follows.app

Smart Contract Security Assessment

October 26, 2023

ABSTRACT

Dedaub was commissioned to audit the follows.app protocol implementation, located at the repository https://github.com/Folome-online/smart-contracts, commit number 91f02d8. Following this a number of fixes were audited, up to commit number a510f11.

Follows is a social media app allowing users to buy and sell access to their favorite celebrities. The celebrity can create a token and allow users to buy and sell shares in this token. Owners of shares receive access to private chats and other benefits. When a token is created, an immutable parameter alpha can be specified, which affects the computation of the price in terms of native tokens. Save for this parameter, which is decided by the creator of the token at its creation, the price is computed algorithmically depending on the supply and demand. The price has a polynomial dependence on both the current supply (polynomial of 2nd degree) and the trading amount (3rd degree polynomial). In addition to the price of the token, users also pay fees, which are distributed between the platform, the creator of the token and (possibly) the token's referrer. The percentage of these fees is decided by the owner of the contract and can be changed. Users can also sell their tokens and get back native tokens (losing access to the benefits provided by the celebrity). When a user sells his tokens, there is no guarantee that he will get back all the native tokens he paid to buy the tokens.

SETTING AND CAVEATS

The audit scope consists of the following files:

```
contracts/
├── FolowV1.sol
```

Two auditors worked on the code for 4 days.

The audit's main target is security threats, i.e., what the community understanding would likely call "hacking", rather than regular use of the protocol. Functional correctness (i.e., issues in "regular use") is a secondary consideration. Functional correctness of most aspects (e.g., relative to low-level calculations, including units, scaling, quantities returned from external protocols) is generally most effectively done through thorough testing rather than human auditing. Importantly, thorough integration testing in the setting of final use is also an aspect that is not effectively covered by human auditing and remains the responsibility of the development team.

VULNERABILITIES & FUNCTIONAL ISSUES

This section details issues that affect the functionality of the contracts. Dedaub generally categorizes issues according to the following severities, but may also take other considerations into account such as impact or difficulty in exploitation:

Category	Description
CRITICAL	Can be profitably exploited by any knowledgeable third party attacker to drain a portion of the system's or users' funds OR the contract does not function as intended and severe loss of funds may result.
HIGH	Third party attackers or faulty functionality may block the system or cause the system or users to lose funds. Important system invariants can be violated.
MEDIUM	Examples: -User or system funds can be lost when third party systems misbehave. -DoS, under specific conditions. -Part of the functionality becomes unusable due to programming error.

LOW	Examples: -Breaking important system invariants, but without apparent consequences. -Buggy functionality for trusted users where a workaround exists. -Security issues which may manifest when the system evolves.

Issue resolution includes “dismissed” or “acknowledged” but no action taken, by the client, or “resolved”, per the auditors.

CRITICAL SEVERITY:

ID	Description	
C1	Malicious owner could change `priceFactor` and drain the protocol	**RESOLVED**

The price of each token is computed in the `getTokenPrice` function as a difference of two 3rd degree polynomials. The final result is multiplied by the `_priceFactor`. This local variable equals either the `alpha` of the token (if the creator of the token has set it), or a standard value (the global variable `priceFactor`).

Suppose that the user has not set the alpha of a token. Suppose also that we execute a buyToken operation, followed by a sellToken operation, with no change of supply or demand in between, and with the priceFactor remaining constant. In this case, the user will simply get back the money originally invested. However, if the _priceFactor changes between a buy and a sell operation, the symmetry between buying and selling is broken.

A malicious owner could drain the protocol, manipulating the price of the tokens through the priceFactor, as follows:

- He sets a low, or even zero, `priceFactor.`
- Creates a new token without setting its alpha (therefore the priceFactor will be used for the computations of the token price).
- Buys a huge amount of this new token (the price will be extremely low or even zero).
- Increase the priceFactor.
- Sell all his tokens at the new increased value. The amount of native tokens (NaT) he will get back suspases his initial deposit and if the increase in the priceFactor was significant he could get back even the total amount of NaT paid to the protocol for all the tokens.

