Reserve Protocol Solana DTFs

Security Assessment

March 5, 2025

Prepared for:
Patrick Mckelvy
ABC Labs, LLC

Prepared by: **Samuel Moelius and Coriolan Pinhas**

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

Jeff Braswell, Project Manager
jeff.braswell@trailofbits.com

The following engineering director was associated with this project:

Josselin Feist, Engineering Director, Blockchain
josselin.feist@trailofbits.com

The following consultants were associated with this project:

Samuel Moelius, Consultant
samuel.moelius@trailofbits.com

Coriolan Pinhas, Consultant
coriolan.pinhas@trailofbits.com

Project Timeline

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

Date	Event
January 23, 2025	Pre-project kickoff call
January 31, 2025	Status update meeting #1
February 7, 2025	Delivery of report draft
February 7, 2025	Report readout meeting
March 5, 2025	Delivery of final comprehensive report

Executive Summary

Engagement Overview

ABC Labs engaged Trail of Bits to review the security of its Reserve Procol Solana Decentralized Token Folios (DTFs). DTFs represent baskets of funds of which users can hold shares. The DTF program is intended to be the protocol's main entrypoint, while most of the protocol's logic lies in the Folio program.

A team of two consultants conducted the review from January 27 to February 7, 2025, for a total of two engineer-weeks of effort. With full access to source code and documentation, we performed static and dynamic testing of the codebase, using automated and manual processes.

Observations and Impact

The Solana implementation is complex. To meet ABC Labs' security requirements, two programs are needed (`dtfs` and `folio`). However, this increases the size of the codebase by about 32% (see TOB-DTFSSOLANA-2). Moreover, it requires a reviewer to understand the relationship between the two programs, and how an attacker might use either of them in an unintended way.

Several instructions use "remaining accounts," which Anchor does not explicitly check. We gave special attention to such instructions since they must perform checks manually that Anchor would perform automatically. To the best of our knowledge, each such account's address is used in the derivation of some PDA with a known program ID. Thus, we know of no forgery-related flaws presently involving the use of remaining accounts. Nonetheless, we believe that the code could be hardened to protect against future changes.

Several of the issues we found might have been uncovered through more extensive negative testing (e.g., TOB-DTFSSOLANA-8, TOB-DTFSSOLANA-9, and TOB-DTFSSOLANA-11). Generally speaking, ensuring that code handles invalid values correctly is necessary to maintain the code's security.

Finally, the amount of code we received (11,109 lines) exceeded the amount we assumed we would receive (1,000 lines or less, for which we budgeted two engineer-weeks). To maximize the efficiency of our review, ABC Labs did provide a list of seven instructions to focus on.

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that ABC Labs take the following steps:

- **Remediate the findings disclosed in this report.** These findings should be addressed as part of a direct remediation or any refactor that may occur when addressing other recommendations.

- **Continue to look for ways to simplify the Solana implementation.** As mentioned above, the current implementation is complex. Generally speaking, complexity increases the attack surface. The risks associated with this may outweigh the benefits that the current solution provides.

- **Explicitly check the owner of each remaining account.** Specifically, add an expected owner argument to the `next_account` function. Have the function check the account's owner and return an error if it is not the expected one. This will help prevent attacks involving account forgery.

- **Expand the project's tests, particularly negative tests.** Several of the issues found during this review may have been uncovered through more extensive testing. We recommend expanding the project's tests to help uncover problems not found during this review.

Finding Severities and Categories

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

Severity	*Count*
High	3
Medium	1
Low	2
Informational	6
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	3
Data Validation	3
Error Reporting	1
Patching	3
Testing	1
Undefined Behavior	1

Project Goals

The engagement was scoped to provide a security assessment of the Reserve Procol Solana Decentralized Token Folios (DTFs). Specifically, we sought to answer the following non-exhaustive list of questions:

- Can a basket's underlying collateral be stolen?
- Can shares be minted for free (i.e., without depositing collateral)?
- Is it possible to make unauthorized changes to the Folio account?
- Can role-based access controls be bypassed?
- Can a user execute a trade that does not respect a DTF's pricing curve?
- Are all mathematical operations performed correctly and with appropriate precision?

Project Targets

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

dtfs-solana

Repository	https://github.com/reserve-protocol/dtfs-solana
Version	`f4273c898cbb752f21108438d04d27f56f73075c`
Type	Rust/Anchor
Platform	Solana

Project Coverage

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

- **Static analysis:** We ran Clippy over the codebase with pedantic lints enabled, and reviewed the results.
- **Test coverage review:** We verified that the Cargo and Anchor tests pass. We also reviewed the tests to determine whether important conditions are tested.
- **Manual review:** We manually review the code, focusing on the following elements:
    - Control transitions between the `dtfs` and `folio` programs
    - Correct use of the `program_registar` account
    - Validation of remaining accounts, i.e., accounts not explicitly checked by Anchor
    - Correct use of data resulting from `try_deserialize`
    - The following seven instructions:
        - `bid`
        - `distribute_fees`
        - `accrue_rewards`
        - `mint_folio_token`
        - `open_trade`
        - `claim_rewards`
        - `crank_fee_distribution`

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 found it difficult to understand the threat model motivating the current Solana implementation. Additionally, the current implementation lacks Solana-specific documentation. It is possible that with additional documentation, we would find additional flaws in the system.

