Zellic

STFX

Smart Contract Security Assessment

April 13, 2023

Prepared for:

STFX

Prepared by:

Varun Verma and Ulrich Myhre

Zellic Inc.

Contents

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please contact us at hello@zellic.io.

1 Executive Summary

Zellic conducted a security assessment for STFX from March 21st to March 27th, 2023. During this engagement, Zellic reviewed STFX's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.1 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:

- Do fund managers have the ability to perform spot trades to open and close positions?
- Are there any ways in which fund managers could potentially misuse funds outside of the protocol's intended purposes?
- Is the proportion of investment returned to investors accurate and consistent with their investment amounts?
- Are the protocol's parameters and constraints rational considering the goals and objectives of the system?

1.2 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

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

Delays in receiving architectural design diagrams during the assessment made it difficult to quickly understand the system's design, resulting in a slower assessment process.

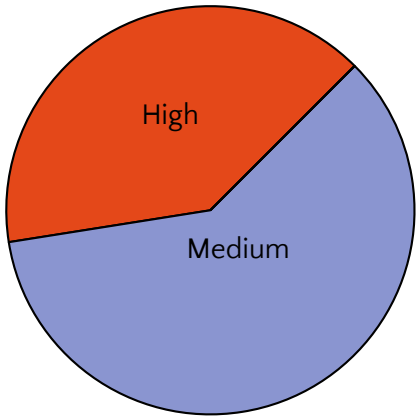
1.3 Results

During our assessment on the scoped STFX contracts, we discovered five findings. No critical issues were found. Of the five findings, two were of high impact and three were of medium impact.

Additionally, Zellic recorded its notes and observations from the assessment for STFX's benefit in the Discussion section (4) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	2
Medium	3
Low	0
Informational	0



2 Introduction

2.1 About STFX

STFX, Single Trade Finance Exchange, is a DeFi and SocialFi protocol for short-term asset management. It is a DeFi marketplace for investable trades.

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 the contracts' 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 code base 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, upgradeability 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.

2.3 Scope

The engagement involved a review of the following targets:

STFX Contracts

Repositories	https://github.com/STFX-IO/tokenomics-contracts
	https://github.com/STFX-IO/stfx-single-contract
Versions	tokenomics-contracts: `9296d8187afcfbe55b405f6321fd637d0bc38ca4`
	stfx-single-contract: `945899afea2fedde94b576bc75053802d8f8ec31`
Programs	• Spot.sol
	• Swap.sol
	• Trade.sol
	• Vesting.sol
	• VestingFactory.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 eight engineer-days. The assessment was conducted over the course of two calendar weeks.

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:

Varun Verma, Engineer
varun@zellic.io

Ulrich Myhre, Engineer
unblvr@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

March 21, 2023	Start of primary review period
March 22, 2023	Kick-off call
March 27, 2023	End of primary review period

3 Detailed Findings

3.1 ABI-encoded inputs can mismatch specified amount

- **Target**: Swap.sol
- **Category**: Coding Mistakes
- **Likelihood**: Medium
- **Severity**: High
- **Impact**: High

Description

A manager or admin can execute a swap via Uniswap's universal router. However, they can potentially cause a mismanagement of funds if they abi.encode a different value in the `inputs` parameter than what is specified in the `amountIn` parameter for the swap.

Impact

The following function permits the swap:

```
function swapUniversalRouter(
    address tokenIn,
    address tokenOut,
    uint160 amountIn,
    bytes calldata commands,
    bytes[] calldata inputs,
    ...
) external override onlyTrade returns (uint96) {
    ...
    if (deadline > 0) universalRouter.execute(commands, inputs, deadline);
    ...
}
```

As seen in this snippet, `universalRouter.execute(commands, inputs, deadline)` has no accordance to the `amountIn` parameter and thus `inputs`, which is supposed to encode the `amountIn`, can be a different value. The protocol uses `amountIn` for its internal accounting and therefore can become out of sync.

Recommendations

We recommend extracting the `amountIn` from the ABI-encoded `inputs` function param.

Remediation

STFX acknowledged and resolved the issue in fb58bb9f

3.2 Possible denial of service in `claim`

- **Target**: VestingFactory.sol
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Medium
- **Impact**: High

Description

When `vestingAddresses` attempt to `claim`, the system iterates through all addresses and sends funds accordingly. However, if the size of the `vestingAddresses` array becomes too large, a denial-of-service gas error can occur, preventing anyone from being able to claim funds.

Impact

The following code corresponds to the claim function:

```
for (uint256 i = 0; i < vestingAddresses.length;) {
    address v = vestingAddresses[i];
    if (!IVesting(v).cancelled()) {
        if (IVesting(v).totalClaimedAmount() < IVesting(v).amount()) {
            IVesting(v).claim();
        }
    }
    unchecked {++i;}
}
```

New vesting addresses can be added using the `createVestingStartingFrom` and `createVestingStartingFromNow` methods. However, if the treasury calls these methods excessively, a large number of vesting addresses may accumulate, which can prevent anyone from being able to claim. Unfortunately, there is no way to remove vesting addresses once they have been added.

Recommendations

We recommend exploring one of the following possibilities to address the issue:

1. Modify the `claim` function to take start and end indices, allowing users to claim their tokens in batches instead of all at once.
2. Implement a way to remove vesting addresses once they have been added to the system. This would prevent the accumulation of a large number of ad-

dresses that could lead to denial-of-service errors.

