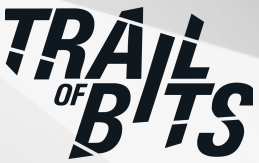

Zebec Payment Stream Contract

Security Assessment

September 8, 2022

Prepared for:
Ajay Gautam
Zebec

Prepared by: **Anders Helsing and Troy Sargent**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 80+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Executive Summary

Engagement Overview

Zebec engaged Trail of Bits to review the security of its Payment Stream contract. From August 22 to September 2, 2022, a team of two consultants conducted a security review of the client-provided source code, with four person-weeks of effort. Details of the project's timeline, test targets, and coverage are provided in subsequent sections of this report.

Project Scope

Our testing efforts were focused on the identification of flaws that could result in a compromise of confidentiality, integrity, or availability of the target system. We conducted this audit with full knowledge of the system, including access to the source code and documentation. We performed static and dynamic testing of the target system and its codebase, using both automated and manual processes.

Summary of Findings

The audit uncovered flaws that could impact system confidentiality, integrity, or availability. A summary of the findings is provided below.

EXPOSURE ANALYSIS

Severity	*Count*
Medium	3
Low	1
Informational	4

CATEGORY BREAKDOWN

Category	*Count*
Data Validation	3
Patching	1
Undefined Behavior	4

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Mary O'Brien, Project Manager
mary.obrien@trailofbits.com

The following engineers were associated with this project:

Anders Helsing, Consultant
anders.helsing@trailofbits.com

Troy Sargent, Consultant
troy.sargent@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
August 17, 2022	Pre-project kickoff call
August 29, 2022	Status update meeting #1
September 2, 2022	Delivery of report draft
September 2, 2022	Report readout meeting
September 8, 2022	Delivery of final report

Project Goals

The engagement was scoped to provide a security assessment of the Zebec Payment Stream contract. Specifically, we sought to answer the following non-exhaustive list of questions:

- Can a malicious actor withdraw funds from the Zebec vault account in a manner other than intended?
- Can funds become frozen?
- Can math operations within the contract instructions result in overflow or underflow conditions?
- Is it possible to bypass the checks on accounts used by instructions?
- Can instructions use the wrong type of accounts?
- Can one type of token be deposited while another is withdrawn?

Project Targets

The engagement involved a review and testing of the following target.

Zebec Payment Stream Solana/Anchor contract

Repository	https://github.com/Zebec-protocol/zebec-anchor
Version	a313108b3bc22fa82667f7462bcb1e204c63c67f
Type	Rust
Platform	Solana/Anchor

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches include the following:

- **Static analysis with `cargo-audit`, `cargo-outdated`, and Clippy**: We ran `cargo-audit` and `cargo-outdated` over the `Cargo.lock` file. We ran Clippy over all of the Rust source files. We reviewed the results of each run.
- **Fuzzing**: We used `test-fuzz` to verify properties of the allowed amount calculation.
- **Manual review**: We manually reviewed the Payment Stream contract, with a focus on answering the questions listed under Project Goals.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- We did not review the payment stream contract operation in a multisig context.
- We did not extend the review into the Anchor framework or the SPL Token program.

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in house, to perform automated testing of source code and compiled software.

Test Harness Configuration

We used the following tools in the automated testing phase of this project:

Tool	Description	Result
`cargo-audit`	A Cargo subcommand for auditing `Cargo.lock` files for crates with security vulnerabilities reported to the RustSec Advisory Database	No significant results
`cargo-outdated`	A Cargo subcommand for displaying outdated Rust dependencies	No significant results
`rust-clippy`	A collection of lints to catch common Rust mistakes and to improve Rust code	TOB-ZEB-2, TOB-ZEB-5, TOB-ZEB-8
`test-fuzz`	A collection of Rust macros and a Cargo subcommand that automate certain fuzzing-related tasks	Appendix D