- We focused our manual review on the seven instructions listed above. We also reviewed some instructions outside of that list, but they did not receive the same level of scrutiny.

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	We found one minor arithmetic problem related to an incorrect comparison. The implementation uses checked arithmetic where necessary.	**Satisfactory**
Auditing	The protocol emits events under important conditions and with pertinent information.	**Satisfactory**
Authentication / Access Controls	ABC Labs has gone to great lengths to reduce the influence that `folio` program owners have over `dtfs` program deployments. However, the use of two programs (as opposed to one) creates additional attack surface. Further investigation is required to judge the maturity of the implementation's authentication and access controls.	**Further Investigation Required**
Complexity Management	The two-program solution adds complexity to the implementation. We cannot see a way to satisfy ABC Labs' requirements with a single program. However, we are also not convinced that the two-program solution satisfies those requirements, as they are not documented. Further investigation is required to judge the implementation's complexity management maturity.	**Further Investigation Required**
Cryptography and Key Management	The programs do not perform hashing or manage keys, so this category is not applicable.	**Not Applicable**
Decentralization	As mentioned under Authentication / Access Controls, ABC Labs has tried to reduce the influence that `folio` program owners have over `dtfs` program deployments. Under the assumption that arbitrary users (and not ABC	**Satisfactory**

	Labs) deploy `folio` and `dtfs` programs, this category is satisfactory.	
Documentation	The implementation has no Solana-specific documentation. Very few functions have comments explaining their purpose and how they work. Some terminology has changed between the Solidity and Solana implementations, making it difficult to apply the Solidity documentation to the Solana implementation.	**Weak**
Low-Level Manipulation	The implementation does not use assembly or unsafe Rust. Furthermore, the implementation does little bit- or byte-level manipulation. As a result, this category is not applicable.	**Not Applicable**
Testing and Verification	The Anchor tests are not run in CI. When the Anchor tests are run locally, only one of ten test files is run. Several important conditions are not tested.	**Moderate**
Transaction Ordering	Based on our understanding of the code, a Folio owner can remove an asset from a basket. Thus, a user could bid on an asset, be front-run by the Folio owner, and have their transaction revert.	**Moderate**

Summary of Findings

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

ID	Title	Type	Severity
1	Incomplete building and testing instructions	Patching	Informational
2	No Solana-specific documentation	Patching	Informational
3	Testing deficiencies	Testing	Informational
4	Accounts structs should store fields in the same order to make them easier to compare	Patching	Informational
5	DTF owner key compromise allows manipulation of DAOFeeConfig	Access Controls	Medium
6	Trade instructions do not require a dtf_pogram_signer account	Access Controls	High
7	Comparison against wrong constant in accrue_rewards	Undefined Behavior	High
8	remove_from_registrar succeeds if passed program IDs are not in accepted_programs	Error Reporting	Informational
9	update_folio has error-prone interface that can lock out the owner	Data Validation	Low
10	add_tokens_to_basket does not check whether any of mints is Pubkey::default()	Data Validation	Informational
11	Incorrect TTL check in approve_trade	Data Validation	Low
12	Folio owner can rug DTF shareholders	Access Controls	High

Detailed Findings

1. Incomplete building and testing instructions

Severity: **Informational**	Difficulty: **High**
Type: Patching	Finding ID: TOB-DTFSSOLANA-1
Target: `README.md`	

Description

The `README.md` file contains instructions for building and testing the codebase. However, the instructions are incomplete. Requiring developers to infer building and testing instructions could cause them to perform either activity incorrectly.

We noticed the following deficiencies in the building and testing instructions:

- There are no instructions for installing `amman`. The `README.md` file should at least link to `https://github.com/metaplex-foundation/amman`.
- `@metaplex-foundation/amman` must be installed with `npm` to satisfy `.ammanrc.js` (figure 1.1).
- The `README.md` file says to run `./start-amman` (figure 1.2). The command should be `./start-amman.sh`.
- The `README.md` file does not mention that the user must create a `.env` file.
- The `.env.example` file contains the wrong value for `ADMIN_PUBKEY` (figure 1.3). There are no instructions for assigning it the correct value from the `utils/keys/keys-local.json` file (figure 1.4).

```
3    const { tmpLedgerDir } = require("@metaplex-foundation/amman");
```

Figure 1.1: Excerpt of `.ammanrc.js` (`dtfs-solana/.ammanrc.js#3`)

```
78    ./start-amman
```

Figure 1.2: Excerpt of `README.md` (`dtfs-solana/README.md#78`)

```
1    ADMIN_PUBKEY=99zqUzQGohamfYxyo8ykTEbi91iom3CLmwCA75FK5zTg
```

Figure 1.3: Excerpt of `.env.example` (`dtfs-solana/.env.example#1`)

```
35    141,
36    120,
// 29 lines deleted
66    124
```

Figure 1.4: Excerpt of `keys-local.json`
(`dtfs-solana/utils/keys/keys-local.json#35–66`)

Exploit Scenario

Alice is writing code to interact with the Reserve Protocol Solana DTFs program. Alice builds the DTFs program incorrectly. Her code seems to work correctly locally; however, it works incorrectly when interacting with the real DTFs program. Alice suffers financial loss as a result.