We suggest either making the priceFactor immutable or storing the `priceFactor` for all the tokens with no set alpha, at their creation and using this for all the price computations.

C2	Users can take advantage of changes of the `priceFactor` and drain the protocol	**RESOLVED**

The impact of an increase in the priceFactor described above can also be taken advantage of by a malicious user as follows:

- Frontrun an increase of the `priceFactor` and create a new token, without setting its `alpha`, and buy a large amount x of token using y native tokens.
- The owner increases the `priceFactor.`
- The attacker sells all his tokens (except of the last one, which is not allowed by the protocol) at a higher price (since the `priceFactor` has increased) and therefore should get back an amount of native tokens greater than y i.e. he gets also funds deposited for other tokens. If x was sufficiently large, the attacker can drain the protocol.

Even if no one manages to frontrun a priceFactor change, a change in priceFactor is problematic per se. All the tokens bought before the change will have a different price after and therefore any user willing to sell his tokens will also get funds corresponding to other tokens.

We suggest either making the priceFactor immutable or storing the `priceFactor` for all the tokens with no set alpha, at their creation and using this for all the price computations.

HIGH SEVERITY:

ID	Description	STATUS
H1	Requirement that referrer be already invested in the protocol can be bypassed	**WON'T FIX**

The addReferrer function currently requires a referrer to be invested in the protocol by having created a token with a certain supply. However a new user can easily circumvent this process by creating a new referrer account with just one token (which costs nothing) and then executing a buy or sell operation using that referrer account, pocketing the referrer fee himself.

MEDIUM SEVERITY:

ID	Description	STATUS
M1	Fee distribution logic can result in DoS	**WON'T FIX**

The functions `buyToken` and `sellToken` perform a number of call operations to transfer native tokens for the purpose of fee distribution.

If the address on the receiving end of the call is a contract, then this will trigger its fallback function. There can be any code inside such a fallback function, including malicious code.

One way this could be exploited by an attacker, is for the attacker to create a malicious referer contract. Once this contract has been registered as the referrer of a token, it will simply revert each time its fallback function is triggered. This will cause every buy and sell operation on that token to fail, causing a denial of service for that token.

Due to the nature of the contract we could not find another way of exploiting the call operations. However one would do well to keep this problem in mind when the contract is expanded in the future, especially when it comes to reentrancy attacks.

In general, when funds need to be disbursed by a contract, the safest pattern is to have users withdraw funds, instead of sending funds to them, as this avoids the problem with transfer of control. However this would require a more sophisticated accounting system than the one currently employed by the contract.

M2	Missing frontrunning protection in `sellToken`	**FIXED**

When a user wants to buy or sell tokens, he only specifies the amount of tokens and the protocol computes the amount of native tokens he should pay or get back. The user cannot explicitly set an acceptable maximum (when he is buying tokens) or minimum (when he is selling) amount of native tokens. He can only check the current price of the tokens using the `getTokenBuy(Sell)PriceAfterFee` functions, but the price he will get and the actual price he will pay when we will execute the action could be different

if the state of the contract has changed in between i.e. if the current supply of the tokens has changed.

In `buyToken` there is an (indirect) protection of the users. There, the user does not only provide the amount of tokens he wants to buy, but also sends the corresponding amount of native tokens. If the price has increased, the transaction will revert.

In `sellToken` there is no protection and a user trying to sell his tokens could get back less than the expected amount of native tokens. Moreover, an attacker could frontrun a sellToken action, selling his tokens before the victim, decreasing the price of the token.

We suggest adding an extra variable minAmountOut in `sellToken` to mitigate this problem, allowing the user to specify a minimum sale price and to revert if this is not achieved.

M3	Rounding errors could be exploited by users	**WON'T FIX**

Even though the formulas for the computation of the price and the fees are additive i.e. the total amount a user should pay and the fees do not change if the user splits the transaction into smaller ones, this holds only in theory (infinite accuracy). Rounding errors break this additive nature of the formulas. Therefore the total native tokens a user should pay and the corresponding total protocol fees are less if the user buys x tokens in several steps e.g. x buys of 1 token, compared to a single buy of x tokens. Users could exploit this issue if they buy several tokens one at the time and then sell them all together.