Areas of Focus

Our automated testing and verification work focused on the following system properties:

- Vulnerable or outdated dependencies
- Common Rust mistakes
- The impacts of potential rounding errors

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	Unchecked arithmetic is used in several places without justification (TOB-ZEB-3, TOB-ZEB-5, and TOB-ZEB-8). The use of unchecked arithmetic allows under- or overflows, which should be addressed.	**Weak**
Authentication / Access Controls	The Anchor framework's mechanisms for expressing account roles, constraints, and authentication are consistently used.	**Strong**
Complexity Management	The project would benefit from additional documentation to manage the codebase's complexity, such as state transition diagrams. Additionally, we identified several cases of code duplication. The code is not formatted according to style guidelines, making it harder to read.	**Moderate**
Cryptography and Key Management	We did not uncover issues related to cryptography and key management.	**Satisfactory**
Decentralization	We did not consider decentralization or upgradeability in our review.	**Further Investigation Required**
Documentation	The contract's documentation is inadequate. Additionally, the documentation on `docs.zebec.io` appears to have missing pages and link target errors. Furthermore, it would be beneficial to outline how users are expected to interact with the contracts and then identify properties	**Weak**

	that should not be broken for the Zebec program.	
Testing and Verification	The tests verifies that the normal sequence of events is successful, but does not check edge cases. The project would benefit from property and fuzz testing.	**Moderate**

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	No check for available funds when creating or updating stream	Data Validation	Medium
2	Divide-before-multiply and truncating cast causes rounding errors	Data Validation	Low
3	Missing bounds check on fee percentage	Data Validation	Informational
4	The data_account account may not be closed on full payment	Undefined Behavior	Informational
5	The withdraw_state.amount property could overflow or underflow	Undefined Behavior	Medium
6	Insecure method used to close accounts	Undefined Behavior	Medium
7	Code duplication across instructions and modules	Patching	Informational
8	Inconsistent use of checked math	Undefined Behavior	Informational

Detailed Findings

1. No check for available funds when creating or updating a stream	
Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ZEB-1
Target: `programs/zebec/src/processor/native_stream/mod.rs`, `programs/zebec/src/processor/token_stream/mod.rs`	

Description
When a user creates or updates a payment stream, there is no check that the vault tied to the user account holds enough funds for the stream. This could create a situation where the recipient of a payment stream is unable to withdraw funds. The absence of this check also exacerbates the overflow problem in TOB-ZEB-5, because it allows invoking the stream's create and update instructions with very large values supplied for the `amount` parameter.

Exploit Scenario
Mallory creates a payment stream with Alice as a recipient, but does not transfer funds to the vault. When Alice tries to withdraw funds from the stream, the transaction fails.

Recommendations
Short term, when creating or updating a stream, check that the vault contains enough funds to cover all streams tied to the creating account. This will ensure that the recipient is able to withdraw the full amount of the payment stream.

Long term, ensure that all operations that increase the escrowed amount are covered with sufficient funds in the vault.

2. Divide-before-multiply and truncating cast causes rounding errors

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ZEB-2
Target: `programs/zebec/src/processor/native_stream/mod.rs`, `programs/zebec/src/processor/token_stream/mod.rs`	

Description

In several places, the Zebec program calculates how much a given user can withdraw as a proportion of the duration of a stream to how much time has elapsed since its start. This calculation loses precision when it casts an unsigned, 64-bit integer to a 64-bit float that is only 52 bits wide. As a result, the `allowed_amt` for users can be incorrect, affecting whether a user can complete a withdrawal.

```
pub fn allowed_amt(&self, now: u64) -> u64 {
    (
    ((now - self.start_time) as f64) / ((self.end_time - self.start_time) as f64) *
self.amount as f64
    ) as u64
}
```

Figure 2.1: The `allowed_amt` function

```
((((now - start_time) as u128) * amount as u128) / (end_time - start_time) as u128)
as u64
```

Figure 2.2: The recommended calculation

Exploit Scenario

A user withdraws their funds and does not receive the expected amount. For example, if a user attempts to withdraw 1 unit of time into a stream with an amount of 214, start time of 0, and end time of 214, the allowed amount calculated will be zero. The user will not be able to draw the correct amount of 1.