Recommendations

Short term, address the bullet points regarding the incomplete building and testing instructions above. This will save developers from having to figure out how to build and test the code themselves.

Long term, regularly review the building and testing instructions. Documentation must be kept up to date to be of value.

2. No Solana-specific documentation	
Severity: **Informational**	Difficulty: **High**
Type: Patching	Finding ID: TOB-DTFSSOLANA-2
Target: Entire codebase	

Description

The codebase contains no Solana-specific documentation. The Solana code is complex both in its implementation and its requirements. The reasons for this complexity should be documented.

ABC Labs communicated the following requirements for the Solana implementation:

- The implementation should allow for upgrades.
- There should be no single point of failure. For example, if an ABC Labs key were compromised, it should not allow all existing DTF deployments to be drained.

To achieve the above, ABC Labs chose to deploy two programs, `folio` and `dtfs`, which behave as follows:

- The core logic of the protocol lies in the `folio` program, and all assets are owned by the `folio` program.
- Users interact with the `dtfs` program, which generates CPI calls to the `folio` program.
- A program registrar account records the `dtfs` program deployments. The `folio` program can be configured to accept requests from only `dtfs` deployments in the registrar.
- The `folio` program is immutable and therefore cannot be compromised.
- The `dtfs` program is mutable. However, if the `dtfs` program were to be compromised, the compromised version would not appear in the program registrar.

ABC Labs' requirements are reasonable. However, the means they have chosen for satisfying them add significant additional complexity to the codebase. Specifically, the need for two programs can be seen as overhead introduced by the chosen solution.

In the commit we reviewed, there are 11,085 lines of non-test Rust source files. 3,551 of those belong to the `dtfs` program. Thus, 32% (3,551/11,085) of the code can be viewed as overhead introduced by the chosen solution.

This finding does not question ABC Labs' requirements, nor their means for satisfying them. However, we do recommend that both these details be recorded and made available in the repository.

Adding to the above, very few functions have comments explaining their purpose and how they work. Running `rg 'pub fn'` in the `programs` directory produces 252 hits. However, only three of those public functions are preceded by comments.

Finally, terminology seems to have changed between the Solidity and Solana implementations. Such changes make it difficult to apply existing Folio documentation to the Solana implementation. Example changes we noticed include:

- An "auction" is called a "trade."
- An "auction approver" is called a "trade proposer."

Exploit Scenario

Bob, a newly hired ABC Labs employee, is tasked with adding a new functionality into `folio` and `dtfs` programs. However, Bob misunderstands the threat model and the reason for the two programs. Bob's changes introduce a vulnerability.

Recommendations

Short term, take the following steps:

- Write comprehensive documentation describing the project's requirements and how they are achieved with the current implementation.
- Add doc comments to all publicly accessible functions in the `folio` and `dtfs` programs.
- Prepare a complete list of terms that changed between the Solidity and Solana implementations.

Taking these steps will make the code more accessible to developers and reviewers.

Long term, regularly review the documentation and doc comments to ensure they are accurate. Documentation must be kept up to date to be of value.

3. Testing deficiencies

Severity: **Informational**	Difficulty: **Low**
Type: Testing	Finding ID: TOB-DTFSSOLANA-3
Target: `.github/workflows/ci.yml`, `Anchor.toml`, `tests-ts/tests-dtfs.ts`, various test files	

Description

The current testing setup has at least three problems, described below. Tests should be run regularly to help ensure the code they test is free of bugs.

We noticed the following problems with the current testing setup:

- The Anchor tests are not run in CI (figure 3.1).

```
85    # - name: Anchor Test
86    #   run: |
87    #     export
PATH="/home/runner/.local/share/solana/install/active_release/bin:$PATH"
88    #     anchor test
```

Figure 3.1: Excerpt of `ci.yml` (`dtfs-solana/.github/workflows/ci.yml#85–88`)

- Only one of the tests in the `tests-ts` directory is currently run (figure 3.2).

```
25    [scripts]
26    test = "tsc && yarn run ts-mocha -p ./tsconfig.json -t 1000000
tests-ts/**/tests-dtfs.ts"
```

Figure 3.2: Excerpt of `Anchor.toml` (`dtfs-solana/Anchor.toml#25–26`)

- The tests in `tests-dtfs.ts` hang unless the line in figure 3.3 is commented out. (Also, the line seems unnecessary for the tests to pass.)

```
155    await createGovernanceAccounts(userTokenRecordPda, 1000);
```

Figure 3.3: Excerpt of `tests-dtfs.ts` (`dtfs-solana/tests-ts/tests-dtfs.ts#155`)

- None of the tests exercise the loops beginning on the following lines:[1]

[1] This problem was noticed while investigating TOB-DTFSSOLANA-7.

- dtfs-solana/programs/folio/src/instructions/crank/crank_fee_distribution.rs#L135
- dtfs-solana/programs/folio/src/instructions/stake/accrue_rewards.rs#L142
- dtfs-solana/programs/folio/src/instructions/stake/claim_rewards.rs#L129

Exploit Scenario
Alice, a Reserve Protocol developer, adds functionality to the `folio` and `dtfs` program. Alice's tests pass locally. However, unbeknownst to her, her tests pass because of peculiarities of her system. The bugs in her changes go unnoticed because the Anchor tests are not run in CI.

