SMART CONTRACT AUDIT REPORT

for

SUPERFLUID

Prepared By: Shuxiao Wang

PeckShield

February 9, 2021

Document Properties

Client	Superfluid
Title	Smart Contract Audit Report
Target	Superfluid
Version	1.0
Author	Xuxian Jiang
Auditors	Huaguo Shi, Xuxian Jiang
Reviewed by	Shuxiao Wang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	February 9, 2021	Xuxian Jiang	Final Release
1.0-rc2	February 2, 2021	Xuxian Jiang	Release Candidate #2
1.0-rc1	January 31, 2021	Xuxian Jiang	Release Candidate #1
0.2	January 25, 2021	Xuxian Jiang	Additional Findings
0.1	January 18, 2021	Xuxian Jiang	Initial Draft

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the Superfluid protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Superfluid

Superfluid is a smart contract framework on layer 1 Ethereum, enabling you to move assets on-chain following predefined rules called agreements. With a single on-chain transaction, the money will flow from your balance to the receiver in real time. This framework is designed to be flexible and aims to empower various scenarios, including constant flows on-chain with no capital lockups as well as fixed cost distribution in a single transaction for any number of receivers. Superfluid has a set of white-listed agreements contracts as building blocks, and provides a development framework for building real-time finance applications.

The basic information of the Superfluid protocol is as follows:

Table 1.1: Basic Information of The Superfluid Protocol

Item	Description
Issuer	Superfluid
Website	https://superfluid.finance/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	February 9, 2021

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/superfluid-finance/protocol-monorepo (e31d135)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/superfluid-finance/protocol-monorepo (88c9725)

1.2 About PeckShield

PeckShield Inc. [11] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [10]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.

- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.

- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [9], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the Superfluid implementation. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	1	■
Low	5	■■■■■
Informational	2	■■
Total	8	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities that need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 medium-severity vulnerability, 5 low-severity vulnerabilities, and and 2 informational recommendations.

Table 2.1: Key Superfluid Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Safe-Version Replacement With safeTransfer() And safeTransferFrom()	Coding Practices	Fixed
PVE-002	Low	Accommodation of approve() Idiosyncrasies	Business Logic	Fixed
PVE-003	Low	Incompatibility with Deflationary/Rebasing Token	Business Logic	Fixed
PVE-004	Low	Potential Front-Running DOS For updateSuperTokenFactory()	Security Features	Confirmed
PVE-005	Informational	Improved Superfluid App Registration	Business Logic	Fixed
PVE-006	Low	Permission-Less Event Generation	Time and State	Confirmed
PVE-007	Low	Inaccurate rewardAmount In AgreementLiquidated Events	Business Logic	Fixed
PVE-008	Informational	RoundingDown Consistency Of appAllowance in _changeFlow()	Coding Practices	Fixed

Besides recommending specific countermeasures to mitigate these issues, based on the fact that compiler upgrades might bring unexpected compatibility or inter-version consistencies, it is always suggested to use fixed compiler versions whenever possible. As an example, we highly encourage to explicitly indicate the Solidity compiler version, e.g., `pragma solidity 0.6.11` instead of specifying a range, e.g., `pragma solidity ^0.6.11`.

In addition, we also emphasize that it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Safe-Version Replacement With safeTransfer() And safeTransferFrom()

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: SuperToken
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `transfer()` routine and possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular token, i.e., `ZRX`, as our example. We show the related code snippet below. On its entry of `transfer()`, there is a check, i.e., `if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to])`. If the check fails, it returns `false`. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: *"Transfers _value amount of tokens to address _to, and MUST fire the Transfer event. The function SHOULD throw if the message caller's account balance does not have enough tokens to spend."*

```
function transfer(address _to, uint _value) returns (bool) {
    //Default assumes totalSupply can't be over max (2^256 - 1).
    if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
        balances[msg.sender] -= _value;
        balances[_to] += _value;
        Transfer(msg.sender, _to, _value);
        return true;
    } else { return false; }
}
```

```
function transferFrom(address _from, address _to, uint _value) returns (bool) {
    if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
        balances[_to] + _value >= balances[_to]) {
        balances[_to] += _value;
        balances[_from] -= _value;
        allowed[_from][msg.sender] -= _value;
        Transfer(_from, _to, _value);
        return true;
    } else { return false; }
}
```

Listing 3.1: ZRX.sol

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. To use this library you can add a using SafeERC20 for IERC20. Similarly, there is a safe version of `transferFrom()` as well, i.e., `safeTransferFrom()`.