3. Set a maximum cap on the number of vesting addresses that can be added to the system. This would limit the potential for denial-of-service errors by preventing the system from becoming overloaded with too many vesting addresses.

Remediation

STFX acknowledged and resolved the issue in 6503abf8

3.3 Protocol does not check return value of ERC20 swaps

- **Target**: Swap.sol
- **Category**: Coding Mistakes
- **Likelihood**: Medium
- **Severity**: Medium
- **Impact**: Medium

Description

The ERC20 standard requires that transfer operations return a boolean success value indicating whether the operation was successful or not. Therefore, it is important to check the return value of the transfer function before assuming that the transfer was successful. This helps ensure that the transfer was executed correctly and helps avoid potential issues with lost or mishandled funds.

Impact

The protocol's internal accounting will record failed transfer operations as a success if the underlying ERC20 token does not revert on failure.

Recommendations

We recommend implementing one of the following solutions to ensure that ERC20 transfers are handled securely:

1. Utilize OpenZeppelin's `SafeERC20` transfer methods, which provide additional checks and safeguards to ensure the safe handling of ERC20 transfers.
2. Strictly whitelist ERC20 coins that do not return false on failure and revert. This will ensure that only safe and reliable ERC20 tokens are used within the protocol.

In general, it is important to exercise caution when integrating third-party tokens into the protocol. Tokens with hooks and atypical behaviors of the ERC20 standard can present security vulnerabilities that may be exploited by attackers. We recommend thoroughly researching and reviewing any tokens that are considered for integration and performing a comprehensive security review of the entire system to identify and mitigate any potential vulnerabilities.

Remediation

STFX acknowledged and resolved the issue in 67276712

3.4 High minimum investment amount

- **Target**: Spot.sol
- **Category**: Coding Mistakes
- **Likelihood**: Medium
- **Severity**: Medium
- **Impact**: Medium

Description

While the minimal investment permitted by the protocol is intended to establish a reasonable lower bound for investment amounts, the current restriction of 1e18 can be excessive for certain tokens, such as wBTC, particularly during a bull market when prices are high. This can make it difficult for everyday users to enter the protocol and limits the accessibility of the system.

Impact

The following code sets a lower bound on the minimal investment amount:

```
function addMinInvestmentAmount(address _token, uint96 _amount)
    external override onlyOwner {
    if (_amount < 1e18) revert ZeroAmount();
    minInvestmentAmount[_token] = _amount;
    emit MinInvestmentAmountChanged(_token, _amount);
}
```

The current minimal investment amount of 1e18 may be too high for certain high-value coins such as wBTC, where this amount equates to approximately $30K USD and could potentially be even higher in the future. This high barrier to entry may limit accessibility for everyday users and could ultimately impact the growth and sustainability of the system.

Recommendations

We recommend removing a minimal investment amount.

Remediation

STFX acknowledged and resolved the issue in ca4e2157

3.5 Inconsistent coding conventions

- **Target**: All Contracts
- **Category**: Coding Mistakes
- **Likelihood**: Medium
- **Severity**: Medium
- **Impact**: Medium

Description

While the codebase generally follows good coding practices, there are some minor instances where stronger coding conventions could be applied to improve code readability, maintainability, and security.

Impact

Inconsistent coding conventions can diminish code readability and make it harder for third party integrations and future developers to understand the code.

Here are some specific instances:

1. The function `VestingFactory.sol → function createVestingFromNow(address _recipient, uint40 _duration, uint256 _amount, bool _isCancellable)` does not have input validation on the duration, allowing for someone to create an irrational duration period. This can lead to some unexpected results; for example, if duration is larger than the amount, the last section of the `getAccruedTokens()` function in `Vesting.sol` will always return 0.

2. The following if condition in `Spot.sol → createNewStf(STfSpot calldata _stf)` reverts if the token is not a base or deposit token - but uses the `NoBaseToken` error each time.

```
if (!isBaseToken[_stf.baseToken] || !isDepositToken[_stf.depositToken]) {
    revert NoBaseToken(_stf.baseToken);
}
```

3. The function `addTokenCapacity(address baseToken, address depositToken, uint96 capacity)` in `Spot.sol` requires the capacity to be nonzero. This means the capacity can never be unset once set.

4. The `Trade.sol → openSpot( ... )` emits an event `emit OpenSpot(spot.managerCurrentStf(msg.sender), ... )` at the end. This relays the manager of the `stf`;

however, this information is previously gathered earlier from `spot.getManagerCurrentStfInfo(msg.sender)`. This presents extra, unnecessary gas usage.

5. In `Trade.sol` → `_closeSpotSwap()`, before and after the call to `swap.swapUniversalRouter()`, the balance of the `depositToken` is measured and the difference is calculated. Inside `Swap.sol` → `swapUniversalRouter()` the same balance measurement is happening on the `receiver` parameter, which happens to be `depositToken`. This difference is returned to `closeSpotSwap()`. This ends up doing the balance difference calculation twice.

Recommendations

1. Make sure the duration has a sane maximum, preferably less than the amount.

2. Revert with a different reason when the given `depositToken` is not whitelisted.

3. Consider removing the requirement for capacity being nonzero, unless it is acceptable with no unset functionality.

4. Initialize the `OpenSpot` event with the existing variable (`_stf.manager`).

5. If the `swapUniversalRouter` always returns the same balance change, it can be removed from the `closeSpotSwap()` function. Otherwise, the measurement can be removed from Swap.