Recommendations
Short term, address each of the above bullets. Specifically:

- Run the Anchor tests in CI.
- Run all of the tests in the `tests-ts` directory.
- Remove the line that causes the `tests-dtfs.ts` tests to hang, and ensure that the tests behave correctly following the line's removal.
- Add tests to exercise each of the loops mentioned above.

Taking these steps will allow the tests to be run more frequently, thereby helping the code to remain free of bugs.

Long term, regularly review the project's tests to ensure that all important conditions are tested.

4. Accounts structs should store fields in the same order to make them easier to compare

Severity: **Informational**	Difficulty: **High**
Type: Patching	Finding ID: TOB-DTFSSOLANA-4
Target: various source files	

Description

The folio and dtfs programs have very similar interfaces. In particular, most dtfs instructions result in a CPI call to folio program. To make the two programs instructions easy to compare, their accounts structs should store fields in the same relative order. However, this is not currently the case.

Consider the open_trade instruction. The dtfs program's version of this instruction generates a CPI call to the folio program's instruction. As such, the dtfs program's instruction takes all accounts the folio program's instruction needs. The two instructions' accounts structs appear in figure 4.1 with highlighting to emphasize their similarities. If the structs stored their fields in the same relative order, they would be easier to compare.

```
#[derive(Accounts)]
pub struct OpenTrade<'info> {
    pub system_program: Program<'info, System>,

    #[account(mut)]
    pub trade_launcher: Signer<'info>,

    /// CHECK: Done within the folio program
    #[account()]
    pub actor: UncheckedAccount<'info>,

    /*
   DTF Program Accounts
   */
    #[account(
```

```
#[derive(Accounts)]
pub struct OpenTrade<'info> {
    pub system_program: Program<'info, System>,

    #[account(mut)]
    pub trade_launcher: Signer<'info>,

    #[account(
        seeds = [ACTOR_SEEDS, trade_launcher.key().as_ref(), folio.key().as_ref()],
        bump = actor.bump,
    )]
    pub actor: Account<'info, Actor>,

    #[account()]
```

```
            seeds = [DTF_PROGRAM_SIGNER_SEEDS],
            bump = dtf_program_signer.bump
        )]
        pub dtf_program_signer: Account<'info, DtfProgramSigner>,

        /// CHECK: DTF Program
        #[account(address = DTF_PROGRAM_ID)]
        pub dtf_program: UncheckedAccount<'info>,

        /// CHECK: DTF Program Data
        #[account(
            seeds = [DTF_PROGRAM_ID.as_ref()],
            bump,
            seeds::program = &bpf_loader_upgradeable::id()
        )]
        pub dtf_program_data: UncheckedAccount<'info>,

        /*
        Folio Program Accounts
        */
        /// CHECK: Folio Program
        #[account(address = FOLIO_ID)]
        pub folio_program: UncheckedAccount<'info>,

        /// CHECK: Done within the folio program
        #[account()]
        pub folio: UncheckedAccount<'info>,

        /// CHECK: Done within the folio program
        #[account(mut)]
```

```
        pub folio: AccountLoader<'info, Folio>,

        #[account(mut)]
        pub trade: AccountLoader<'info, Trade>,

        /*
        Account to validate
        */
        #[account(
            seeds = [PROGRAM_REGISTRAR_SEEDS],
            bump = program_registrar.bump
        )]
        pub program_registrar: Box<Account<'info, ProgramRegistrar>>,

        /// CHECK: DTF program used for creating owner record
        #[account()]
        pub dtf_program: UncheckedAccount<'info>,

        /// CHECK: DTF program data to validate program deployment slot
        #[account()]
        pub dtf_program_data: UncheckedAccount<'info>,
    }
```

```
        pub trade: UncheckedAccount<'info>,

        /// CHECK: Done within the folio program
        pub program_registrar: UncheckedAccount<'info>,
    }
```

Figure 5.1: `dtfs-solana/programs/dtfs/src/instructions/trade/open_trade.rs#L10-L59` *(left) and* `dtfs-solana/programs/folio/src/instructions/trade/open_trade.rs#L13-L48` *(right)*

Exploit Scenario

Alice, a Reserve Protocol developer, adds a new instruction to the `dtfs` and `folio` programs. Alice incorrectly performs a check in the `dtfs` program that should be performed in the `folio` program. Because the two instructions accounts structs are misaligned, the bug is missed during review.

Recommendations

Short term, store accounts that are common to both `dtfs` and `folio` instructions in the same positions within the instructions' accounts structs. Store other accounts after the common accounts. Taking these steps will make it easier to compare the two programs' accounts structs.

Long term, as new instructions are added to the two programs, ensure the standard in the short-term recommendation above is maintained. This will help prevent bugs from being introduced into the codebase.

