough tokens, and this value must be greater than zero.
  - **Impact**: The amount of tokens to transfer.
- `token`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The token must be supported by this contract.
  - **Impact**: The token to transfer.

Branches and code coverage

Intended branches

- Transfer the supported tokens to the given address.
  - ☐ Test coverage

Negative behavior

- Revert when the token is not supported.
  - ☐ Negative test
- Revert when the caller is not the owner.
  - ☐ Negative test

5.4. Module: PoolDepositU.sol

Function: `depositNative(address contributor)`

This function emits the `Deposit` event and transfers the received native tokens to the `rabbit` on behalf of the `contributor`.

Inputs

- `contributor`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: The contributor of the deposit.

Branches and code coverage

Intended branches

- Transfer the received native tokens to the `rabbit`.
  - ☐ Test coverage
- Increase the `nextDepositNum` by one when the transfer succeeds.
  - ☐ Test coverage

Negative behavior

- Revert when the native token is not supported.
  - ☐ Negative test
- Revert when the amount is smaller than `minDeposits[native]`.
  - ☐ Negative test
- Revert when the transfer fails.
  - ☐ Negative test

Function: `depositToken(address contributor, uint256 amount, address token)`

This function emits the `Deposit` event and transfers the supported tokens from the caller to the `rabbit` on behalf of the `contributor`.

Inputs

- `contributor`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: The contributor of the deposit.
- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The value must not be smaller than `minDeposits[token]`, and the caller must have enough tokens to transfer.
  - **Impact**: The amount to transfer.
- `token`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The token must be supported by this contract.
  - **Impact**: The token to transfer.

Branches and code coverage

Intended branches

- Transfer the supported tokens from the caller to the `rabbit`.
  - ☐ Test coverage
- Increase the `nextDepositNum` by one when the transfer succeeds.
  - ☐ Test coverage

Negative behavior

- Revert when the token is not supported.
  - ☐ Negative test
- Revert when the amount is smaller than `minDeposits[token]`.
  - ☐ Negative test
- Revert when the transfer fails.
  - ☐ Negative test

Function: `individualDeposit(address contributor, uint256 amount)`

This function emits the `Deposit` event and transfers the default tokens from the caller to the `rabbit` on behalf of the `contributor`.

Inputs

- `contributor`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: The contributor of the deposit.
- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The value must not be smaller than `minDeposits[defaultToken]`, and the caller must have enough tokens to transfer.
  - **Impact**: The amount to transfer.

Branches and code coverage

Intended branches

- Transfer the default tokens from the caller to the `rabbit`.
  - ☐ Test coverage
- Increase the `nextDepositNum` by one when the transfer succeeds.
  - ☐ Test coverage

Negative behavior

- Revert when the amount is smaller than `minDeposits[defaultToken]`.
  - ☐ Negative test

- Revert when the transfer fails.
  - ☐ Negative test

Function: `pooledDepositCommon(Contribution[] contributions, address token)`

This function emits the `Deposit` event for each contributor and the `PooledDeposit` event for this deposit. It returns the total amount that needs to be transferred.

Inputs

- `contributions`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The array length must not be greater than `MAX_CONTRIBUTIONS`. Each contribution must not have an amount smaller than `minDeposits[token]`.
  - **Impact**: The contributors and amounts of each deposit.
- `token`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The token must be supported by this contract.
  - **Impact**: The token to deposit.

Branches and code coverage

Intended branches

- Increase the `nextStakeNum` by one for each deposit.
  - ☐ Test coverage
- Increase the `nextPoolId` by one for the pooled deposit.
  - ☐ Test coverage

Negative behavior

- Revert when the token is not supported.
  - ☐ Negative test
- Revert when the length of the `contributions` array is greater than `MAX_CONTRIBUTIONS`.
  - ☐ Negative test
- Revert when a deposit amount is smaller than `minDeposits[token]`.
  - ☐ Negative test

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

This function transfers the native tokens from this contract to the given address. The caller must be the owner.