Remediation

STFX acknowledged and resolved the issue in 043fecb3, 5fcfa1a4, and 0389c664.

4 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment.

4.1 Centralization

The `Trade.closeSpotByAdmin` function allows admins to close any position in case of unexpected events, providing a fail-safe mechanism. However, this also introduces centralization risks that users should be aware of, as it grants a single point of control over the system. Therefore, it is recommended to implement additional measures to mitigate these risks, such as implementing a multi-signature requirement for admin access or limiting the frequency of usage.

Other centralization risks include the ability of the admin to pause and unpause the protocol at their discretion as well as their ability to cancel a vault.

We recommend that these centralization risks be clearly documented for users so that they are aware of the extent of the admin's control over the protocol. This can help users make informed decisions about their participation and investment in the protocol. Additionally, clear communication about the circumstances in which the admin may exercise these powers can help build trust and transparency with users.

4.2 Eighteen-decimal compliance

The decimal normalization factor in the `claim` function in `Spot.sol` is as follows:

```
amount / (10 ** (18 - tokenDecimals));
```

However, this approach only works for tokens with 18 or fewer decimals and will cause the function to revert with tokens that have more than 18 decimals. It is worth noting, however, that tokens with more than 18 decimals are currently rare in the ecosystem.

5 Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the smart 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.

> Please be advised that our threat model was conducted on the commit 945899af, which reflects the state of the codebase at a specific point in time. Thus, please note that the absence of a particular test in our report may not necessarily reflect the current state of the test suite.
>
> During the remediation phase STFX added test cases for all findings that could have one and covered the missing code branches with multiple tests, demonstrating their commitment to improving the code quality and robustness. This is commendable and enhances the overall reliability of the system.

5.1 Module: Spot.sol

Function: `addMinInvestmentAmount(address _token, uint96 _amount)`

Setter for the `minInvestmentAmount` mapping. This sets the lower bound for the amount of tokens that can be deposited into a fund. Can only be set by the owner.

Inputs

- `_token`
  - **Control**: Full.
  - **Constraints**: None, can even be the zero address.
  - **Impact**: Address to change lower bound for.
- `_amount`
  - **Control**: Full.
  - **Constraints**: 1e18 or greater.
  - **Impact**: The lower bound.

Branches and code coverage (including function calls)

Intended branches

- Set minimum investment amount.
  ☑ Test coverage
- Change an existing bound.
  ☐ Test coverage

Negative behavior

- No amount less than 1e18.
  ☐ Negative test

Function: `addPlugin(address _plugin, bool _isPlugin)`

Whitelists or blacklists a given address as a plugin, granting it access to open and close spots, plus transfer tokens to itself. Only callable by the contract owner.

Inputs

- `_plugin`
  - **Control**: Full.
  - **Constraints**: Nonzero address.
  - **Impact**: Address of the plugin to whitelist or blacklist.
- `_isPlugin`
  - **Control**: Full.
  - **Constraints**: Bool.
  - **Impact**: The whitelist status to set.

Branches and code coverage (including function calls)

Intended branches

- Whitelist address.
  ☑ Test coverage
- Blacklist address.
  ☐ Test coverage

Negative behavior

- Zero address.
  ☐ Negative test

Function: `addTokenCapacity(address baseToken, address depositToken, uint96 capacity)`

Sets the capacity for a given token trading pair during fundraising. This limit is per STF. Can only be set by the owner.

Care must be taken to avoid setting capacity for a non-whitelisted or invalid address, as it is not allowed to reset the capacity back to 0.

Inputs

- `baseToken`
  - **Control**: Full, even non-whitelisted.
  - **Constraints**: Nonzero.
  - **Impact**: Decides the address of the base token to add a trading capacity to.
- `depositToken`
  - **Control**: Full, even non-whitelisted.
  - **Constraints**: Nonzero.
  - **Impact**: Decides the address of the deposit token to add trading capacity to.
- `capacity`
  - **Control**: Full.
  - **Constraints**: Nonzero.
  - **Impact**: Sets the actual capacity value. Cannot be reset to less than 1.

Branches and code coverage (including function calls)

Intended branches

- Add capacity to multiple base/deposit pairs.
  - ☑ Test coverage
- Change the capacity once set.
  - ☐ Test coverage

Negative behavior

- Zero addresses.
  - ☐ Negative test
- Zero amount capacity.
  - ☐ Negative test

Function: `cancelVault(byte[32] salt)`

Cancels a vault. Can only be called by the admin or the manager of the STF. When an admin calls it, the cancellation is successful if there is nothing raised within the endTime or if the fundDeadline has passed. The manager can close it at any point if totalRaised is 0 or if fundDeadline has passed. A successful call to this function changes the status of the STF to one of the various cancellation reasons `CANCELLED_*`.

The function also frees the manager from being a manager, letting them open up a new STF in the future.

Inputs

- `salt`
  - **Control**: Full.
  - **Constraints**: Must be a valid salt such that it fetches a valid STF struct, and that STF must have status NOT_OPENED.
  - **Impact**: Identifier for an STF, of which it deletes the stf.manager from `isManagingFund` and zeros out its fundDeadline and endTime, plus setting the stf.status to a cancellation reason.