5. DTF owner key compromise allows manipulation of DAOFeeConfig

Severity: **Medium**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-DTFSSOLANA-5
Target: `programs/folio/src/instructions/user/mint_folio/mint_folio_token.rs`	

Description[2]

`DAOFeeConfig` accounts hold information about how fees are paid for DTF deployments. Currently, `DAOFeeConfig` accounts are owned by the `dtfs` program, not the `folio` program. If the `dtfs` program owner's keys were compromised, its `DAOFeeConfig` account could be manipulated, leading to incorrect accounting in the `folio` program.

The `mint_folio_token` instruction provides an example (figure 5.1). The instruction obtains the `dao_fee_numerator` from the `DAOFeeConfig` account to compute a number of shares. If an attacker were to manipulate the account, the `folio` program could compute an incorrect number of shares.

```
// Mint folio token to user based on shares
let (dao_fee_numerator, dao_fee_denominator, _) =
DtfProgram::get_dao_fee_config(&ctx.accounts.dao_fee_config.to_account_info())?;

let fee_shares = ctx.accounts.folio.load_mut()?.calculate_fees_for_minting(
    shares,
    dao_fee_numerator,
    dao_fee_denominator,
)?;
```

Figure 5.1: Excerpt of `mint_folio_token.rs` (`dtfs-solana/programs/folio/src/instructions/user/mint_folio/mint_folio_token.rs#151–159`)

Exploit Scenario

Alice deploys `folio` and `dtfs` programs. Mallory steals the key that Alice used to deploy the `dtfs` program. Mallory manipulates the `DAOFeeConfig` account's contents. Alice's `folio` program performs incorrect accounting.

[2] This issue was found by ABC Labs, not Trail of Bits, during our review. The issue is included in this report at ABC Labs' request.

Recommendations

Short term, make the `DAOFeeConfig` account owned by the `folio` program rather than the `dtfs` program. This will make it more difficult for an attacker to manipulate the contents of the `DAOFeeConfig` account.

Long term, if new accounts must be added to the Solana implementation, lean toward having them owned by the `folio` program.

6. Trade instructions do not require a dtf_pogram_signer account

Severity: **High**	Difficulty: **Low**
Type: Access Controls	Finding ID: TOB-DTFSSOLANA-6
Target: `programs/folio/src/instructions/trade/*`	

Description

The Solana implementation uses a `dtf_pogram_signer` account (figure 6.1) to prove that the `dtfs` program is calling the `folio` program. However, none of the trade instructions require this account. An attacker could call `folio` program trade instructions while pretending to be the `dtfs` program.

```
/// PDA Seeds ["dtf_program_signer"]
#[account]
#[derive(Default, InitSpace)]
pub struct DtfProgramSigner {
    pub bump: u8,
}
```

Figure 6.1: Definition of the `DtfProgramSigner` struct (`dtfs-solana/programs/dtfs/src/state.rs#3–8`)

Exploit Scenario

Alice approves a trade that Mallory does not want to occur. Just after the trade begins, Mallory pretends to be the `dtfs` program and calls `kill_trade` to end the trade.

Recommendations

Short term, require a `dtf_pogram_signer` account in each trade instruction. This will make it more difficult for an attacker to pretend to be the `dtfs` program and call trade instructions.

Long term, if new instructions must be added to the Solana implementation, lean toward requiring a `dtf_pogram_signer` account. Aside from initialization and certain administrative instructions, it is difficult to imagine a `folio` instruction that should not be called from the `dtfs` program.

7. Comparison against wrong constant in accrue_rewards

Severity: **High**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-DTFSSOLANA-7
Target: `programs/folio/src/instructions/stake/accrue_rewards.rs`	

Description

The `accrue_rewards` instruction compares the `remaining_account_divider` variable to the constant 4. However, the variable can take only the values 5 or 7. Hence, the block guarded by the comparison is never executed.

```
 14    const REMAINING_ACCOUNT_DIVIDER_FOR_CALLER: usize = 5;
 15    const REMAINING_ACCOUNT_DIVIDER_FOR_USER: usize = 7;
 // ...
128        let remaining_account_divider = if ctx.accounts.user.key() ==
ctx.accounts.caller.key() {
129            REMAINING_ACCOUNT_DIVIDER_FOR_CALLER
130        } else {
131            REMAINING_ACCOUNT_DIVIDER_FOR_USER
132        };
 // ...
226            // All the logic for the extra user if user != caller
227            if remaining_account_divider == 4 {
```

Figure 7.1: Excerpt of `accrue_rewards.rs`
(`dtfs-solana/programs/folio/src/instructions/stake/accrue_rewards.rs#14–227`)

Exploit Scenario

The `accrue_rewards` instruction is called with a caller that is not the user, indicating that the block beginning on line 227 should be executed. However, because of the incorrect comparison on line 227, the block is never executed. The caller accrues rewards, but the user does not.

Recommendations

Short term, correct the code on line 227 of figure 7.1 to check for `REMAINING_ACCOUNT_DIVIDER_FOR_CALLER` rather than 4. This will ensure that proper accounting occurs in `accrue_rewards`.

Long term, avoid using constants like on line 227 of figure 7.1. One risks losing track of all the places where the constant is used. A better approach is to give the constant a name, and to use the name instead.