Inputs

- `to`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The value must not be zero.
  - **Impact**: The address to transfer the tokens to.
- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: This contract must have enough native tokens, and this value must be greater than zero.
  - **Impact**: The amount of native tokens to transfer.

Branches and code coverage

Intended branches

- Transfer the native tokens to the given address.
  - ☐ Test coverage

Negative behavior

- Revert when the caller is not the owner.
  - ☐ Negative test

Function: `withdrawTokensTo(address to, uint256 amount, address token)`

This function transfers the supported tokens from this contract to the given address. The caller must be the owner.

Inputs

- `to`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The value must not be zero.
  - **Impact**: The address to transfer the tokens to.
- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: This contract must have enough tokens, and this value must be

greater than zero.
    - **Impact**: The amount of tokens to transfer.
- `token`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: The token must be supported by this contract.
    - **Impact**: The token to transfer.

Branches and code coverage

Intended branches

- Transfer the supported tokens to the given address.
    - ☐ Test coverage

Negative behavior

- Revert when the token is not supported.
    - ☐ Negative test
- Revert when the caller is not the owner.
    - ☐ Negative test

5.5. Module: RabbitU.sol

Function: `depositToken()`

The function handles the deposit of tokens to the RabbitX exchange. This is a call to the `handleDeposit` function. Internally, the token passed as a parameter is checked to be supported.

The `deposit` function calls the same `handleDeposit` function with the `defaultToken` address passed as a parameter. The same threat model applies to it.

Inputs

- `amount`
    - **Control**: Completely controlled by the caller.
    - **Impact**: The amount to deposit. The value must not be smaller than the minimum deposit, and the transfer must be successful. The final contract balance must be correct after the transfer.
- `token`
    - **Control**: Completely controlled by the caller.
    - **Impact**: The token to deposit. The token address is checked to be supported.

Branches and code coverage (including function calls)

Intended branches

- Transfer some `defaultToken` from the sender to the contract.
  - ☐ Test coverage
- The contract balance is correctly updated when the transfer succeeds.
  - ☐ Test coverage

Negative behavior

- Revert when the amount is smaller than the minimum deposit.
  - ☐ Negative test
- Revert when the transfer fails.
  - ☐ Negative test
- Revert when the token is not supported.
  - ☐ Negative test

Function call analysis

- `depositToken -> handleDeposit(uint256 amount, address token)`
  - **External/Internal?** Internal.
  - **Argument control?** Both arguments are controlled.
  - **Impact**: Handle underlying checks on arguments and perform the transfer. Ensure the balance is updated correctly.

Function: `function withdraw()`

This function withdraws an amount of the `defaultToken` if the ECDSA signature verifies the given `id`, `trader`, and `amount`.

The `withdrawToken` and `withdrawNative` functions work similarly except that the token address is included in the signature. The same threat model applies to them.

Inputs

- `id`
  - **Control**: Controlled by the caller.
  - **Constraints**: The mapping `processedWithdrawals` of the given `id` must not be true. The `id` in the given signature must match.
  - **Impact**: Prevent replay attacks.
- `trader`
  - **Control**: Controlled by the caller.
  - **Constraints**: The `trader` in the given signature must match.

    - **Impact**: Ensure the address of the recipient was authorized.
- `amount`
    - **Control**: Completely controlled by the caller.
    - **Constraints**: The `amount` in the given signature must match, and the value must be bigger than zero.
    - **Impact**: The amount to transfer.
- `v`
    - **Control**: Completely controlled by the caller.
    - **Constraints**: The `v` argument of the ECDSA signature can be either 27 or 28, arbitrary to be compatible with the AWS KMS.
    - **Impact**: A part of the signature to verify.
- `r`
    - **Control**: Completely controlled by the caller.
    - **Constraints**: This must be a part of the valid ECDSA signature for the given `id`, `trader`, and `amount`.
    - **Impact**: The first part of the signature to verify.
- `s`
    - **Control**: Completely controlled by the caller.
    - **Constraints**: This must be a part of the valid ECDSA signature with the given `id`, `trader`, and `amount`.
    - **Impact**: The second part of the signature to verify.