Branches and code coverage (including function calls)

Intended branches

- Admin cancels with zero raise.
  ☑ Test coverage
- Admin cancels with no fill.
  ☑ Test coverage
- Manager cancels with zero raise.
  ☑ Test coverage
- Manager cancels with generic reason.
  ☑ Test coverage

Negative behavior

- Called on opened STF.
  ☑ Negative test
- Called by nonmanager/nonadmin.
  ☑ Negative test
- Called by admin before fundDeadline.
  ☑ Negative test

Function: `changeBaseToken(address _baseToken, bool _isBaseToken)`

Updates the whitelist of eligible base tokens that can be used in a spot. Can only be set by the owner of the Spot contract, which is a multi-sig.

Due diligence must be used to make sure these tokens do not introduce vulnerabilities.

Inputs

- `_baseToken`
  - **Control**: Full.
  - **Constraints**: Nonzero.
  - **Impact**: Whitelists or blacklists a given token address for use in spots.
- `_isBaseToken`
  - **Control**: Full.
  - **Constraints**: Bool.
  - **Impact**: Decides whitelist status.

Branches and code coverage (including function calls)

Intended branches

- Whitelist token.
  ☑ Test coverage
- Blacklist token.
  ☐ Test coverage

Negative behavior

- Zero address.
  ☐ Negative test

Function: `changeDepositToken(address _depositToken, bool _isDepositToken)`

Updates the whitelist of eligible deposit tokens that can be used in a spot. Can only be set by the owner of the Spot contract, which is a multi-sig.

Due diligence must be used to make sure these tokens do not introduce vulnerabilities.

Inputs

- `_depositToken`
  - **Control**: Full.

  - **Constraints**: Nonzero.
  - **Impact**: Whitelists or blacklists a given token address for use in spots.
- `_isDepositToken`
  - **Control**: Full.
  - **Constraints**: Bool.
  - **Impact**: Decides whitelist status.

Branches and code coverage (including function calls)

Intended branches

- Whitelist token.
  ☑ Test coverage
- Blacklist token.
  ☐ Test coverage

Negative behavior

- Zero address.
  ☐ Negative test

Function: `claim(bytes32 _salt)`

Transfers the deposit to the investor depending on the investor's weightage to the `totalRaised` by the STF. Will revert if the investor did not invest in the STF during the `fundraisingPeriod`.

Inputs

- `_salt`
  - **Control**: User has full control of input parameter.
  - **Constraints**: The `_stf` corresponding to the salt must not be in a NOT_OPENED state.
  - **Impact**: Claims can only be made to STFs that are in an OPENED or DISTRIBUTED state.

Branches and code coverage (including function calls)

Intended branches

☑ An STF of open status pays out the correct amount to the investor.
☑ An STF of distributed status pays out the correct amount to the investor.

Negative behavior

☐ An STF of not opened status reverts.

Function call analysis

- `claim` → `getStfInfo(_salt)`
  - **What is controllable?** `_salt`.
  - **If return value controllable, how is it used and how can it go wrong?** Return value is used to get the corresponding STF info. It could go wrong if a user is able to claim for an STF that they have not invested in.
  - **What happens if it reverts, reenters, or does other unusual control flow?** User can reenter if ERC777 tokens are used; the protocol should do their due diligence to ensure that vetted tokens are used.
- `claim` → `claimableAmount(_salt, msg.sender)`
  - **What is controllable?** `_salt`.
  - **If return value controllable, how is it used and how can it go wrong?** Return value is used to get the amount that can be claimed by the user. It could go wrong if a user is able to claim for an STF that they have not invested in or the calculation is wrong.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Function can only revert if `totalRaised` is equal to zero because there would be a division by zero error. The control flow for determining how much a user should get paid out is complex, should be reviewed carefully, and ideally should be simplified.

Function: `closeFundraising()`

Stop the fundraising period by setting endTime equal to current time. This function can only be called by the manager of an STF and only before it has opened (status is NOT_OPENED). The function can not be called if endTime is in the past or if no amount has been raised.

Branches and code coverage (including function calls)

Intended branches

- Close fundraising.
  ☑ Test coverage

Negative behavior

- Called by nonmanager.
  ☐ Negative test
- Called on an STF that is opened.

    - ☐ Negative test
- Called with 0 raised amount.
    - ☐ Negative test
- Called past the endTime.
    - ☑ Negative test

Function: `closeSpot(uint96 remaining, byte[32] salt)`

Closes the Spot and removes the manager address as a current manager, letting them open up new Spots in the future. This can only be called by a plugin (e.g., the Trade contract). The status of the Spot is set to DISTRIBUTED, and the remaining amount is set for future claims.

Inputs

- `remaining`
    - **Control**: Full.
    - **Constraints**: None.
    - **Impact**: Sets the remaining amount (after fees) used to calculate the claimable amount.
- `salt`
    - **Control**: Full.
    - **Constraints**: None.
    - **Impact**: Decides the spot to close. Even when using the wrong salt, it can be opened again.

Function: `createNewStf(StfSpot _stf)`

Creates a new STF and stores the information about it. A salt is picked based on the hash of tokens, period, and block timestamp/number/chainid.

Inputs

- `_stf`
    - **Control**: Full, struct can be set by any caller. `fundDeadline` is hardcoded.
    - **Constraints**: Fundraising period must be at least 15 minutes and smaller than `maxFundraisingPeriod`. `baseToken` and `depositToken` must be whitelisted. Called cannot already be a manager.
    - **Impact**: Creates a new STF and occupies a mapping slot based on the salt that refers to the parameters. The caller becomes the manager of the STF.