8. remove_from_registrar succeeds if passed program IDs are not in accepted_programs

Severity: **Informational**	Difficulty: **High**
Type: Error Reporting	Finding ID: TOB-DTFSSOLANA-8
Target: `programs/folio/src/utils/accounts/program_registrar.rs`	

Description

The `remove_from_registrar` function removes program IDs from the program registrar's `accepted_programs` array. If presented with an ID not in the array, the function succeeds. In such a case, the function should return an error to indicate to the caller that something has gone wrong.

The relevant code appears in figure 8.1. The function loops through each element of `accepted_programs`. If a `program` cannot be found in the argument `program_ids`, the function simply proceeds to the next entry. Thus, if the caller passes a program ID that is not in `accepted_programs`, the caller will not be alerted to the error.

```rust
    pub fn remove_from_registrar(&mut self, program_ids: Vec<Pubkey>) -> Result<()> {
        let mut new_programs = self.accepted_programs.to_vec();

        new_programs.iter_mut().for_each(|program| {
            if program_ids.contains(program) {
                *program = Pubkey::default();
            }
        });

        self.accepted_programs = new_programs.try_into().unwrap();

        Ok(())
    }
```

Figure 8.1: Excerpt of `program_registrar.rs` (`dtfs-solana/programs/folio/src/utils/accounts/program_registrar.rs#30–42`)

Exploit Scenario

Alice operates `dtfs` and `folio` deployments. Alice deploys a new copy of the `dtfs` program and calls `remove_from_registrar` to remove the old one. However, a typo causes the old deployment's program ID to remain. Alice is unaware of the typo, as the call to `remove_from_registrar` succeeds.

Recommendations

Short term, have `remove_from_registrar` return an error when it is passed a program ID that is not in `accepted_programs`. This will reduce the likelihood that such situations will go unnoticed.

Long term, expand the program registrar's tests. It is possible that this bug could have been found with more extensive negative (i.e., failing) tests.

9. update_folio has error-prone interface that can lock out the owner

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-DTFSSOLANA-9
Target: `programs/folio/src/instructions/owner/update_folio.rs`	

Description

The `update_folio` instruction allows `program_version` and `program_deployment_slot` arguments to be passed independently (figure 9.1). However, if the owner passes one without the other, they risk locking themselves out of the program.

The relevant code appears in figure 9.1. The first thing `update_folio` does is call validate on line 96, which calls `validate_folio_program_post_init` on line 70 of figure 9.2. The call to `validate_folio_program_post_init` results in a call to `validate_program_registrar`, part of which appears in figure 9.3.

```
85    pub fn handler(
86        ctx: Context<UpdateFolio>,
87        program_version: Option<Pubkey>,
88        program_deployment_slot: Option<u64>,
89        folio_fee: Option<u128>,
90        minting_fee: Option<u128>,
91        trade_delay: Option<u64>,
92        auction_length: Option<u64>,
93        fee_recipients_to_add: Vec<FeeRecipient>,
94        fee_recipients_to_remove: Vec<Pubkey>,
95    ) -> Result<()> {
96        ctx.accounts.validate()?;
```

Figure 9.1: `update_folio`'s function signature (`dtfs-solana/programs/folio/src/instructions/owner/update_folio.rs#85–95`)

```
68    pub fn validate(&self) -> Result<()> {
69        let folio = self.folio.load()?;
70        folio.validate_folio_program_post_init(
71            &self.folio.key(),
72            Some(&self.program_registrar),
73            Some(&self.dtf_program),
74            Some(&self.dtf_program_data),
75            Some(&self.actor),
76            Some(Role::Owner),
77            None, // Can update no matter the status
78        )?;
```

```

        Ok(())
    }
```

Figure 9.2: Definition of `UpdateFolio::validate` (`dtfs-solana/programs/folio/src/instructions/owner/update_folio.rs#68–81`)

```
    let deployment_slot = DtfProgram::get_program_deployment_slot(
        &dtf_program.key(),
        &dtf_program.to_account_info(),
        &dtf_program_data.to_account_info(),
    )?;

    check_condition!(
        self.program_deployment_slot == deployment_slot,
        InvalidProgram
    );
```

Figure 9.3: Excerpt of `Folio::validate_program_registrar` (`dtfs-solana/programs/folio/src/utils/accounts/folio.rs#87–96`)

The code in the figure 9.3 determines the deployment slot using values stored in the `Folio`. The determined deployment slot is compared to the one stored in the `Folio`. If there is a mismatch, the call fails.

If the owner calls `update_folio` with only one of the two values, and then calls `update_folio` again to correct the error, the second call will fail in `UpdateFolio::validate`.

Exploit Scenario

Alice calls `update_folio` to update her deployment's program ID and deployment slot; however, she forgets to pass the deployment slot. Alice notices her error and calls `update_folio` a second time to correct it. The second call to `update_folio` fails when it calls `UpdateFolio::validate`.

Recommendations

Short term, call `UpdateFolio::validate` before returning from `update_folio` to ensure that subsequent calls to `UpdateFolio::validate` will succeed. This will reduce the likelihood that Filio owners will lock themselves out of the program.

Long term, expand the `folio` program's tests. It is possible that this bug could have been found with more extensive negative (i.e., failing) tests.