Recommendations

Short term, instead of casting to a floating point representation, cast the operands to 128-bit unsigned integers and multiply before dividing, as shown in Figure 2.2. This will minimize loss of precision.

Long term, run `cargo clippy --workspace -- -W clippy::pedantic` and use property and fuzz testing to analyze arithmetic for rounding errors. This will help to identify similar vulnerabilities.

3. Missing bounds check on fee percentage

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ZEB-3
Target: `programs/zebec/src/processor/create_fee_account/mod.rs`	

Description

The `process_create_fee_account` instruction is used to initialize the accounts used for fee collection. However, there is no check that the `amount` parameter does not exceed the divisor used (10,000). If the accounts are created with a fee percentage larger than the value of the divisor, the calculation of commission in the contract will exceed the `allowed_amt` variable. This would also cause the `receiver_amount` variable to underflow.

```
let comission: u64 = ctx.accounts.vault_data.fee_percentage*allowed_amt/10000;
let receiver_amount:u64=allowed_amt-comission;
```

Figure 3.1: The `commission` and `receiver_amount` calculation

Exploit Scenario

Mallory creates fee accounts with a large value for `fee_percentage`, making all attempts to withdraw from the stream fail.

Recommendations

Short term, add a check to make sure that the value for the `fee_percentage` parameter falls within the expected range. This will ensure that the calculations of `commission` and `receiver_amount` work correctly.

Long term, assign boundaries for expected outcomes of calculations, and use property testing to make sure the assumptions hold. This will ensure unexpected results are caught.

4. The data_account account may not be closed on full payment

Severity: **Informational**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-ZEB-4
Target: `programs/zebec/src/processor/native_stream/mod.rs`, `programs/zebec/src/processor/token_stream/mod.rs`	

Description

The `process_withdraw_stream` instruction includes a check that the withdrawn amount is equal to the total amount of the stream. If it is, the remaining lamports of the `data_account` account are transferred to the `sender` account, effectively deleting the `data_account` account at the end of the transaction. However, the check does not account for a nonzero `paused_amt` amount, which would prevent the `data_account` account from being deleted. If the end time of the stream has passed, it will not be possible to cancel the stream, leaving the `data_account` account in limbo.

Note that the same issue also affects the token stream.

```
//allowed amount is subtracted from paused amount
allowed_amt =
allowed_amt.checked_sub(data_account.paused_amt).ok_or(ErrorCode::PausedAmountExceeds)?;
...
data_account.withdrawn=
data_account.withdrawn.checked_add(allowed_amt).ok_or(ErrorCode::NumericalOverflow)?;

// This will never be true of paused_amt > 0
if data_account.withdrawn == data_account.amount {
create_transfer_signed(data_account.to_account_info(),ctx.accounts.sender.to_account_info(),
data_account.to_account_info().lamports())?;
}
```

Figure 4.1: programs/zebec/src/processor/native_stream/mod.rs#L83-L101

Exploit Scenario

Alice creates a payment stream, which is then paused long enough to ensure that the `data_account.paused_amt` property is not zero. When the receiver of the payment stream has withdrawn the full amount to be collected, the `data_account` account is not deleted.

Recommendations

Short term, consider the paused amount when deciding if the `data_account` account should be deleted. This will ensure that the account is deleted upon full payment of a stream.

Long term, update the contract documentation with information on how and when accounts are to be deleted. This will guide contract developers implementing the code for deleting accounts.