An attack of this kind is probably not economically feasible, given that the user should pay extra transaction fees to execute these extra small transactions, but extra care and extensive simulations to identify the possibility of such an attack are needed.

A common mitigation for this type of issue is to set a minimum amount a user should be allowed to buy or sell.

LOW SEVERITY:

ID	Description	STATUS
L1	platformFeeDestination should be a constructor parameter	**WON'T FIX**
At the moment, the platformFeeDestination is set directly in the constructor, and there is a comment indicating that this assignment will be removed in the future. If this occurs, the result would be problematic. For instance, if a transaction takes place before the platformFeeDestination is set, the fees will be sent to address(0). We suggest that platformFeeDestination is provided as a parameter to the constructor.		
L2	Inconsistency between prices returned by getTokenSellPriceAfterFee and actual final price charged by sellToken	**FIXED**
When selling tokens, the getTokenSellPriceAfterFee function charges a trader fee, but when sellToken is called, no trader fees are deducted.		

CENTRALIZATION ISSUES:

It is often desirable for DeFi protocols to assume no trust in a central authority, including the protocol's owner. Even if the owner is reputable, users are more likely to engage with a protocol that guarantees no catastrophic failure even in the case the owner gets hacked/compromised. We list issues of this kind below. (These issues should be

considered in the context of usage/deployment, as they are not uncommon. Several high-profile, high-value protocols have significant centralization threats.)

ID	Description	
N1	The owner can unrestrictedly set crucial protocol parameters	**PARTIALLY FIXED**

The owner of the protocol can set (and change) several critical parameters: the fee factors/percentages, and the priceFactor. All these parameters affect the buy and sell prices of the tokens (check for example C1). There are no restrictions posed by the contract for the values of these parameters.

A malicious owner could, for instance, change the protocol fee to 100%.

Alternatively, a user who bought tokens under a given fee structure, coud have the fee change without warning before he can sell the tokens back.

In relation to the `priceFactor`, except for the more serious issues related to an increase of its price, a decrease could be also problematic, breaking the invariant that the `alpha` of each token is less than or equal to the `priceFractor`.

We suggest either making these parameters immutable or at least adding restrictions/bounds on their possible values, when they are getting set in the `setFee` function.

The owner can no longer change the priceFactor, but is able to change the protocol fees at any time.

OTHER/ ADVISORY ISSUES:

This section details issues that are not thought to directly affect the functionality of the project, but we recommend considering them.

ID	Description	STATUS
A1	Typo in getTokenPrice function parameter	**FIXED**
The getTokenPrice function has a parameter called currrentSupply instead of currentSupply (3 r's).		
A2	Magic numbers in contract	**FIXED**
The contract makes use of a number of magic numbers such as 16000 and 10 when calculating the token's price. We suggest that these be declared as constants with an appropriate name to enhance the transparency of the protocol.		
A3	Code duplication	**PARTIALLY FIXED**
There are instances of code duplication in the code base which can be refactored out. For instance, the computation of the fees in getTokenBuyPriceAfterFee and getTokenSellPriceAfterFee are identical. There is also duplicated code relating to fee transfers between buyToken and sellToken. There is also duplicate code within buyToken and sellToken itself, between the if and else branches relating to fee transfers. We recommend that these be factored out to help in the future maintainability of the protocol, so that changes only need to happen in one place. --- Duplicated code related to fee computations was refactored.		
A4	Redundant fee charges and refunds	**WON'T FIX**

In buyToken, the user sends his trader fee as part of the call, and is then refunded the same amount at the end of the transaction. Similarly, when a user creates a token when calling buyToken, he first gets charged a creator fee, which is then refunded again at the end of the transaction.