10. add_tokens_to_basket does not check whether any of mints is Pubkey::default()

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-DTFSSOLANA-10
Target: `programs/folio/src/utils/accounts/folio_basket.rs`	

Description

The `add_tokens_to_basket` function takes a vector of public keys and adds them to the `token_amounts` array. The function uses `Pubkey::default()` as a sentinel value to represent an empty slot in the array. The function should check its inputs for this value and reject them when they are present.

The relevant code appears in figure 10.1. Consider the case where `token_amounts` is full, and the argument, `mints`, consists of just one value, `Pubkey::default()`. Arguably, the function could return success. Instead, the function will return failure because no free slots can be found in `token_amounts`.

```rust
pub fn add_tokens_to_basket(&mut self, mints: &Vec<Pubkey>) -> Result<()> {
    for mint in mints {
        if self.token_amounts.iter_mut().any(|ta| ta.mint == *mint) {
            // Continue if already exists or error out?
            continue;
        } else if let Some(slot) = self
            .token_amounts
            .iter_mut()
            .find(|ta| ta.mint == Pubkey::default())
        {
            slot.mint = *mint;
            slot.amount_for_minting = 0;
            slot.amount_for_redeeming = 0;
        } else {
            // No available slot found, return an error
            return Err(error!(MaxNumberOfTokensReached));
        }
    }

    Ok(())
}
```

Figure 10.1: Definition of `add_tokens_to_basket`
(`dtfs-solana/programs/folio/src/utils/accounts/folio_basket.rs#49–69`)

Exploit Scenario

Alice calls `mint_initial_shares`, but passes `Pubkey::default()` as the mint by mistake. The call succeeds despite the invalid mint value.

Recommendations

Short term, have `add_tokens_to_basket` check its argument for `Pubkey::default()` values and reject them when they are present. This will alert the caller to the problem when such values are passed by accident.

Long term, if a function takes public keys as arguments and uses `Pubkey::default()` as a sentinel value, have the function check for `Pubkey::default()` among its arguments. This will present users with more predictable and less error-prone interfaces.

11. Incorrect TTL check in approve_trade

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-DTFSSOLANA-11
Target: `programs/folio/src/instructions/trade/approve_trade.rs`	

Description

The `approve_trade` instruction performs an incorrect TTL check (figure 11.1). Specifically, the instruction checks that the provided TTL is at least `MAX_TTL` rather than no more than `MAX_TTL`. Thus, trades will be required to live longer than they should.

```
112     check_condition!(ttl >= MAX_TTL, InvalidTtl);
```

Figure 11.1: Excerpt of the `approve_trade` instruction
(`dtfs-solana/programs/folio/src/instructions/trade/approve_trade.rs#112`)

Exploit Scenario

Alice proposes a trade with an unreasonably large TTL, and forgets about the trade. Mallory obtains the `TradeLauncher` role through social engineering, and launches the forgotten trade. Mallory uses a Sybil address to drain the associated basket of one of its tokens.

Recommendations

Short term, correct the TTL check in figure 11.1. This will ensure that trades do not live longer than they should.

Long term, expand the `folio` program's tests. It is possible that this bug could have been found with more extensive negative (i.e., failing) tests.

12. Folio owner can rug DTF shareholders

Severity: **High**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-DTFSSOLANA-12
Target: various source files	

Description

A user who holds `Role::Owner` can remove arbitrary tokens from a basket (figure 12.1). In doing so, the user decreases the value of all shares associated with the basket. Based on our understanding, this violates ABC Labs' threat model, as it does not require the user to make a code change to the `folio`.

```rust
impl RemoveFromBasket<'_> {
    pub fn validate(&self, folio: &Folio) -> Result<()> {
        folio.validate_folio_program_post_init(
            &self.folio.key(),
            Some(&self.program_registrar),
            Some(&self.dtf_program),
            Some(&self.dtf_program_data),
            Some(&self.actor),
            Some(Role::Owner),
            Some(vec![FolioStatus::Initializing, FolioStatus::Initialized]),
        )?;

        Ok(())
    }
}

pub fn handler<'info>(
    ctx: Context<'_, '_, 'info, 'info, RemoveFromBasket<'info>>,
    removed_mints: Vec<Pubkey>,
) -> Result<()> {
    {
        let folio = ctx.accounts.folio.load()?;
        ctx.accounts.validate(&folio)?;
    }

    ctx.accounts
        .folio_basket
        .load_mut()?
        .remove_tokens_from_basket(&removed_mints)?;

    Ok(())
}
```

Figure 12.1: Excerpt of `remove_from_basket.rs`
(`dtfs-solana/programs/folio/src/instructions/owner/remove_from_basket.rs#57–88`)

Exploit Scenario

Mallory deploys `folio` and `dtfs` programs. Over time, Mallory's deployments obtain a significant number of users. Mallory removes all tokens from all baskets and transfers the tokens to herself.

Recommendations

Short term, document the fact that users with `Role::Owner` must be trusted, reputable parties. This will reduce the likelihood of such users rugging other users.

Long term, document ABC Labs' threat model. This will make it easier to uncover problems (e.g., when users have greater authority than they should).

