Follows Smart Contract Final Audit Report

Executive Summary

Project Name	Follows
Project URL	
Overview	Follows is a social app allowing people to buy/sell access to their favourite celebrity.
Audit Scope	https://github.com/Folome-online/smart-contracts/tree/feature/v2/contracts/follow-wrapper
Contracts in Scope	Branch: Main Contracts:- - FollowWrapperERC20.sol - FollowWrapperFactory.sol
Commit Hash	`9524d72`
Language	Solidity
Blockchain	Ethereum
Method	Manual Review, Automated tools,Functional testing
Review 1	29th April – 11 May
Updated Code Received	
Fixed In	

Number of security issues per severity.

TYPE	HIGH	MEDIUM	LOW	INFORMATIONAL
OPEN	0	0	0	0
ACKNOWLEDGED	0	0	1	1
Partially Resolved	0	0	0	0
Resolved	1	2	2	2

Check Vulnerabilities

- Access Management
- Arbitrary write to storage
- Centralization of control
- Ether theft
- Improper or missing events
- Logical issues and flaws
- Arithmetic Computations Correctness
- Race conditions/front running
- SWC Registry
- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- Exception Disorder
- Gasless Send
- Use of tx.origin
- Malicious libraries
- Compiler version not fixed
- Address hardcoded
- Divide before multiply
- Integer overflow/underflow
- ERC’s conformance
- Dangerous strict equalities
- Tautology or contradiction
- Return values of low-level calls
- Missing Zero Address Validation
- Private modifier
- Revert/require functions
- Multiple Sends
- Unsafe type inference
- Implicit visibility level

Techniques and Methods

Throughout the audit of smart contracts, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments, match logic and expected behavior.
- Token distribution and calculations are as per the intended behavior mentioned.
- Implementation of ERC standards.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods, and tools were used to review all the smart contracts.

Structural Analysis
In this step, we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis
A static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis
Manual Analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption
In this step, we have checked the behavior of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms used for Audit
Foundry, Solhint, Slither, Solidity static analysis.

Types of Severity

Every issue in this report has been assigned to a severity level. There are four levels of severity, and each of them has been explained below.

High Severity Issues
A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we

recommend these issues be fixed before moving to a live environment.

Medium Severity Issues

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.

Low Severity Issues

Low-level severity issues can cause minor impact and are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

Informational

These are four severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the initial audit and have been successfully fixed.

Acknowledged

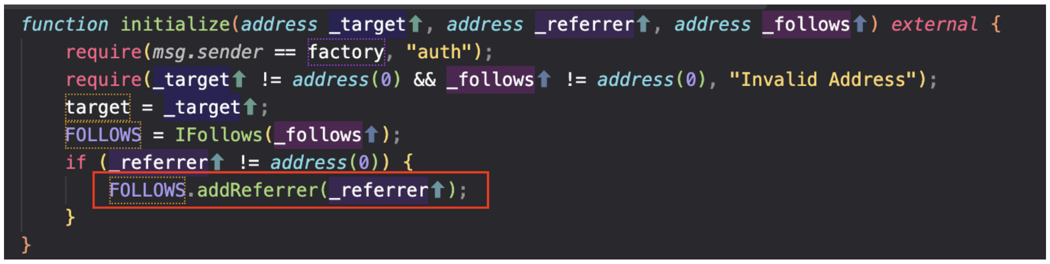
Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

High Severity Issues

A.1 Missing addReferrer(address) function in FollowV1

Line	Function - `initialize`
	

```
function initialize(address _target, address _referrer, address _follows) external {
    require(msg.sender == factory, "auth");
    require(_target != address(0) && _follows != address(0), "Invalid Address");
    target = _target;
    FOLLOWS = IFollows(_follows);
    if (_referrer != address(0)) {
        FOLLOWS.addReferrer(_referrer);
    }
}
```

Description:
The FollowsWrapperERC20 uses addReferrer() of the IFollows interface in the initialize function. However the there no such function exposed publically/externally in the FollowsV1 smart contract. This leads to EvmRevert error.
Remediation:
To address this issue create an addReferrer(address) function in the FollowsV1 contract.
Status: Resolved

