Folome Online

Security Assessment

June 7th, 2024 — Prepared by OtterSec

Nicola Vella email@osec.io

Robert Chen r@osec.io

Table of Contents

01 — Executive Summary

Overview

Folome engaged OtterSec to assess the `follows-solana` program. This assessment was conducted between May 16th and June 4th, 2024. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 3 findings throughout this audit engagement.

In particular, we identified a high-risk vulnerability, where the current code calculates Follows fees based on the price after wrapper fees, resulting in inflated fees for users and unintended revenue for the Follows program (OS-FOL-ADV-00).

We also recommended modifying the code base for improved efficiency and for inclusion of missing validations (OS-FOL-SUG-00) and made suggestions to ensure adherence to coding best practices (OS-FOL-SUG-01).

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/Folome-online/follows-solana. This audit was performed against commit 8ab7781.

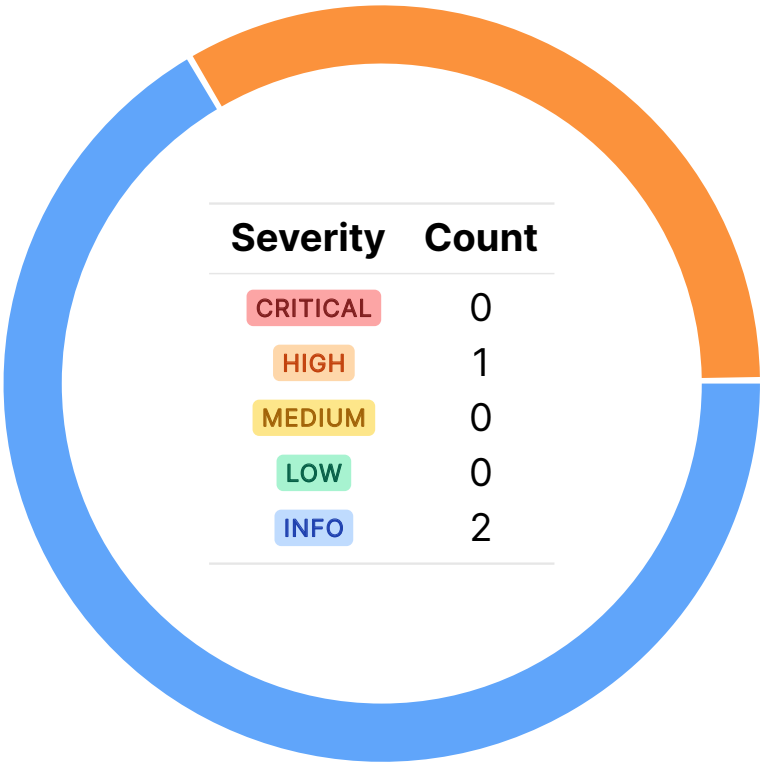
A brief description of the programs is as follows:

Name	Description
follows-solana	Follows protocol is a platform for managing and trading tokens on the Solana blockchain including features for platform initialization, admin management, fee structure configuration, token creation and management, buying and selling tokens, and distributing lamports. pen$_{spark}$

03 — Findings

Overall, we reported 3 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 do not have an immediate impact but will aid in mitigating future vulnerabilities.



Severity	Count
CRITICAL	0
HIGH	1
MEDIUM	0
LOW	0
INFO	2

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 A.

ID	Severity	Status	Description
OS-FOL-ADV-00	HIGH	RESOLVED	The current code calculates Follows fees based on the price after wrapper fees, resulting in inflated fees for users and unintended revenue for the Follows program.

Incorrect Follows Fee Calculation HIGH OS-FOL-ADV-00

Description

In the `buy_token::handler` function, the Follows fees (`follows_fees_lamports`) are calculated based on the price after the wrapper program's fees have been added (`follows_price_after_wrapper_fees`). This results in the Follows program receiving a higher effective price for the transaction, which can lead to the collection of higher fees than originally intended. Consequently, users end up paying more for their wrapped token purchases due to these inflated Follows fees.