Branches and code coverage (including function calls)

Intended branches

- Verify the signature.
    - ☐ Test coverage
- Verify the signature with another `v` value.
    - ☐ Test coverage
- Send the `defaultToken` to the given `trader`, and verify the balance is correct.
    - ☐ Test coverage

Negative behavior

- Revert when it fails to verify an incorrect signature.
    - ☐ Negative test
- Revert when the amount is zero.
    - ☐ Negative test
- Revert when the `id` is already withdrawn.
    - ☐ Negative test
- Revert when the transfer fails.
    - ☐ Negative test

Function call analysis

- `withdraw -> getDigest(id, trader, amount, defaultToken, false)`
  - **External/Internal?** Internal.
  - **Argument control?** `id`, `trader`, and `amount` are controlled.
  - **Impact**: Compute the hash of the parameters before the signature.
- `withdraw -> verify(digest, v, r, s)`
  - **External/Internal?**: Internal.
  - **Argument control?**: `v`, `r`, and `s` are controlled.
  - **Impact**: Verify the ECDSA signature.
- `withdraw -> makeTransfer(trader, amount, defaultToken)`
  - **External/Internal?** Internal.
  - **Argument control?**: `trader` and `amount` are controlled but must be signed correctly.
  - **Impact**: Transfer the token to the trader.

5.6. Module: VaultU.sol

Function: `stakeNative()`

This function calls the `handleReceivedNative` function, which emits the `Stake` event.

Branches and code coverage

Intended branches

- Increase the `nextStakeNum` by one.
  - ☐ Test coverage

Negative behavior

- Revert when the native token is not supported.
  - ☐ Negative test
- Revert when the `msg.value` is smaller than `minStakes[native]`.
  - ☐ Negative test

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

This function emits the `Stake` event and transfers supported tokens from the caller to the `rabbitx`.

Inputs

- `amount`

    - **Control**: Fully controlled by the caller.
    - **Constraints**: The value must not be smaller than `minStakes[token]`, and the caller must have enough tokens to transfer.
    - **Impact**: The amount to transfer.
- `token`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: The token must be supported by this contract.
    - **Impact**: The token to transfer.

Branches and code coverage

Intended branches

- Transfer the supported tokens from the caller to the `rabbitx`.
    - ☐ Test coverage
- Increase the `nextStakeNum` by one when the transfer succeeds.
    - ☐ Test coverage

Negative behavior

- Revert when the token is not supported.
    - ☐ Negative test
- Revert when the amount is smaller than `minStakes[token]`.
    - ☐ Negative test
- Revert when the transfer fails.
    - ☐ Negative test

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

This function transfers the native tokens from this contract to the given address. The caller must be the owner.

Inputs

- `to`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: The value must not be zero.
    - **Impact**: The address to transfer the tokens to.
- `amount`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: This contract must have enough native tokens, and this value must be greater than zero.
    - **Impact**: The amount of native tokens to transfer.

Branches and code coverage

Intended branches

- Transfer the native tokens to the given address.
  - ☐ Test coverage

Negative behavior

- Revert when the caller is not the owner.
  - ☐ Negative test

Function: `withdrawTokensTo(address to, uint256 amount, address token)`

This function transfers the supported tokens from this contract to the given address. The caller must be the owner.

Inputs

- `to`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The value must not be zero.
  - **Impact**: The address to transfer the tokens to.
- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: This contract must have enough tokens, and this value must be greater than zero.
  - **Impact**: The amount of tokens to transfer.
- `token`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The token must be supported by this contract.
  - **Impact**: The token to transfer.

Branches and code coverage

Intended branches

- Transfer the supported tokens to the given address.
  - ☐ Test coverage

Negative behavior

- Revert when the caller is not the owner.
  - ☐ Negative test

6. Assessment Results

At the time of our assessment, the reviewed code was not deployed to Ethereum Mainnet and Blast.

During our assessment on the scoped Ethereum and Blast exchanges' contracts, we discovered eight findings. No critical issues were found. Two findings were of high impact, four were of low impact, and the remaining findings were informational in nature.

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.