Branches and code coverage (including function calls)

Intended branches

- Successfully creates a new STF.
  ☑ Test coverage

Negative behavior

- Existing manager tries to create a new STF.
  ☐ Negative test
- Fundraising period is out of bounds.
  ☐ Negative test
- The `baseToken` / `depositToken` pair is not whitelisted.
  ☐ Negative test

Function: `depositIntoFund(bytes32 salt, uint96 amount)`

Deposit a particular amount into an STF for the manager to open a position. The `fundraisingPeriod` has to end, and the `totalRaised` should not be more than `maxInvestmentPerStf`. The amount has to be more than `minInvestmentAmount`. Approve has to be called before this method for the investor to transfer USDC to this contract.

Inputs

- `salt`
  - **Control**: User has full control over this input.
  - **Constraints**: The STF corresponding to this salt must exist.
  - **Impact**: Salt corresponding to legitimate STF must be provided.
- `amount`
  - **Control**: User has full control over this input.
  - **Constraints**: The user must have at least this deposit amount in their wallet, and it must be scaled to 18 decimals.
  - **Impact**: User must provide an 18-decimal–scaled amount of the deposit token they wish to deposit.

Branches and code coverage (including function calls)

Intended branches

☑ Investor deposits into an STF.

Negative behavior

☐ STF does not exist.

- ☐ STF fundraising period has ended.
- ☐ STF has reached max investment.
- ☐ Less than minimum investment amount is attempted to be deposited.

Function call analysis

- `depositIntoFund` → `getStfInfo(salt)`
  - **What is controllable?** Salt.
  - **If return value controllable, how is it used and how can it go wrong?** Return value is controllable and is used to get the corresponding STF. It could go wrong if two STFs have the same salt; however, that appears in an improbable scenario.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Function cannot revert, reenter, or do other unusual control flow.
- `depositIntoFund` → `IERC20(_stf.depositToken).decimals()`
- **What is controllable?** Nothing, `_stf.depositToken` is unchangeable after STF creation.
  - **If return value controllable, how is it used and how can it go wrong?** Return value is not controllable and is used to get the number of decimals of the deposit token. It could go wrong if the deposit token has more than 18 decimals; however, it is an improbable scenario.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The control flow is dependent on the deposit token, and as such, we suggest the protocol to exercise caution when whitelisting tokens.
- `depositIntoFund` → `IERC20(_stf.depositToken).balanceOf(msg.sender)`
- **What is controllable?** `msg.sender` is somewhat controllable, as it is the address of the investor.
  - **If return value controllable, how is it used and how can it go wrong?** Return value is not controllable and is used to get the balance of the deposit token of the investor. It could go wrong if the protocol has whitelisted a malicious token.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The control flow is dependent on the deposit token, and as such, we suggest the protocol to exercise caution when whitelisting tokens.
  - `depositIntoFund` → `IERC20(_stf.depositToken).transferFrom(msg.sender, address(this), depositAmount)`
  - **What is controllable?** The deposit amount is controllable, as it is the amount the investor wants to deposit, and the `msg.sender` is controllable, as it is the address of the investor.
  - **If return value controllable, how is it used and how can it go wrong?** No

return value in this function call. It could go wrong if the protocol has whitelisted a malicious token. Though by EIP20 standards, the return value should be true if the transfer is successful, and thus the protocol should be checking for that.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The control flow is dependent on the deposit token, and as such, we suggest the protocol to exercise caution when whitelisting tokens.

Function: `openSpot(uint96 amount, uint96 received, byte[32] salt)`

Opens a Spot by setting the status to OPENED. Only callable by plugins and will be used by the Trade contract after actually opening up the position.

Inputs

- `amount`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Affects `totalAmountUsed`, used when calculating claims.
- `received`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Sets `totalReceived`, but it is not used in any contracts in the scope.
- `salt`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The salt that identifies the STF to open.

5.2 Module: Swap.sol

Function: `swapUniversalRouter(address tokenIn, address tokenOut, uint160 amountIn, byte[] commands, byte[][] inputs, uint256 deadline, address receiver)`

Transfers `amountIn` of `tokenIn` tokens from the Spot to Swap and approves them for trading. By using the swap commands in `commands` to decide parameters like reverting on failure, and the ABI-encoded inputs in `inputs`, multiple trades will be executed within the `deadline` block timestamp - or everything reverts. The amount of `tokenOut` is measured before and after the trades, and the difference is returned.

This function can only be called by the address in `trade`, and that happens whenever

a Spot is opened or closed.

Inputs

- `tokenIn`
  - **Control**: Full.
  - **Constraints**: Must be a contract with IERC20 ABI.
  - **Impact**: The type of token to transfer from Spot before executing the trades.
- `tokenOut`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The type of token to return the balance change for.
- `amountIn`
  - **Control**: Full.
  - **Constraints**: Cannot be more tokens than `spot` has.
  - **Impact**: The amount of tokens to transfer and approve.
- `commands`
  - **Control**: Full.
  - **Constraints**: Special bitfield format (Uniswap).
  - **Impact**: For each input, decides the command and if a command is allowed to revert.
- `inputs`
  - **Control**: Full.
  - **Constraints**: ABI encoded and supported further up the chain (Uniswap).
  - **Impact**: ABI-encoded inputs for address, available trade amount, minimum amount to trade, Uniswap path, and if funds come from caller (Permit2) or if they are already in the router.
- `deadline`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: A block timestamp for when the execution should fail if not completed. If 0, an execute function without the deadline parameter is called.
- `receiver`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The receiver address to use when calculating the balance change.