In the following, we show the `_upgrade/_downgrade()` routines in the `SuperToken` contract. If the `USDT` token is supported as `underlyingToken`, the unsafe version of `_underlyingToken.transferFrom(account, address(this), underlyingAmount)` (line 547) may revert as there is no return value in the `USDT` token contract's `transferFrom()` implementation (but the `IERC20` interface expects a return value)!

```
537     function _upgrade(
538         address operator,
539         address account,
540         address to,
541         uint256 amount,
542         bytes memory userData,
543         bytes memory operatorData
544     ) private {
545         require(address(_underlyingToken) != address(0), "SuperToken: no underlying
            token");
546         (uint256 underlyingAmount, uint256 actualAmount) = _toUnderlyingAmount(amount);
547         _underlyingToken.transferFrom(account, address(this), underlyingAmount);
548         _mint(operator, to, actualAmount,
549             // if `to` is diffferent from `account`, we requireReceptionAck
550             account != to, userData, operatorData);
551         emit TokenUpgraded(account, actualAmount);
552     }

554     function _downgrade(
555         address operator,
556         address account,
557         uint256 amount,
558         bytes memory data,
559         bytes memory operatorData) private {
```

```
        require(address(_underlyingToken) != address(0), "SuperToken: no underlying
            token");
        // - in case of downcasting of decimals, actual amount can be smaller than
            requested amount
        (uint256 underlyingAmount, uint256 actualAmount) = _toUnderlyingAmount(amount);
         // _burn will check the (actual) amount availability again
        _burn(operator, account, actualAmount, data, operatorData);
        _underlyingToken.transfer(account, underlyingAmount);
        emit TokenDowngraded(account, actualAmount);
    }
```

Listing 3.2: SuperToken::_upgrade()/_downgrade()

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `transfer()` and `transferFrom()`.

Status The issue has been fixed by this commit: `77d93f4`.

3.2 Accommodation of approve() Idiosyncrasies

- ID: PVE-002
- Severity: Low
- Likelihood: medium
- Impact: Low
- Target: `SuperUpgrader`
- Category: Business Logic [7]
- CWE subcategory: N/A

Description

In Section 3.1, we have examined certain non-compliant ERC20 tokens that may exhibit specific idiosyncrasies in their `transfer()` and `transferFrom()` implementations. In this section, we examine the `approve()` routine and possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below. On its entry of `approve()`, there is a requirement, i.e., `require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()/transferFrom()` race condition (https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729).

```
    /**
    * @dev Approve the passed address to spend the specified amount of tokens on behalf
        of msg.sender.
    * @param _spender The address which will spend the funds.
    * @param _value The amount of tokens to be spent.
    */
```

```
    function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

        // To change the approve amount you first have to reduce the addresses`
        //  allowance to zero by calling `approve(_spender, 0)` if it is not
        //  already 0 to mitigate the race condition described here:
        //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
        require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

        allowed[msg.sender][_spender] = _value;
        Approval(msg.sender, _spender, _value);
    }
```

Listing 3.3: USDT Token Contract

Because of that, a normal call to `approve()` with a currently non-zero allowance may fail. In the following, we use as an example the `SuperUpgrader` contract that is designed to upgrade certain tokens to be `Superfluid`-friendly. To accommodate the specific idiosyncrasy, there is a need to `approve()` twice: the first one reduces the allowance to 0; and the second one sets the new allowance.

```
    /**
     * @notice The user should ERC20.approve this contract.
     * @dev Execute upgrade function in the name of the user
     * @param superTokenAddr Super Token Address to upgrade
     * @param account User address that previous approved this contract.
     * @param amount Amount value to be upgraded.
     */
    function upgrade(
        address superTokenAddr,
        address account,
        uint256 amount
    )
    external
    {
        require(msg.sender == account
            (hasRole(BACKEND_ROLE, msg.sender) &&
            !_optout[account])
        , "operation not allowed");
        //get underlaying token
        ISuperToken superToken = ISuperToken(superTokenAddr);
        //get tokens from user
        IERC20 token = IERC20(superToken.getUnderlyingToken());
        token.transferFrom(account, address(this), amount);
        token.approve(address(superToken), amount);
        //upgrade tokens and send back to user
        superToken.upgradeTo(account, amount, "");
    }
```

Listing 3.4: SuperUpgrader::upgrade()

Recommendation Accommodate the above-mentioned idiosyncrasy of `approve()`.

Status The issue has been elaborated and addressed by the following tracking number: `208`.

3.3 Incompatibility with Deflationary/Rebasing Tokens

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [5]

Description

In Superfluid, the `SuperUpgrader` contract is designed to be the main entry for users who want to delegate the token-upgrade task to certain accounts who have the so-called `BACKEND_ROLE`. In particular, one entry routine, i.e., `upgrade()`, accepts user deposits of supported assets (e.g., `DAI`). Naturally, the contract implements a number of low-level helper routines to transfer assets in or out of the protocol. These asset-transferring routines work as expected with standard ERC20 tokens: namely the vault's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract.

```
    /**
     * @notice The user should ERC20.approve this contract.
     * @dev Execute upgrade function in the name of the user
     * @param superTokenAddr Super Token Address to upgrade
     * @param account User address that previous approved this contract.
     * @param amount Amount value to be upgraded.
     */
    function upgrade(
        address superTokenAddr,
        address account,
        uint256 amount
    )
    external
    {
        require(msg.sender == account
            (hasRole(BACKEND_ROLE, msg.sender) &&
            !_optout[account])
        , "operation not allowed");
        //get underlaying token
        ISuperToken superToken = ISuperToken(superTokenAddr);
        //get tokens from user
        IERC20 token = IERC20(superToken.getUnderlyingToken());
        token.transferFrom(account, address(this), amount);
        token.approve(address(superToken), amount);
        //upgrade tokens and send back to user
        superToken.upgradeTo(account, amount, "");
    }
```

Listing 3.5: SuperUpgrader::upgrade()

However, there exist other ERC20 tokens that may make certain customizations to their ERC20 contracts. One type of these tokens is deflationary tokens that charge a certain fee for every `transfer()` or `transferFrom()`. (Another type is rebasing tokens such as `YAM`.) As a result, this may not meet the assumption behind these low-level asset-transferring routines.