>_ *follows-token-wrapper/src/instructions/buy_token.rs* rust

```rust
// Simulation will return price after all fees
pub fn handler(ctx: Context<BuyToken>, amount: u64, max_price: u64) -> Result<u64> {
    [...]
    // Take fees for wrapper
    let follows_price = ctx.accounts.token_info.get_token_buy_price(amount)?;
    let wrapper_fees_lamports = ctx.accounts.platform_info.get_fees_lamports(follows_price);
    let follows_price_after_wrapper_fees = follows_price.try_add(wrapper_fees_lamports.total());

    // Get expected final price
    let follows_fees_lamports =
        ↪  ctx.accounts.follows_platform_info.get_fees_lamports(follows_price_after_wrapper_fees);
    let follows_token_price_after_all_fees =
        ↪  follows_price_after_wrapper_fees.try_add(follows_fees_lamports.total());
    [...]
}
```

Remediation

Ensure the Follows fees are calculated based on the original buy price (`follows_price`) before any fees are added.

Patch

Fixed in 117d9cc.

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 represent anti-patterns and may result in security issues in the future.

ID	Description
OS-FOL-SUG-00	Recommendation for modifying the code base for improved efficiency and for inclusion of missing validations.
OS-FOL-SUG-01	Suggestions to ensure adherence to coding best practices.

Code Refactoring OS-FOL-SUG-00

Description

1. Include a sanity check in the `buy_token` instruction to ensure that the minted amount of wrapped tokens matches the requested buy amount. This check should be similar to the one implemented in the `sell_token` instruction. Doing so will prevent discrepancies between the requested buy amount (`amount`) and the actual amount of underlying tokens received in escrow.

>_ *follows-token-wrapper/src/instructions/sell_token.rs* rust

```rust
pub fn handler(ctx: Context<SellToken>, amount: u64, min_sale_price: u64) -> Result<u64> {
    [...]
    ctx.accounts.log_amount_to_burn(amount_to_burn, amount, spl_token_to_burn);
    require!(amount_to_burn == amount,
        ↪  FollowsError::WrapperProgramDiffInTokenBurnedAndSold);
    [...]
}
```

2. The `WALLET_RENT_EXEMPT_LAMPORTS` checks in the codebase are generally unnecessary and can be removed, except for those in the `buy_token` and `sell_token` instructions that validate transfers to the referrer. These checks are crucial to prevent a denial of service when transferring to an empty wallet, as Solana requires a minimum balance (`solana rent 0`) to avoid transaction reversion.

Remediation

Implement the above mentioned suggestions.

Patch

1. Fixed in f9d3ae9.
2. Fixed in 0571dd8.

Code Maturity OS-FOL-SUG-01

Description

1. There is a potential for overflow in the `TokenInfoV1::get_token_price` function due to the limitations of the `u64` data type. This function calculates the token price using a formula that factors in the current supply, the amount being bought or sold, and the alpha value. Given the current alpha value of one, the technical supply cap is approximately 12,148,002 tokens. At this supply level, the price calculation would exceed the limits of a `u64`, leading to an overflow. Consequently, this imposes a practical limit on the maximum token supply before encountering overflow issues.

>_ $token_{info.rs}$ rust

```rust
pub fn get_token_price(current_supply: u64, amount: u64, alpha: u8) -> Result<u64> {
    require!(amount > 0u64, FollowsError::AmountCannotBeZero);
    [...]
    let price_u64 = u64::try_from(final_price).unwrap_with_msg("as u64");
    Ok(price_u64)
}
```

2. Wrapped token arbitrage, arising from price discrepancies between Decentralized Exchanges (DEXs) and the Follows platform's token curve, can be highly risky. The price of the wrapped token can fluctuate rapidly, potentially leading to losses if the price drops before the arbitrage can be completed. Therefore, it's crucial to ensure that users are fully aware of the risks associated with wrapped tokens and are advised to carefully evaluate these risks before engaging in arbitrage activities.

Remediation

1. Consider the above recommendation to prevent a potential revert due to overflow.
2. Ensure the risks of engaging in wrapped token arbitrage are clearly conveyed to the users.

A — Vulnerability Rating Scale

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

CRITICAL

Vulnerabilities that immediately result in a loss of user funds with minimal preconditions.

Examples:

- Misconfigured authority or access control validation.
- Improperly designed economic incentives leading to loss of funds.

HIGH

Vulnerabilities that may result in a 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 that may result in denial of service scenarios or degraded usability.

Examples:

- Computational limit exhaustion through malicious input.
- Forced exceptions in the normal user flow.

LOW

Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk.

Examples:

- Oracle manipulation with large capital requirements and multiple transactions.

INFO

Best practices to mitigate future security risks. These are classified as general findings.

Examples:

- Explicit assertion of critical internal invariants.
- Improved input validation.

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 on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, 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 comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. 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.