5. The withdraw_state.amount property could overflow or underflow

Severity: **Medium**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-ZEB-5
Target: `programs/zebec/src/processor/native_stream/mod.rs`, `programs/zebec/src/processor/token_stream/mod.rs`	

Description

Since the stream contract does not require the `zebec_vault` account to have sufficient funds for all streams, it is possible to execute the `process_native_stream` and `process_update_native_stream` instructions with an `amount` parameter value large enough for the `withdraw_state.amount` property to overflow.

```
pub fn process_native_stream(
    ctx: Context<Initialize>,
    start_time: u64,
    end_time: u64,
    amount: u64,
    can_cancel:bool,
    can_update:bool,
) -> Result<()> {
    let data_account = &mut ctx.accounts.data_account;
    let withdraw_state = &mut ctx.accounts.withdraw_data;
    check_overflow(start_time, end_time)?;
    data_account.start_time = start_time;
    data_account.end_time = end_time;
    data_account.paused = 0;
    data_account.amount = amount;
    data_account.withdraw_limit = 0;
    data_account.sender = ctx.accounts.sender.key();
    data_account.receiver = ctx.accounts.receiver.key();
    data_account.fee_owner=ctx.accounts.fee_owner.key();
    data_account.paused_at=0;
    data_account.paused_amt=0;
    data_account.can_cancel=can_cancel;
    data_account.can_update=can_update;
    withdraw_state.amount+=amount;
```

Figure 6.1: programs/zebec/src/processor/native_stream/mod.rs#L11-L34

Furthermore, because both the `process_cancel_stream` and `process_update_native_stream` instructions reduce the `amount` property by the total amount of a stream, executing both instructions for a given stream would underflow the value (please refer to TOB-ZEB-6 for a detailed explanation of how this can occur). The

same underflow situation would also occur if two `process_cancel_stream` instructions were executed for a given stream. This latter example would allow multiple payments to the receiver, because the `data_account.withdrawn` property is not updated in the `process_cancel_stream` instruction. Note that the same issue also affects the token stream.

```
pub fn process_cancel_stream(
    ctx: Context<Cancel>,
) -> Result<()> {
    let data_account = &mut ctx.accounts.data_account;
    let withdraw_state = &mut ctx.accounts.withdraw_data;
    let zebec_vault =&mut  ctx.accounts.zebec_vault;
    let now = Clock::get()?.unix_timestamp as u64;
// * snip * //
    withdraw_state.amount-=data_account.amount-data_account.withdrawn;
```

Figure 6.1: programs/zebec/src/processor/native_stream/mod.rs#L132-L175

Exploit Scenario

Mallory creates a regular payment stream A, then proceeds to create another payment stream B, which is immediately canceled, after which stream B is finally updated. When creating the payment stream B, the `amount` parameter can be used to control the resulting value of the `withdraw_state.amount` property. For example, by setting the `withdraw_state.amount` property to zero, this would allow Mallory to transfer all the lamports out of the `zebec_vault` account, regardless of any active streams belonging to Mallory.

Recommendations

Short term, use checked or saturating math operations instead of integer arithmetic. This will ensure that any under- or overflow of the `withdraw_state.amount` property is detected.

Long term, refer to TOB-ZEB-8 for guidance on how to use Clippy to detect integer arithmetic.

6. Insecure method used to close accounts

Severity: **Medium**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-ZEB-6
Target: `programs/zebec/src/processor/native_stream/mod.rs`, `programs/zebec/src/processor/token_stream/mod.rs`	

Description

In several places throughout the code, the contract transfers all the remaining lamports out of an account in order to delete it. Although accounts without lamports are deleted by the Solana runtime, this does not occur until all instructions in a transaction have been executed.