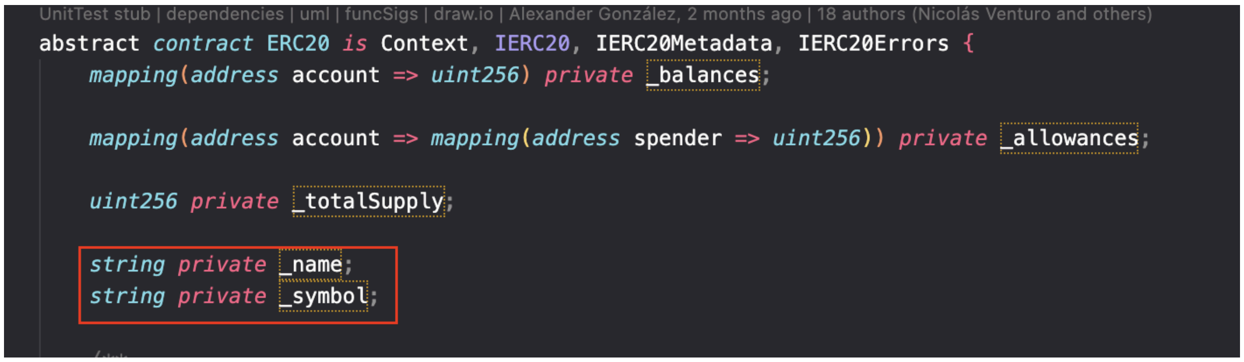
Medium Severity Issues

A.2 setNameAndSymbol(string,string) function is incompatible with latest openzeppelin inplementation

Line	Function - `setNameAndSymbol`

	`function setNameAndSymbol(string memory __name, string memory __symbol) external {`
	`        require (getFriend[msg.sender] != address(0), "FollowWrapperFactory: token not found");`
	
	`FollowWrapperERC20(payable(getFriend[msg.sender])).setNameAndSymbol(__name, __symbol);`
	`    }`

Description:
In the latest implementation of openzeppelin's ERC20 token contract, the __name and __symbol variables are made private. As a result, they can be set only during contract creation.



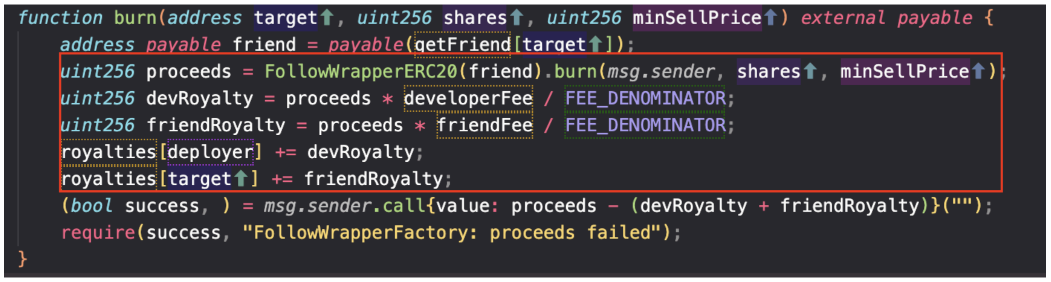
```
UnitTest stub | dependencies | uml | funcSigs | draw.io | Alexander González, 2 months ago | 18 authors (Nicolás Venturo and others)
abstract contract ERC20 is Context, IERC20, IERC20Metadata, IERC20Errors {
    mapping(address account => uint256) private _balances;

    mapping(address account => mapping(address spender => uint256)) private _allowances;

    uint256 private _totalSupply;

    string private _name;
    string private _symbol;
```

Remediation:

1. It is advised to pass name and symbol as parameters during contract creation of FollowsWrapperERC20.

Status: Resolved

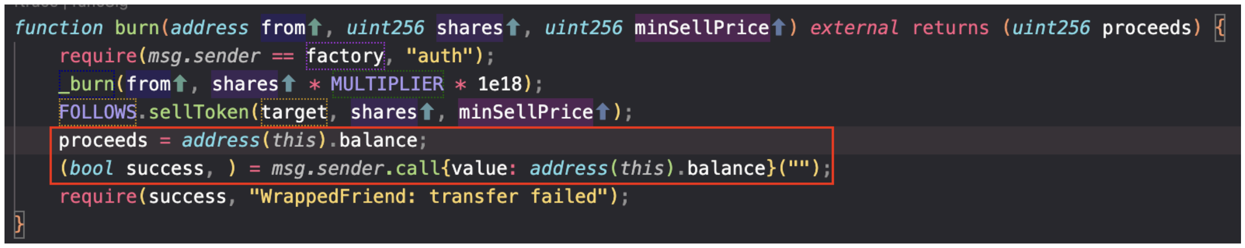
A.3 Incorrect accounting of royalties in the FollowsWrapperFactory in the burn() function

