Snowflake

Audit

Presented by:

OtterSec contact@osec.io

Nathan Ostrowski n@osec.io

Robert Chen notdeghost@osec.io

Contents

Appendices

01 | Executive Summary

Overview

Snowflake engaged OtterSec to perform an assessment of the `snowflake-safe-program` application. This assessment was conducted between July 12th and July 16th, 2022.

Critical vulnerabilities were communicated to the team prior to the delivery of the report to speed up remediation. After delivering our audit report, we worked closely with the team over to streamline patches and confirm remediation.

We delivered final confirmation of the patches July 26th, 2022.

Key Findings

The following is a summary of the major findings in this audit.

- 4 findings total
- 1 medium issue found related to bad user input
  - OS-SNW-ADV-00: Explicate Crank Payment Implications

02 | Scope

The source code was delivered to us in a git repository at github.com/snowflake-so/snowflake-safe-program. This audit was performed against commit `2dc1929`.

There was a total of 1 program included in this audit. A brief description of the program is as follows. A full list of program files and hashes can be found in Appendix A.

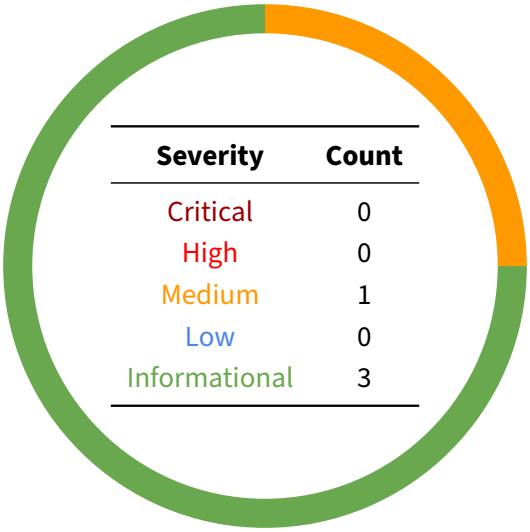
Name	Description
snowflake	Multisig application built on Solana with a focus on usability

03 | Findings

Overall, we report 4 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings don't have an immediate impact but will help mitigate future vulnerabilities.

The below chart displays the findings by severity.



Severity	Count
Critical	0
High	0
Medium	1
Low	0
Informational	3

04 | Vulnerabilities

Here we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have **immediate** security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix D.

ID	Severity	Status	Description
OS-SNW-ADV-00	Medium	Resolved	Explicate implications of crank payment to users

OS-SNW-ADV-00 [Med] [Resolved] | Explicate Crank Payment Implications

Description

The `charge_fee` function in `execute_scheduled_multisig_flow()` refunds the Solana transaction fee to the caller. Calling a flow multiple times in the same transaction will lead to more funds refunded than the transaction cost.

`charge_fee()`, shown below, refunds invocation fees based on `Fees::fee_calculator`.

snowflake-rust-v2/programs/snowflake/src/common/fee.rs RUST

```
pub fn charge_fee(ctx: &Context<ExecuteMultisigFlow>) -> Result<()> {
    let safe = &ctx.accounts.safe;
    let fee =
    ↪  Fees::get().unwrap().fee_calculator.lamports_per_signature;
    let safe_signer = &ctx.accounts.safe_signer;
    let caller = &ctx.accounts.caller;

    let ix =
    ↪  solana_program::system_instruction::transfer(&safe_signer.key,
    ↪  &caller.key, fee);
```

The following is the internal fee calculator logic, which notably calculates the fee based on the required signatures of the `message` object.

.cargo/registry/src/github.com-1ecc6299db9ec823/solana-program-1.9.18/src/fee_calculator.rs RUST

```
    #[deprecated(
        since = "1.9.0",
        note = "Please do not use, will no longer be available in the
    ↪  future"
    )]
    pub fn calculate_fee(&self, message: &Message) -> u64 {
        >{...}
        self.lamports_per_signature
            * (u64::from(message.header.num_required_signatures) +
    ↪  num_signatures)
    }
}
```

This fee reflects the cost to send a transaction. However, a transaction might include many invocations of the same instruction, making it profitable to repeatedly call this instruction.

Proof of Concept

A malicious user who wished to create and exploit a vulnerable configuration could:

1. Consider a multisig which has a`TriggerType::Program` flow with `RECURRING_FOREVER` `remaining_runs`. For example, a crank function.
2. Execute the flow once using `execute_multisig_flow()` to switch the flow to `ExecutionInProgress`
3. Chain many invocations of the `execute_scheduled_multisig_flow()` instruction together on a single transaction. For each successful instruction invocation, the safe will pay the calling keypair the total cost of the transaction, draining funds.