For example, the `process_cancel_stream` instruction closes the `data_account` account by transferring the lamports to the `sender` account. However, because the account is not deleted until the transaction is completed, any succeeding instructions in the transaction that reference the `data_account` account will perceive the account as valid. Note that the same issue also affects the token stream.

```
//closing the data account to end the stream
create_transfer_signed(data_account.to_account_info(),ctx.accounts.sender.to_account
_info(), data_account.to_account_info().lamports())?;
```

Figure 5.1: programs/zebec/src/processor/native_stream/mod.rs#L176-L177

Exploit Scenario

Mallory creates a payment stream, then issues a transaction containing the instructions `process_cancel_stream` and `process_update_native_stream`. Because the stream's `data_account` account is not deleted until all instructions in the transaction have been executed, the `process_update_native_stream` instruction succeeds even though the stream has been canceled. Mallory could exploit this issue to effectively control the value of the `withdraw_data` account, which in turn could allow Mallory to transfer all lamports out of the `zebec_vault` account even if streams are active.

Recommendations

Short term, delete accounts using the `#[account(close = <target_account>)]` functionality provided by the Anchor framework. This will ensure that the account discriminator is set to `CLOSED_ACCOUNT_DISCRIMINATOR`, which prevents account revival attacks.

Long term, update the contract documentation with information about how and when accounts are deleted. This will guide contract developers implementing the code for deleting accounts.

7. Code duplication across instructions and modules

Severity: **Informational**	Difficulty: **Not Applicable**
Type: Patching	Finding ID: TOB-ZEB-7
Target: Various locations in the payment stream contract	

Description

Code duplication exists in several places in the payment stream contract implementation. For example, the `process_native_transfer` instruction is nearly identical to the `process_native_withdrawal` instruction, and much of the code for the token payment streams is duplicated for the native payment streams.

```
let withdraw_state = &mut
ctx.accounts.withdraw_data;
let zebec_vault =&mut
ctx.accounts.zebec_vault;

if amount > zebec_vault.lamports()
{
return
Err(ErrorCode::InsufficientFunds.into());
}
// if no any stream is started allow the
instant transfer w/o further checks
if withdraw_state.amount ==0
{
create_transfer_signed(zebec_vault.to_acc
ount_info(),ctx.accounts.receiver.to_acco
unt_info(),amount)?;
}
else
{
//Check remaining amount after transfer
let allowed_amt = zebec_vault.lamports()
- amount;
//if remaining amount is lesser then the
required amount for stream stop making
withdrawal
if allowed_amt < withdraw_state.amount {
    return
Err(ErrorCode::StreamedAmt.into());
}
create_transfer_signed(zebec_vault.to_acc
ount_info(),ctx.accounts.receiver.to_acco
unt_info(),amount)?;
```

```
let withdraw_state = &mut
ctx.accounts.withdraw_data;
let zebec_vault =&mut
ctx.accounts.zebec_vault;

if amount > zebec_vault.lamports()
{
return
Err(ErrorCode::InsufficientFunds.into());
}
// if no any stream is started allow the
withdrawal w/o further checks
if withdraw_state.amount ==0
{
create_transfer_signed(zebec_vault.to_acc
ount_info(),ctx.accounts.sender.to_accoun
t_info(),amount)?;
}
else
{
//Check remaining amount after withdrawal
let allowed_amt = zebec_vault.lamports()
- amount;
//if remaining amount is lesser then the
required amount for stream stop making
withdrawal
if allowed_amt < withdraw_state.amount {
    return
Err(ErrorCode::StreamedAmt.into());
}
create_transfer_signed(zebec_vault.to_acc
ount_info(),ctx.accounts.sender.to_accoun
t_info(),amount)?;
```

```
}
Ok(())
```

```
}
Ok(())
```

Figure 7.1: Example showing code duplication between two instructions

Exploit Scenario

Alice, a Zebec developer, is asked to fix a bug in the payment stream contract. Alice does not realize that the code with the bug exists in several places in the contract and therefore does not fix all instances. Mallory discovers and exploits remaining instances of the bug.

Recommendations

Short term, refactor the payment stream contract to eliminate the code duplication. This will reduce the likelihood of an incomplete fix for a bug affecting duplicated code.