Line	**Function -** `burn`
7	

```
function burn(address target, uint256 shares, uint256 minSellPrice) external payable {
    address payable friend = payable(getFriend[target]);
    uint256 proceeds = FollowWrapperERC20(friend).burn(msg.sender, shares, minSellPrice);
    uint256 devRoyalty = proceeds * developerFee / FEE_DENOMINATOR;
    uint256 friendRoyalty = proceeds * friendFee / FEE_DENOMINATOR;
    royalties[deployer] += devRoyalty;
    royalties[target] += friendRoyalty;
    (bool success, ) = msg.sender.call{value: proceeds - (devRoyalty + friendRoyalty)}("");
    require(success, "FollowWrapperFactory: proceeds failed");
}
```

Description:

Royalties are calculated based on the proceeds sent from the FollowsWrapperERC20.sol contract.

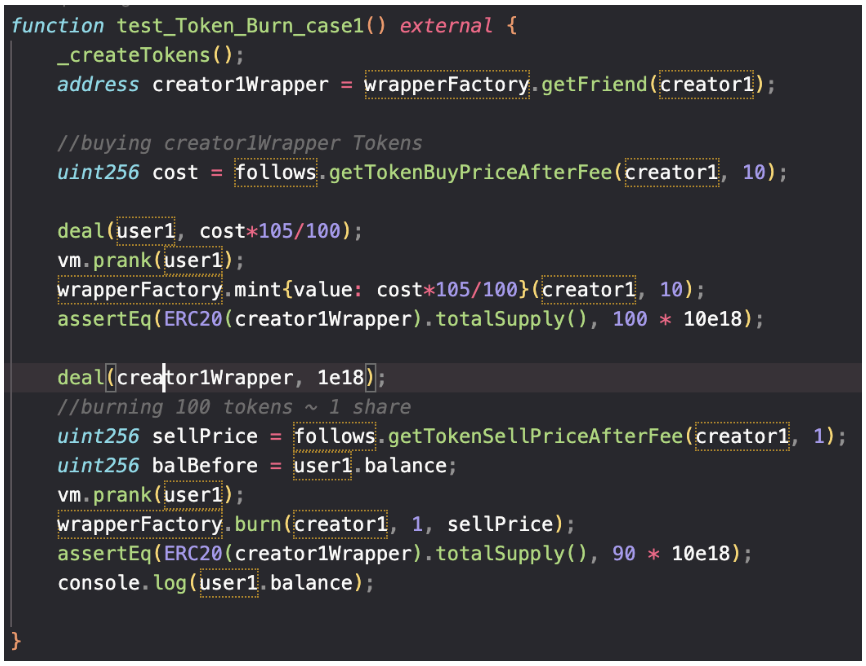


```
function burn(address from, uint256 shares, uint256 minSellPrice) external returns (uint256 proceeds) {
    require(msg.sender == factory, "auth");
    _burn(from, shares * MULTIPLIER * 1e18);
    FOLLOWS.sellToken(target, shares, minSellPrice);
    proceeds = address(this).balance;
    (bool success, ) = msg.sender.call{value: address(this).balance}("");
    require(success, "WrappedFriend: transfer failed");
}
```

However in the FollowsWrapperERC20.sol, the proceeds during burn(..) are accounted incorrectly, rather can accounting for balance before and after sellToken, it takes the total Ether balance of the contract as proceeds.

One possible attack vector can be to send any amount of ether to the contract, which will result in bloated proceeds value, affecting the royalties accounting.