Branches and code coverage (including function calls)

Intended branches

- Called with deadline.
  ☑ Test coverage
- Called with no deadline.
  ☑ Test coverage

Negative behavior

- Called from nontrader.
  ☐ Negative test
- Called with invalid or bad commands/inputs.
  ☐ Negative test

Function call analysis

- `swapUniversalRouter` → `spot.transferToken(tokenIn, amountIn)`
- **What is controllable?** All.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If spot owns less than `amountIn` tokens of type `tokenIn`, this reverts.
- `swapUniversalRouter` → `IERC20(tokenIn).approve(address(permit2), amountIn)`
- **What is controllable?** `tokenIn`, `amountIn`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Approved amount can be changed without reducing it to 0 in between. This allows for the approved spender to spend both allowances with unfortunate transaction ordering.
- `swapUniversalRouter` → `universalRouter.execute(commands, inputs, [deadline])`
  - **What is controllable?** All.
  - **If return value controllable, how is it used and how can it go wrong?** With bad inputs, can be made to trade less than the intended amount and lock up funds. Caller must be sure to sync `amountIn` and `inputs`. Too long deadline could lead to bad trades.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts if deadline has expired, an incorrect number of inputs are provided or the execution fails.

5.3 Module: Trade.sol

Function: `changeStfxSpot(address _spot)`

Setter for the internal Spot contract. Can only be changed by the current Spot owner. Allows upgrading the Spot implementation. Great care must be taken to ensure that the new spot has the correct ABI and owner or this can never be called again.

Inputs

- `_spot`
  - **Control**: Full.
  - **Constraints**: Must be nonzero.
  - **Impact**: The Spot implementation to use.

Branches and code coverage (including function calls)

Intended branches

- Can be called existing spot.
  ☑ Test coverage

Negative behavior

- Disallows zero address.
  ☐ Negative test
- Disallows other addresses that are not the current spot.
  ☐ Negative test

Function call analysis

- `changeStfxSpot` → `spot.owner()`
  - **What is controllable?** Nothing.
  - **If return value controllable, how is it used and how can it go wrong?** If the current spot is a bad contract without an owner property, future spot changes will fail.
  - **What happens if it reverts, reenters, or does other unusual control flow?** In a state where it reverts, the implementation can no longer be upgraded.

Function: `changeStfxSwap(address _swap)`

Setter for the internal Swap contract. Can only be changed by the current spot owner.

Inputs

- `_swap`

- **Control**: Full.
- **Constraints**: Must be a nonzero address.
- **Impact**: Decides which contract implementation is used for swapping tokens.

Branches and code coverage (including function calls)

Intended branches

- Can be set by spot owner.
  - ☐ Test coverage

Negative behavior

- Cannot be set to zero address.
  - ☐ Negative test
- Cannot be set by non-spot-owner.
  - ☐ Negative test

Function: `closeSpot(byte[] commands, byte[][] inputs, uint256 deadline)`

Closes a spot position. This triggers the universal router to swap from `basetokens` to `deposittokens`. Distributes the profit minus fees if there is a profit, otherwise just the remaining amount. Also sets status to DISTRIBUTED. Can only be called by the current manager of the STF and only when the STF has status OPENED.

Inputs

- `commands`
  - **Control**: Full.
  - **Constraints**: Concatenated, 1-byte commands sent to the UniversalRouter (e.g., Uniswap).
  - **Impact**: Decides, for example, if transactions should be able to revert and identifies the command to be carried out.
- `inputs`
  - **Control**: Full.
  - **Constraints**: Array of (ABI-encoded) inputs to each command.
  - **Impact**: Decides recipient, amount of tokens to trade, min. amount, trade path, and so forth. See Uniswap details.
- `deadline`
  - **Control**: Full.
  - **Constraints**: None.

- **Impact**: Maximum timestamp for the tx. Will revert after this time.

Branches and code coverage (including function calls)

Intended branches

- Successfully closes the spot.
  ☑ Test coverage

Negative behavior

- Called by wrong address.
  ☑ Negative test

Function call analysis

- `closeSpot` → `getManagerCurrentStfInfo(msg.sender)`
  - **What is controllable?** Address is `msg.sender` only.
  - **If return value controllable, how is it used and how can it go wrong?** Returns the salt for the STF associated with the address.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If address is unknown, returns the zero struct.
- `closeSpot` → `_closeSpotSwap(_stf, commands, inputs, deadline)` → `swap.swapUniversalRouter(_stf, commands, inputs, deadline)`
  - **What is controllable?** commands, inputs, and deadline are fully controllable. `_stf` is a struct fetched from the spot, based on `msg.sender`.
  - **If return value controllable, how is it used and how can it go wrong?** If deadline expires, everything reverts. Commands/inputs are consumed further up the chain (out of scope).
  - **What happens if it reverts, reenters, or does other unusual control flow?** Depending on the contents of `commands` and the deadline, this is expected to revert if the transfers fail, or fail to execute before the deadline.
- `closeSpot` → `_distribute(_stf, amount)` → `spot.closeSpot(remaining, salt)`
  - **What is controllable?** Amount is somewhat controllable by funding the STF more.
  - **If return value controllable, how is it used and how can it go wrong?** Depending on the ERC20 token used, the transfer functions can fail with or without a revert and with nothing or partial amounts transferred. There is also possibility for reentrancy issues with the wrong token.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If made to revert, the STF status would not change, leading to investor claims not working as expected.