Remediation

The developers may choose to:

1. Enforce no more than a single `execute_scheduled_multisig_flow()` invocation per transaction
2. Place some upper limit on the total executions of a program flow (i.e. the `remaining_runs` attribute)

Patch

Enforce stricter constraints on `remaining_runs`, resolved in 6e1c415.

05 | General Findings

Here we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they do represent antipatterns and could introduce a vulnerability in the future.

ID	Status	Description
OS-SNW-SUG-00	TODO	Centralize validation for flow creation and execution.
OS-SNW-SUG-01	TODO	The pay_fee_from attribute is nonfunctional.
OS-SNW-SUG-02	TODO	Fees::fee_calculator is deprecated.

OS-SNW-SUG-00 | Centralize Flow Validations

Description

Currently, the validation of flows is spread throughout flow creation and execution. Some attributes–for example, `TriggerType`–are validated upon flow creation, but others like the `cron` string are only validated upon flow execution.

src/common/schedule.rs RUST

```
    let schedule = SnowSchedule::parse(_cron).unwrap();
    let next_execution = schedule
        .next_event(&SnowTime::from_time_ts(local_time))
        .unwrap()
        .to_time_ts(utc_offset)
        .unwrap();
    next_execution
```

As a result, a user may create and approve a flow with an invalid `cron` string.

Requiring certain flows to pass through `execute_multisig_flow()` before passing through `execute_scheduled_multisig_flow()` could lead to problems down the line.

For example, certain types of flows (i.e. flows with `TriggerType::Program`) can be run via either flow execution function, but checks on appropriate information on these flows is spread out across these two execute functions.

Remediation

Move all validation logic to a central function and mirror this function across both flow creation and execution. This validation function should switch based on attributes like `TriggerType` that are mutually exclusive.

OS-SNW-SUG-01 | Non-functional Fee-payer Attribute

Description

The `Flow` attribute `pay_fee_from` is unused. The safe invoking the action will always pay the cost of the invocation. This is due to the fact that the nested CPI signer is constructed from the safe PDA, which is passed into `invoke_signed`. This attribute is potentially misleading to users and future code reviewers.

src/state/flow.rs RUST

```rust
    pub last_scheduled_execution: i64,
    pub expiry_date: i64,
    pub expire_on_complete: bool,
    pub app_id: Pubkey,
    pub pay_fee_from: u8,
    pub user_utc_offset: i32,
    pub custom_compute_budget: u32,
```

Remediation

Remove this attribute or build out its functionality.

OS-SNW-SUG-02 | Deprecated Fee Calculator

Description

`Fees::fee_calculator` is deprecated by Solana. It will not be available in the future, and may give inaccurate information when completely unsupported.

src/state/flow.rs RUST

```
        let safe = &ctx.accounts.safe;
-       let fee =
    ↪   Fees::get().unwrap().fee_calculator.lamports_per_signature;
        let safe_signer = &ctx.accounts.safe_signer;
        let caller = &ctx.accounts.caller;
```

Remediation

Replace `Fees::fee_calculator` with `solana_program::FeeRateGoverner`, as defined here.