Example:



```
function test_Token_Burn_case1() external {
    _createTokens();
    address creator1Wrapper = wrapperFactory.getFriend(creator1);

    //buying creator1Wrapper Tokens
    uint256 cost = follows.getTokenBuyPriceAfterFee(creator1, 10);

    deal(user1, cost*105/100);
    vm.prank(user1);
    wrapperFactory.mint{value: cost*105/100}(creator1, 10);
    assertEq(ERC20(creator1Wrapper).totalSupply(), 100 * 10e18);

    deal(creator1Wrapper, 1e18);
    //burning 100 tokens ~ 1 share
    uint256 sellPrice = follows.getTokenSellPriceAfterFee(creator1, 1);
    uint256 balBefore = user1.balance;
    vm.prank(user1);
    wrapperFactory.burn(creator1, 1, sellPrice);
    assertEq(ERC20(creator1Wrapper).totalSupply(), 90 * 10e18);
    console.log(user1.balance);

}
```

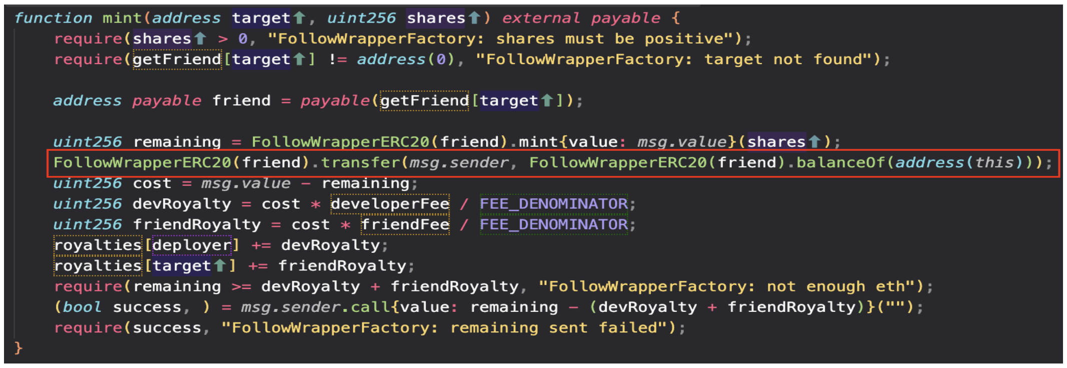
Remediation:

1. It is advised to take balance of contract before and after calling the sellToken(..) function, and then accounting the proceeds.

Status: Resolved

Low Severity Issues

A.4 Incorrect tokens sent during mint() in the FollowsWrapperFactory.sol

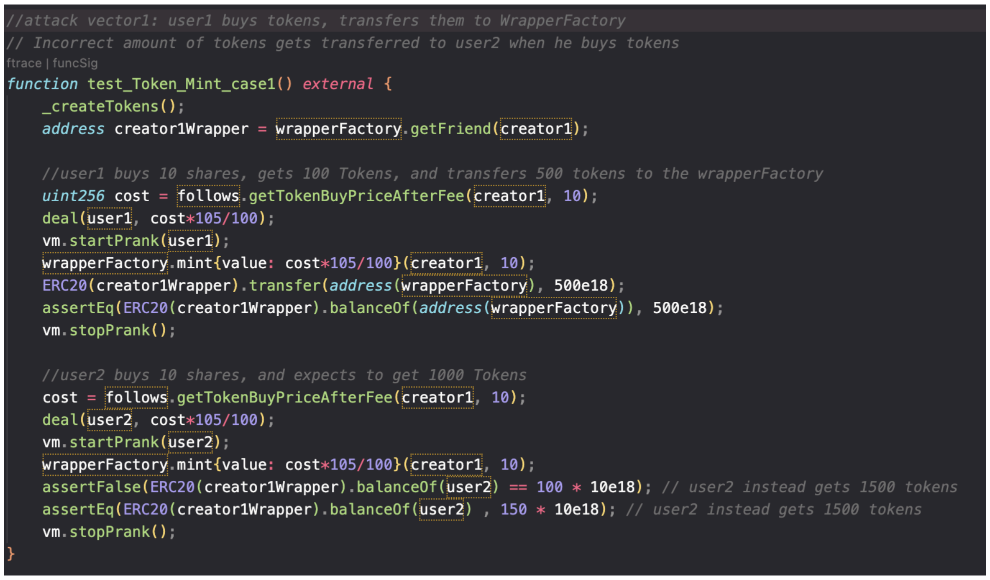
Line	Function - mint(..)
	

```
function mint(address target, uint256 shares) external payable {
    require(shares > 0, "FollowWrapperFactory: shares must be positive");
    require(getFriend[target] != address(0), "FollowWrapperFactory: target not found");

    address payable friend = payable(getFriend[target]);

    uint256 remaining = FollowWrapperERC20(friend).mint{value: msg.value}(shares);
    FollowWrapperERC20(friend).transfer(msg.sender, FollowWrapperERC20(friend).balanceOf(address(this)));
    uint256 cost = msg.value - remaining;
    uint256 devRoyalty = cost * developerFee / FEE_DENOMINATOR;
    uint256 friendRoyalty = cost * friendFee / FEE_DENOMINATOR;
    royalties[deployer] += devRoyalty;
    royalties[target] += friendRoyalty;
    require(remaining >= devRoyalty + friendRoyalty, "FollowWrapperFactory: not enough eth");
    (bool success, ) = msg.sender.call{value: remaining - (devRoyalty + friendRoyalty)}("");
    require(success, "FollowWrapperFactory: remaining sent failed");
}
```