Function: `closeSpotByAdmin(byte[] commands, byte[][] inputs, byte[32] salt)`

Closes a spot position. This is done automatically by an EOA keeper bot with the admin role. When an admin closes a spot position, there are no checks for the status before closing it. The salt is also specified as an input, instead of being fetched from a mapping based on `msg.sender`, and the deadline is hardcoded to 0. A deadline as 0 makes the Swap contract call the execute function that does not use deadline.

Otherwise, this function has the same threat model as `closeSpot(bytes calldata commands, bytes[] calldata inputs, uint256 deadline)`. The threat model below will only explain the differences.

Inputs

- `commands`
- `inputs`
- `salt`
  - **Control**: Full.
  - **Constraints**: Must be a valid salt for an STF or it will become an empty struct and ERC20 functions on it will fail.
  - **Impact**: Decides which STF to close and distribute.

Branches and code coverage (including function calls)

Intended branches

- Closed by an admin.
  ☑ Test coverage

Negative behavior

- Attempted close by nonadmin.
  ☐ Negative test

Function: `openSpot(uint96 amount, bytes calldata commands, bytes[] calldata inputs, uint256 deadline)`

Opens a spot position using the universal router (swaps from `depositToken` to `baseToken`) and can only be called by the manager of the STF.

Inputs

- `amount`

  - **Control**: User has full control over this input, given they are manager of the STF.
  - **Constraints**: The contract must hold the amount of `depositToken` specified by the user.
  - **Impact**: User can only trade as much as the contract holds.
- `commands`
  - **Control**: User has full control over this input, given they are manager of the STF.
  - **Constraints**: Acceptable by Uniswap V3 as this param is passed directly to the router.
  - **Impact**: User has full control over the swap as long as it is acceptable by Uniswap V3 and they are a a manager of the STF.
- `inputs`
  - **Control**: User has full control over this input, given they are manager of the STF.
  - **Constraints**: Acceptable by Uniswap V3 as this param is passed directly to the router.
  - **Impact**: User has full control over the swap as long as it is acceptable by Uniswap V3 and they are a manager of the STF.
- `deadline`
  - **Control**: User has full control over this input, given they are manager of the STF.
  - **Constraints**: Acceptable by Uniswap V3 as this param is passed directly to the router.
  - **Impact**: User has full control over the swap as long as it is acceptable by Uniswap V3 and they are a manager of the STF.

Branches and code coverage (including function calls)

Intended branches

- ☑ User is a manager of the STF and performs a swap.

Negative behavior

- ☑ User is not a manager of the STF.
- ☑ User does not have enough `depositToken` to open the spot position.
- ☐ Amount is greater than the `totalRaised`.
- ☑ Spot end timestamp is greater than the current timestamp.
- ☐ `baseToken` is not linked to this contract.

Function call analysis

- `openSpot` → `spot.getManagerCurrentStfInfo(msg.sender)`
- **What is controllable?** `msg.sender` to the extent that it is the manager of the STF.
  - **If return value controllable, how is it used and how can it go wrong?** Return value is the corresponding STF and could go wrong if the wrong STF is returned; however, that does not appear the case.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Cannot revert, reenter, or do other unusual control flow.
- `openSpot` → `_openSpotCheck(_stf, amount)`
- **What is controllable?** The STF to the extent that it is the manager of the STF.
  - **If return value controllable, how is it used and how can it go wrong?** Return value is the swap amount scaled to the correct decimal units and could go wrong if the wrong amount is returned; however, that does not appear the case.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Cannot revert, reenter, or do other unusual control flow.
- `openSpot` → `_openSpotUpdate(_stf, amount, swapAmount, commands, inputs, deadline)`
  - **What is controllable?** The STF to the extent that it is the manager of the STF.
  - **If return value controllable, how is it used and how can it go wrong?** Return value is the amount received after the swap and could go wrong if the wrong amount is returned. One thing that is interesting is that this number is not scaled to 1e18 before saving it for the `stf.totalReceived` variable. This is unlike `stf.totalAmount`, which is scaled to 1e18 before saving it. This is unlike `depositIntoFund`, where the `stf.totalAmount` is scaled to 1e18 first. Having a consistent measure of scaling for the STF state variables would be good; however, there appears to be nothing problematic about the current setup.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Cannot revert or do any unusual control flow. Reentrancy may be possible if the swap is a flash swap; however, this is not the case for the current implementation or an ERC777 token is traded.

5.4 Module: VestingFactory.sol

Function: `createVestingFromNow(address _recipient, uint40 _duration, uint256 _amount, bool _isCancellable)`

Create a new vesting to a recipient starting from the current `block.timestamp` with a set duration and amount of tokens. Can only be called by the treasury.