After consulting with the team, we understand that this is part of the protocol's marketing. However the fees could simply be computed and corresponding events emitted, avoiding the actual transfer of funds, if so wished.

A5	Missing cases and sanity checks	**FIXED**

In `sellToken` the requirement that the amount is positive should be added, similarly to what is done in `buyToken`.

In `getTokenPrice` the variable `s2` should be directly assigned the 0 value not only whenever `currentSupply == 0` but also when it equals 1 (for gas savings).

A6	Add reentrancy guard in `setAlpha`	**FIXED**

We have thoroughly investigated the possibility of altering the `alpha` of a token after its creation and we were not able to find any possible way to do so, but as an extra measure, and as a defense to possible future changes of the code of the contract, we suggest also adding a reentrancy guard in `setAlpha`.

A7	Separate the funds of each token	**WON'T FIX**

Users can buy tokens by sending native tokens (NaT) to the protocol. The protocol does not distinguish between the funds invested in different tokens. Even though each individual user is not guaranteed that he will get back, when he is selling, the exact amount he had paid when he bought the token (the price could have changed due to changes in supply and demand), the total NaT paid for each type of token can be collectively withdrawn (minus the fees) and it should not be possible to withdraw

funds associated to holders of other tokens. This is an important invariant of the protocol.

Under normal circumstances this invariant seems to hold (check C1 for exceptions), but as an extra safety and a good practice we suggest explicitly storing, not only the current supply of each token, but also the total amount of NaT deposited for that token and checking that when a user sells his tokens, he does not get back more than the total deposited NaT corresponding to the type of token he is holding.

A8	The getTokenSellPrice function does not check if amount is greater than supply	**FIXED**

The getTokenSellPrice function checks whether the supply is non-zero before returning the price, but does not check whether the amount is greater than the supply, thus returning a wrong price in this situation.

```
function getTokenSellPrice
       address tokenAddress,
       uint256 amount
       ) internal view returns (uint256) {
       uint256 _supply = tokensSupply[tokenAddress].currentSupply;
       uint256 _priceFactor = tokensSupply[tokenAddress].alpha == 0
              ? priceFactor
              : tokensSupply[tokenAddress].alpha;
       return
              _supply == 0
              ? 0
              : getTokenPrice(_supply - amount, amount, _priceFactor);
       }
```

We recommend altering the last check before the return in the following way.

```
function getTokenSellPrice
       address tokenAddress,
       uint256 amount
       ) internal view returns (uint256) {
       uint256 _supply = tokensSupply[tokenAddress].currentSupply;
       uint256 _priceFactor = tokensSupply[tokenAddress].alpha == 0
              ? priceFactor
              : tokensSupply[tokenAddress].alpha;
       return
              _supply <= amount
              ? 0
              : getTokenPrice(_supply - amount, amount, _priceFactor);
       }
```

A9	The getTokenPrice function can underflow and revert	**FIXED**

The getTokenPrice function can underflow if both supply and amount are equal to zero. In this case it would be better if the function returned zero instead.

A10	Compiler bugs	**INFO**

The code is compiled with Solidity `0.8.19`. Version `0.8.19`, in particular, has some known bugs, which we do not believe affect the correctness of the contracts.

DISCLAIMER

The audited contracts have been analyzed using automated techniques and extensive human inspection in accordance with state-of-the-art practices as of the date of this report. The audit makes no statements or warranties on the security of the code. On its own, it cannot be considered a sufficient assessment of the correctness of the contract. While we have conducted an analysis to the best of our ability, it is our recommendation for high-value contracts to commission several independent audits, a public bug bounty program, as well as continuous security auditing and monitoring through Dedaub Watchdog.

ABOUT DEDAUB

Dedaub offers significant security expertise combined with cutting-edge program analysis technology to secure some of the most prominent protocols in DeFi. The founders, as well as Dedaub's auditors, have a strong academic research background together with a real-world hacker mentality to secure code. Protocol blockchain developers hire us for our foundational analysis tools and deep expertise in program analysis, reverse engineering, DeFi exploits, cryptography and financial mathematics.