A | Program Files

Below are the files in scope for this audit and their corresponding SHA256 hashes.

```
snowflake
  Cargo.lock                                  9ebb567b0570718a97bc5400645fd6ef
  Cargo.toml                                  7b772f6acb490423fee3d67ad36ecbed
  Xargo.toml                                  815f2dfb6197712a703a8e1f75b03c69
  test.rs                                     1d12cb4d96f719009cd854ba0d026301
  src
    error.rs                                  c7eea7fb2795d31f826795d809b992de
    lib.rs                                    5bf12c0f69cccc61d3d2c3bdf4c72591
    test.rs                                   be535de1cca67c289d3f64858ddf8bdb
    common
      fee.rs                                  9deba91afbef613569f41658b3a9c524
      mod.rs                                  596a14e1aa7efd6800dd7e67278f07a4
      schedule.rs                             24fda08629816c192bd2a2745f758165
    instructions
      abort_flow.rs                           5852758a15acb19e97c4ba40bbc5a010
      approve_proposal.rs                     10fce0f16c979e5c5ba423892a8ddecf
      create_flow.rs                          e5542e3e14769eb471f817165465150c
      create_safe.rs                          3d26c5075f15bcfa5a4dd02a9b7dacdc
      delete_flow.rs                          fc4dabab6723c106f16bf25bd58d7c61
      do_execute_multisig_flow.rs             1a6b3bb2c20f7bcf7bd6194bc42048d7
      execute_multisig_flow.rs                9bd2264b38d6915175e999c887accc4b
      execute_scheduled_multisig_flow.rs      bce77f89045302137efd1d89e2ba78ab
      mod.rs                                  24be02a4ddaf86331153da9a539785c4
      update_safe.rs                          7b5546f54e579f5f0039aa1d62a2f7dd
    state
      action.rs                               66256ae2f1696a719e024054e84fc3c7
      approval_record.rs                      f1fdf75bfb319df4d923925aef063b0d
      flow.rs                                 9f36c353dda6013753cac2d81718907a
      mod.rs                                  5a7b75b439bc05902a4a6d2abdf1ac43
      safe.rs                                 876eecee54e1a0d2c2f1a0997a063e5c
      static_config.rs                        024bb5a52a8237458df83921d3c485a7
      target_acount_spec.rs                   9ac3e8ac8b7bc0c9c36db092c08775d4
```

B | Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an onchain program. In other words, there is no way to steal tokens or deny service, ignoring any Solana specific quirks such as account ownership issues. An example of a design vulnerability would be an onchain oracle which could be manipulated by flash loans or large deposits.

On the other hand, auditing the implementation of the program requires a deep understanding of Solana's execution model. Some common implementation vulnerabilities include account ownership issues, arithmetic overflows, and rounding bugs. For a non-exhaustive list of security issues we check for, see Appendix C.

Implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to get a comprehensive understanding of the program first. In our audits, we always approach any target in a team of two. This allows us to share thoughts and collaborate, picking up on details that the other missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.

C | Implementation Security Checklist

Unsafe arithmetic

Integer underflows or overflows	Unconstrained input sizes could lead to integer over or underflows, causing potentially unexpected behavior. Ensure that for unchecked arithmetic, all integers are properly bounded.
Rounding	Rounding should always be done against the user to avoid potentially exploitable off-by-one vulnerabilities.
Conversions	Rust as conversions can cause truncation if the source value does not fit into the destination type. While this is not undefined behavior, such truncation could still lead to unexpected behavior by the program.

Account security

Account Ownership	Account ownership should be properly checked to avoid type confusion attacks. For Anchor, the safety of unchecked accounts should be clearly justified and immediately obvious.
Accounts	For non-Anchor programs, the type of the account should be explicitly validated to avoid type confusion attacks.
Signer Checks	Privileged operations should ensure that the operation is signed by the correct accounts.
PDA Seeds	PDA seeds are uniquely chosen to differentiate between different object classes, avoiding collision.

Input validation

Timestamps	Timestamp inputs should be properly validated against the current clock time. Timestamps which are meant to be in the future should be explicitly validated so.
Numbers	Sane limits should be put on numerical input data to mitigate the risk of unexpected over and underflows. Input data should be constrained to the smallest size type possible, and upcasted for unchecked arithmetic.
Strings	Strings should have sane size restrictions to prevent denial of service conditions
Internal State	If there is internal state, ensure that there is explicit validation on the input account's state before engaging in any state transitions. For example, only open accounts should be eligible for closing.

Miscellaneous

Libraries	Out of date libraries should not include any publicly disclosed vulnerabilities
Clippy	cargo clippy is an effective linter to detect potential anti-patterns.

D | Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings can be found in the General Findings section.

Critical	Vulnerabilities which immediately lead to loss of user funds with minimal preconditions Examples: • Misconfigured authority/token account validation • Rounding errors on token transfers
High	Vulnerabilities which could lead to loss of user funds but are potentially difficult to exploit. Examples: • Loss of funds requiring specific victim interactions • Exploitation involving high capital requirement with respect to payout
Medium	Vulnerabilities which could lead to denial of service scenarios or degraded usability. Examples: • Malicious input cause computation limit exhaustion • Forced exceptions preventing normal use
Low	Low probability vulnerabilities which could still be exploitable but require extenuating circumstances or undue risk. Examples: • Oracle manipulation with large capital requirements and multiple transactions
Informational	Best practices to mitigate future security risks. These are classified as general findings. Examples: • Explicit assertion of critical internal invariants • Improved input validation • Uncaught Rust errors (vector out of bounds indexing)