Inputs

- `_recipient`
  - **Control**: Only the treasury can supply this input.
  - **Constraints**: Any recipient address.
  - **Impact**: The treasury can create vesting for any recipient.
- `_duration`
  - **Control**: Only the treasury can supply this input.
  - **Constraints**: No constraints.
  - **Impact**: The treasury can create vesting with any duration. Should have an upper and lower bound.
- `_amount`
  - **Control**: Only the treasury can supply this input.
  - **Constraints**: Must be greater than 1 and less than the balance of the factory.
  - **Impact**: Only a valid amount can be vested.
- `_isCancellable`
  - **Control**: Only the treasury can supply this input.
  - **Constraints**: No constraints.
  - **Impact**: The treasury can create a vesting that is cancellable or not.

Branches and code coverage (including function calls)

Intended branches

- ☑ Treasury is able to create a new vesting to a recipient starting from now with a specified duration and amount of tokens.

Negative behavior

- ☑ Recipient address is zero.
- ☑ Amount is zero.
- ☐ Amount is more than the balance of the factory.
- ☑ Vesting already exists for the recipient.

Function call analysis

- `createVestingFromNow → IVesting(vestingAddress).initialise(_recipient, uint40(block.timestamp), _duration, _amount, _isCancellable)`
- **What is controllable?** The treasury can supply the recipient address, duration, amount, and `isCancellable`.
  - **If return value controllable, how is it used and how can it go wrong?** No return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Can only revert if the `vestingAddress` is not unique, which should not happen as it is a clone of the vesting contract. Reentrancy is not a concern, and no unusual control flow.
- `createVestingFromNow → token.transfer(vestingAddress, _amount)`
  - **What is controllable?** The amount of tokens to be transferred is controllable by the treasury.
  - **If return value controllable, how is it used and how can it go wrong?** No return value, but ERC20 transfer can fail without reverting, so it should be checked.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Control flow is dependent on the ERC20 transfer function, thus the protocol should do their due diligence to ensure that the transfer function of the specific ERC20 is safe.

Function: `createVestingStartingFrom(address _recipient, uint40 _start, uint40 _duration, uint256 _amount, bool _isCancellable)`

Create a new vesting to a recipient starting from a custom `_start` time, with a set duration and amount of tokens. Can only be called by the treasury.

Inputs

- `_recipient`
  - **Control**: Only the treasury can supply this input.
  - **Constraints**: Any recipient address.
  - **Impact**: The treasury can create vesting for any recipient.
- `_start`
  - **Control**: Only the treasury can supply this input.
  - **Constraints**: Cannot be earlier than current block timestamp.
  - **Impact**: Custom starting time for the vesting.
- `_duration`
  - **Control**: Only the treasury can supply this input.

  - **Constraints**: None.
  - **Impact**: The treasury can create vesting with any duration. Should have an upper and lower bound.
- `_amount`
  - **Control**: Only the treasury can supply this input.
  - **Constraints**: Must be greater than 1 and less than the balance of the factory.
  - **Impact**: Only a valid amount can be vested.
- `_isCancellable`
  - **Control**: Only the treasury can supply this input.
  - **Constraints**: No constraints.
  - **Impact**: The treasury can create a vesting that is cancellable or not.

Branches and code coverage (including function calls)

Intended branches

- Treasury is able to create a new vesting to a recipient starting from `_start`, with a specified duration and amount of tokens.
  ☑ Test coverage

Negative behaviur

- Recipient address is zero.
  ☐ Negative test
- Start time in the past.
  ☐ Negative test
- Amount is zero.
  ☐ Negative test
- Amount is more than the balance of the factory.
  ☐ Negative test
- Vesting already exists for the recipient.
  ☐ Negative test

Function call analysis

- `createVestingStartingFrom` → `IVesting(vestingAddress).initialise(_recipient, _start, _duration, _amount, _isCancellable)`
- **What is controllable?** The treasury can supply all the parameters.
- **If return value controllable, how is it used and how can it go wrong?** No return value.
- **What happens if it reverts, reenters, or does other unusual control flow?** Can only revert if the `vestingAddress` is not unique, which should not happen as it

is a clone of the vesting contract. Reentrancy is not a concern, and no unusual control flow.

- `createVestingStartingFrom` → `token.transfer(vestingAddress, _amount)`
- **What is controllable?** The amount of tokens to be transferred is controllable by the treasury.
- **If return value controllable, how is it used and how can it go wrong?** No return value, but ERC20 transfer can fail without reverting, so it should be checked.
- **What happens if it reverts, reenters, or does other unusual control flow?** Control flow is dependent on the ERC20 transfer function; thus, the protocol should do their due diligence to ensure that the transfer function of the specific ERC20 is safe.

5.5 Module: Vesting.sol

Function: `claim()`

A function allowing the recipient to claim the vested tokens.

Branches and code coverage (including function calls)

Intended branches

- ☑ Recipient can claim tokens.

Negative behavior

- ☑ Recipient cannot claim tokens if they have already claimed.
- ☑ Recipient cannot claim tokens if the vest has been cancelled.

Function call analysis

- `claim` → `factory.token().transfer(recipient, claimAmount)`
  - **What is controllable?** Nothing is controllable.
  - **If return value controllable, how is it used and how can it go wrong?** N/A, no return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Token transfer can fail silently if the ERC20 corresponding to the factory token is implemented incorrectly. Check-effects-interactions is implemented correctly so reentrancy is not an issue.

6 Audit Results

At the time of our audit, the code was not deployed to mainnet Ethereum.

During our audit, we discovered five findings. Of the five findings, two were of high impact and three were of medium impact. STFX acknowledged all findings and implemented fixes.

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 audit version of 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.

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.