Long term, adopt code practices that discourage code duplication to prevent this problem from recurring.

8. Inconsistent use of checked math	
Severity: **Informational**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-ZEB-8
Target: Various locations in the payment stream contract	

Description
The payment stream contract uses checked math fairly consistently; however, there are several instances in which unchecked math is used without explanation. All instances of unchecked math should use their respective checked math operation unless a justification is documented. This will eliminate a source of undefined behavior in the code.

The following are some of the uses of unchecked math in the payment stream contracts:

```
let allowed_amt = vault_token_account.amount - amount;
```

Figure 8.1: `programs/zebec/src/processor/token_stream/mod.rs#L301`

```
data_account.paused_amt += allowed_amt_now - amount_paused_at;
```

Figure 8.2: `programs/zebec/src/processor/token_stream/mod.rs#L146`

```
ctx.accounts.withdraw_data.amount += amount;
```

Figure 8.3: `programs/zebec/src/processor/token_stream/mod.rs#L26`

Exploit Scenario #1
Bob, a user, sends a payload that causes undefined behavior, and he receives fewer lamports than he anticipated due to arithmetic errors.

Exploit Scenario #2
Mallory, an attacker, crafts a payload that exploits the arithmetic errors. In turn, she receives excess lamports.

Recommendations
Short term, consider using checked math throughout the payment stream contract, or if overflow is desired, use wrapping/saturating arithmetic APIs.

Long term, avoid undefined behavior (e.g., overflows) and document how edge cases are handled. Consider setting Clippy's `integer-arithmetic` lint to deny in order to encourage the use of checked arithmetic.

Summary of Recommendations

The Zebec payment stream contract is a work in progress with multiple planned iterations. Trail of Bits recommends that Zebec address the findings detailed in this report and take the following additional steps prior to deployment:

- In order to strive towards the stated goal of the payment stream contract to be trustless, ensure that proper amounts are funded to the escrow account to cover pending operations.
- Extend the existing tests to cover edge cases and include property and fuzz testing.
- Use checked arithmetic and avoid casting to less wide types. If overflow is desired, use Rust's wrapping / saturating arithmetic API.
- Add code comments and create a technical specification for how streams should behave.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	**Description**
Arithmetic	The proper use of mathematical operations and semantics
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further	Further investigation is required to reach a meaningful conclusion.

Investigation Required	

C. Non-Security-Related Findings

The following recommendations are not associated with specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future.

- For accounts to be initialized, change their hard-coded sizes to implementations on the types. For example, change `space = 8 + 32 + 32 + 8` to the following:

```
space = Vault::MAX_SIZE
...
#[account]
pub struct Vault
{
    pub vault_address:Pubkey,
    pub owner:Pubkey,
    pub fee_percentage:u64,
}
impl Vault {
    // Size of all elements, plus 8 bytes for discriminator
    pub const MAX_SIZE: usize = 8 + 32 + 32 + 8;
}
```

- Name accounts consistently in a manner that hints at their use. For example, renaming `vault_data` to `fee_vault_data` clarifies that the account is tied to the fee features of the contract.

- Variables referring to the `withdraw_data` account are called both `withdraw_state` and `withdraw_data`. Use a common name for these variables to improve code clarity.

- Use `cargo fmt` as part of the CI process in order to ensure the code is formatted according to style guidelines.

- The code uses `Box` in several places to move allocations from the stack to the heap. Unless there is a specific reason for placing this data on the heap, use the stack for storage.