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
Auditing	The use of event auditing and logging to support monitoring
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
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage
Transaction Ordering	The system's resistance to transaction-ordering attacks

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeded 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 does not apply to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Non-Security-Related Recommendations

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

- **Either correct the language, or correct the code, in figure C.1.** Currently, the code does not render correctly because it is not valid TOML.

~~~
```toml
folio = FOLIO_PROGRAM_ID
dtfs = DTF_PROGRAM_ID

cluster = Devnet / Localnet / Mainnet
```
~~~

Figure C.1: Code block that does not render correctly (`dtfs-solana/README.md#61–66`)

- **Eliminate the crate-wide `allow` that currently exists in `shared/src/lib.rs` (figure C.2).**

```
#![allow(clippy::all)]
```

Figure C.2: Crate-wide allow in `shared/src/lib.rs` (`dtfs-solana/shared/src/lib.rs#1`)

- **Eliminate the two uses of `to_account_info` in figure C.3.** The values to which they are applied are already of type `AccountInfo`. As a result, these uses are unnecessary.

```
let deployment_slot = DtfProgram::get_program_deployment_slot(
    &dtf_program.key(),
    &dtf_program.to_account_info(),
    &dtf_program_data.to_account_info(),
)?;
```

Figure C.3: Unnecessary uses of `to_account_info` (`dtfs-solana/programs/folio/src/utils/accounts/folio.rs#87–91`)

- **Have `accrue_rewards` and `claim_rewards` check the actor's role in `validate_folio_program_post_init`.** Most other instructions that require an actor to have a role perform the check in this way. Also, this will result in more concise code.

```
folio.validate_folio_program_post_init(
    &self.folio.key(),
```

```
        Some(&self.program_registrar),
        Some(&self.dtf_program),
        Some(&self.dtf_program_data),
        None,
        None,
        Some(vec![FolioStatus::Initializing, FolioStatus::Initialized]),
    )?;

    // Validate that the folio owner is the correct one
    check_condition!(
        Role::has_role(self.actor.roles, Role::Owner),
        InvalidFolioOwner
    );
```

Figure C.4: Excerpt of `accrue_rewards.rs` *(`dtfs-solana/programs/folio/src/instructions/stake/accrue_rewards.rs#85–99`)*

```
    folio.validate_folio_program_post_init(
        &self.folio.key(),
        Some(&self.program_registrar),
        Some(&self.dtf_program),
        Some(&self.dtf_program_data),
        None,
        None,
        Some(vec![FolioStatus::Initializing, FolioStatus::Initialized]),
    )?;

    // Validate that the folio owner is the correct one
    check_condition!(
        Role::has_role(self.actor.roles, Role::Owner),
        InvalidFolioOwner
    );
```

Figure C.5: Excerpt of `claim_rewards.rs` *(`dtfs-solana/programs/folio/src/instructions/stake/claim_rewards.rs#77–91`)*

- **Either remove the `FolioProgramSigner` struct (figure C.6), or add a comment explaining why it is unused.**

```
    /// PDA Seeds ["folio_program_signer"]
    #[account]
    #[derive(Default, InitSpace)]
    pub struct FolioProgramSigner {
        pub bump: u8,
    }
```

Figure C.6: The definition of `FolioProgramSigner` *(`dtfs-solana/programs/folio/src/state.rs#11–16`)*

- **Complete the terminology changes in the Solana code.** Currently, the changes appear to be incomplete, as can be seen by the use of both “trade” and “auction” in figure C.7. See also TOB-DTFSSOLANA-2.

```
/*
Trade related properties
*/
pub trade_delay: u64,
pub auction_length: u64,
```

Figure C.7: Apparent incomplete terminology changes
(`dtfs-solana/programs/folio/src/state.rs#86–90`)

- **Simplify the code in figure C.8.** The code could be rewritten as in figure C.9.

```
trade_status.is_some() && trade_status.unwrap() == TradeStatus::APPROVED,
```

Figure C.8: Code that could be simplified
(`dtfs-solana/programs/folio/src/utils/accounts/trade.rs#68`)

```
trade_status == Some(TradeStatus::APPROVED),
```

Figure C.9: Proposed rewrite of the code in figure C.8

- **Remove the `GovernanceUtil::folio_owner_is_realm` function (figure C.10).** The function is currently unused, and it does not check its argument’s owner as its name suggests.

```
pub fn folio_owner_is_realm(realm: &AccountInfo) -> Result<()> {
    let realm_account_data = realm.try_borrow_data()?;
    spl_governance::state::realm::RealmV2::deserialize(&mut
&realm_account_data[..])?;

    Ok(())
}
```

Figure C.10: Definition of `GovernanceUtil::folio_owner_is_realm`
(`dtfs-solana/programs/folio/src/utils/external/governance.rs#39–44`)

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 100+ 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 and 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 X 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

Trail of Bits considers this report public information; it is licensed to ABC Labs under the terms of the project statement of work and has been made public at ABC Labs's request. Material within this report may not be reproduced or distributed in part or in whole without Trail of Bits' express written permission.

The sole canonical source for Trail of Bits publications is the Trail of Bits Publications page. Reports accessed through sources other than that page may have been modified and should not be considered authentic.

Test Coverage Disclaimer

Trail of Bits performed all activities associated with this project in accordance with a statement of work and an agreed-upon project plan.

Security assessment projects are time-boxed and often rely on information 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 software controls and security properties. 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. A project's time and resource constraints also limit their use.