Description:

In the mint function in the FollowsWrapperFactory.sol smart contract, the final amount of tokens transferred to the user can be incorrect. The transfer function takes the total balance of the factory contract. A simple case where someone transfers tokens to the contract, can lead to a higher amount of tokens being received during a mint.

Example:



```
//attack vector1: user1 buys tokens, transfers them to WrapperFactory
// Incorrect amount of tokens gets transferred to user2 when he buys tokens
ftrace | funcSig
function test_Token_Mint_case1() external {
    _createTokens();
    address creator1Wrapper = wrapperFactory.getFriend(creator1);

    //user1 buys 10 shares, gets 100 Tokens, and transfers 500 tokens to the wrapperFactory
    uint256 cost = follows.getTokenBuyPriceAfterFee(creator1, 10);
    deal(user1, cost*105/100);
    vm.startPrank(user1);
    wrapperFactory.mint{value: cost*105/100}(creator1, 10);
    ERC20(creator1Wrapper).transfer(address(wrapperFactory), 500e18);
    assertEq(ERC20(creator1Wrapper).balanceOf(address(wrapperFactory)), 500e18);
    vm.stopPrank();

    //user2 buys 10 shares, and expects to get 1000 Tokens
    cost = follows.getTokenBuyPriceAfterFee(creator1, 10);
    deal(user2, cost*105/100);
    vm.startPrank(user2);
    wrapperFactory.mint{value: cost*105/100}(creator1, 10);
    assertFalse(ERC20(creator1Wrapper).balanceOf(user2) == 100 * 10e18); // user2 instead gets 1500 tokens
    assertEq(ERC20(creator1Wrapper).balanceOf(user2) , 150 * 10e18); // user2 instead gets 1500 tokens
    vm.stopPrank();
}
```

Remediation:

1. It is advised to take balance of contract before and after calling the mint(..) function of WrapperERC20, and then calculating the correct amount of tokens to be transferred.

Status: Acknowledged

A.5 Missing zerochecks in the WrapperFactory and WrapperERC20 contract

Description:
There are missing zerochecks of the parameter addresses.
FollowsWrapperFactory.sol

1. function burn(address target, uint256 shares, uint256 minSellPrice)
2. function createToken(address _referrer):

FollowsWrapperERC20.sol

1. function initialize(address _target, address _referrer): missing

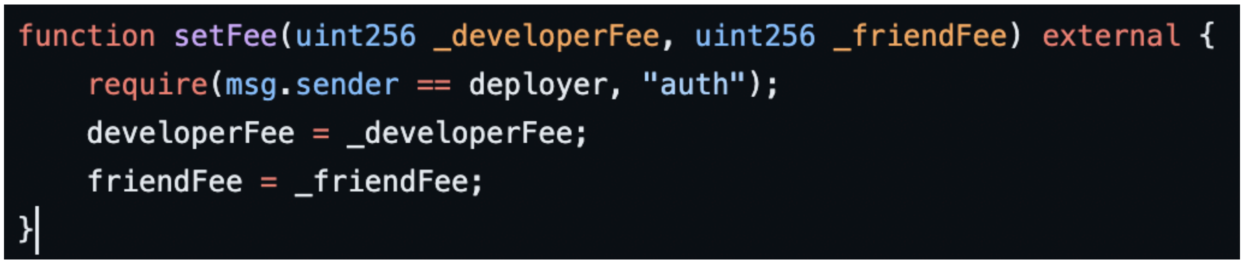
Remediation:

3. It is advised to add zero address checks to make sure incorrect address gets reverted.

Status: Resolved

A.6 No upper threshold limit for fees in setFee(..) function

Description:
In the setFee(..) function of the FollowsWrapperFactory.sol smart contract, no upper limit of the _developerFee and _friendFee is provided. This can lead to them being set 50, 50 %, which will result in royalties ~100% of the cost of the mint.



```
function setFee(uint256 _developerFee, uint256 _friendFee) external {
    require(msg.sender == deployer, "auth");
    developerFee = _developerFee;
    friendFee = _friendFee;
}
```

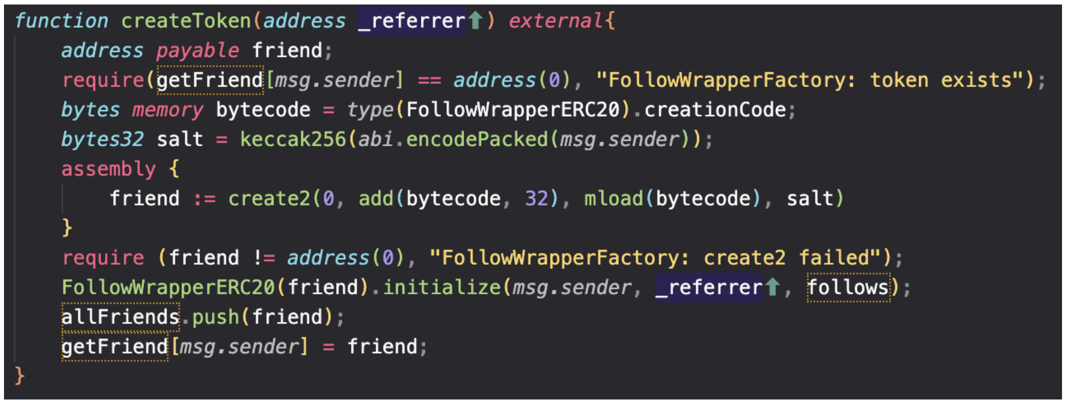
Remediation:

1. It is advised to a require check in to setFee(..) function.

Status: Resolved

Informational Issues

A.7: The createToken(...) function should return the Wrapper address

Line	Function - createToken(..)
	

```
function createToken(address _referrer) external{
    address payable friend;
    require(getFriend[msg.sender] == address(0), "FollowWrapperFactory: token exists");
    bytes memory bytecode = type(FollowWrapperERC20).creationCode;
    bytes32 salt = keccak256(abi.encodePacked(msg.sender));
    assembly {
        friend := create2(0, add(bytecode, 32), mload(bytecode), salt)
    }
    require (friend != address(0), "FollowWrapperFactory: create2 failed");
    FollowWrapperERC20(friend).initialize(msg.sender, _referrer, follows);
    allFriends.push(friend);
    getFriend[msg.sender] = friend;
}
```

Description - The createToken(...) function should return the address of the ERC20 wrapper created.

Status: **Resolved**

A.8: Missing getter functions

There are some missing getter functions:

1. Cost of minting wrapper tokens from the FollowWrapperFactory
2. The expected ether received after burning wrapper tokens from the FollowWrapperFactory

Status: **Acknowledged**

A.9: Missing events

There are events missing in the 2 smart contracts.
FollowWrapperFactory:

1. createToken
2. Mint
3. Burn

4. setFee
5. claimRoyalty

FollowWrapperERC20

1. initialize

Status: Resolved

Functional Tests

Some of the tests performed are mentioned below:

✅ Should be able to deploy the factory and followsV1

✅ Should set the parameters of the followsV1

✅ Should set fee in the factory

✅ Should create token

✅ Should mint

✅ Should burn

✅ Should perform the mentioned attack vector in the mint(..) function

✅ Should perform the mentioned attack vector in the burn(..) function

Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Closing Summary

In this report, we have considered the security of the Follows wrapper and factory contracts. We performed the audit according to the procedure described above.

Some issues of High, Medium, Low and informational severity were found, Some suggestions and best practices are also provided in order to improve the code quality and security posture.