- In the following code, change `key;` to `key();`

```
data_account.sender = *ctx.accounts.source_account.key;
data_account.receiver = *ctx.accounts.dest_account.key;
```

- The following lines would panic if the account name changes:

```
let bump = ctx.bumps.get("zebec_vault").unwrap().to_le_bytes();
```

https://github.com/Zebec-protocol/zebec-anchor/blob/a313108b3bc22fa82667f7462bcb1e204c63c67f/programs/zebec/src/processor/token_stream/mod.rs#L102

https://github.com/Zebec-protocol/zebec-anchor/blob/a313108b3bc22fa82667f7462bcb1e204c63c67f/programs/zebec/src/processor/token_stream/mod.rs#L194

https://github.com/Zebec-protocol/zebec-anchor/blob/a313108b3bc22fa82667f7462bcb1e204c63c67f/programs/zebec/src/processor/token_stream/mod.rs#L234

https://github.com/Zebec-protocol/zebec-anchor/blob/a313108b3bc22fa82667f7462bcb1e204c63c67f/programs/zebec/src/processor/token_stream/mod.rs#L281

```
let bump = ctx.bumps.get("fee_vault").unwrap().to_le_bytes();
```

https://github.com/Zebec-protocol/zebec-anchor/blob/a313108b3bc22fa82667f7462bcb1e204c63c67f/programs/zebec/src/processor/create_fee_account/mod.rs#L19

D. Investigation of TOB-ZEB-2

`test-fuzz` is a collection of Rust macros and a Cargo subcommand that automates certain fuzzing-related tasks, such as generating a fuzzing corpus and implementing a fuzzing harness. Installation instructions and other documentation is located here.

Although this appendix explains only how we used `test-fuzz` to determine that `allowed_amount` has a rounding issue (TOB-ZEB-2), this testing strategy should be extended to cover more of the Zebec codebase. Additionally, these fuzz tests should check that system-wide invariants hold, regardless of user-input arguments or data accounts.

First, paste the code in Figure D.1 at the end of `programs/zebec/processor/token_stream/mod.rs` and add `test-fuzz` as a dependency in the `Cargo.toml` file. Next, run `cargo test` to seed the fuzzing corpus. Then, run `cargo test-fuzz allowed_amount_precision` until the tool reports several crashes.

```
#[cfg(test)]
mod tests {
    #[test_fuzz::test_fuzz]
    fn allowed_amount_precision(now: u64, start_time: u64, end_time: u64, amount:
u64) {
        if (now > end_time) {
            return;
        }
        assert_eq!((((now - start_time) as f64) /
        ((end_time - start_time) as f64)
        * amount as f64) as u64
        ,
        (now - start_time) * amount / (end_time - start_time)  as u64)
    }
    #[test]
    fn test_allowed_amount_precision() {
        allowed_amount_precision(36,10,100,100);
    }
}
```

Figure D.1: Fuzz test to identify precision loss

To triage crashes, one can pretty print the arguments by running `cargo test-fuzz --replay crashes --display crashes`.

```
id:000003,sig:06,src:000006,time:3128,execs:3359,op:havoc,rep:2: Args { now: 1,
start_time: 0, end_time: 214, amount: 214 }
thread 'processor::token_stream::tests::allowed_amount_precision_fuzz::entry'
panicked at 'assertion failed: `(left == right)`
  left: `0`,
```

```
 right: `1`'
id:000004,sig:06,src:000006,time:4195,execs:3981,op:havoc,rep:16: Args { now: 1,
start_time: 0, end_time: 28, amount: 10663680541141827583 }
thread 'processor::token_stream::tests::allowed_amount_precision_fuzz::entry'
panicked at 'assertion failed: `(left == right)`
  left: `380845733612208128`,
 right: `380845733612208127`'
id:000005,sig:06,src:000005,time:7415,execs:9531,op:havoc,rep:8: Args { now:
9223372036854775807, start_time: 7668084709108416512, end_time: 9223372036854775808,
amount: 1 }
thread 'processor::token_stream::tests::allowed_amount_precision_fuzz::entry'
panicked at 'assertion failed: `(left == right)`
  left: `1`,
 right: `0`'
```

Figure D.2: Pretty print of crashing test cases