One possible mitigation is to regulate the set of ERC20 tokens that are permitted into the protocol. In our case, it is indeed possible to effectively regulate the set of tokens that can be supported. Keep in mind that there exist certain assets (e.g., `USDT`) that may have control switches that can be dynamically exercised to suddenly become one.

Recommendation If current codebase needs to support possible deflationary tokens, it is better to check the balance before and after the `transfer()/transferFrom()` call to ensure the book-keeping amount is accurate. This support may bring additional gas cost. Also, keep in mind that certain tokens may not be deflationary for the time being. However, they could have a control switch that can be exercised to turn them into deflationary tokens. One example is widely-adopted `USDT`.

Status The issue has been fixed by this commit: `2fc937b`.

3.4 Potential Front-Running DOS For updateSuperTokenFactory()

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Superfluid`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

In the Superfluid protocol, there is a central contract named `Superfluid Host`. This contract connects together white-listed `super agreements`, `super apps`, and `super tokens`. It is also the entry point for the protocol users, who can make batch-calls and submit meta-transactions.

In the following, we show the `updateSuperTokenFactory()` routine from the `Superfluid Host` contract. As the name indicates, this routine is designed to update the `super token factory` contract. We note the protocol supports two types of factory upgrade: `non-upgradeable` and `upgradeable`.

```
    function updateSuperTokenFactory(ISuperTokenFactory newFactory)
        external override
        onlyGovernance
    {
        if (address(_superTokenFactory) == address(0)) {
            if (!NON_UPGRADABLE_DEPLOYMENT) {
```

```
                // initialize the proxy
                UUPSProxy proxy = new UUPSProxy();
                proxy.initializeProxy(address(newFactory));
                _superTokenFactory = ISuperTokenFactory(address(proxy));
            } else {
                _superTokenFactory = newFactory;
            }
            _superTokenFactory.initialize();
        } else {
            require(!NON_UPGRADABLE_DEPLOYMENT, "SF: non upgradable");
            UUPSProxiable(address(_superTokenFactory)).updateCode(address(newFactory));
        }
    }
```

Listing 3.6: Superfluid :: updateSuperTokenFactory()

In either case, the new `super token factory` will need to be initialized when it is updated. To elaborate, we also show below the related routines when the factory contract is initialized.

```
50      function initialize()
51          external override
52          initializer // OpenZeppelin Initializable
53      {
54          _updateSuperTokenLogic();
55      }
56
57      function proxiableUUID() public pure override returns (bytes32) {
58          return keccak256("org.superfluid-finance.contracts.SuperTokenFactory.
                implementation");
59      }
60
61      function updateCode(address newAddress) external override {
62          require(msg.sender == address(_host), "only host can update code");
63          _updateCodeAddress(newAddress);
64          _updateSuperTokenLogic();
65      }
66
67      function _updateSuperTokenLogic() private {
68          // use external call to trigger the new code to update the super token logic
                contract
69          _superTokenLogic = SuperToken(this.createSuperTokenLogic(_host));
70          emit SuperTokenLogicCreated(_superTokenLogic);
71      }
```

Listing 3.7: Related SuperTokenFactoryBase Routines: initialize () and updateSuperTokenLogic()

The `initialize()` routine has an `initializer` modifier (line 52), inherited from `OpenZeppelin Initializable`. This modifier is proposed to ensure this routine can only be called once in a proxy-based deployment. However, it comes to our attention that this `initialize()` routine is permissionless, indicating that any one can invoke it to update the super token logic contract. While the update on the super token logic contract does not introduce malicious logic, the fact of being initialized via the `initializer` modifier blocks the legitimate call from the `Superfluid Host` contract, hence presenting a denial-of-service to governance-issued `updateSuperTokenFactory()` call.

Recommendation Revise the `initialize()` routine in `SuperTokenFactoryBase` to be permissioned. In fact, we can validate whether the caller is from the registered `host`. An example revision is shown below.

```
    function initialize()
        external override
        initializer // OpenZeppelin Initializable
    {
        require(msg.sender == address(_host), "only host can update code");
        _updateSuperTokenLogic();
    }
```

Listing 3.8: Related SuperTokenFactoryBase Routines: initialize () and updateSuperTokenLogic()

Status The issue has been confirmed. Considering that `NON_UPGRADABLE_DEPLOYMENT` is an experimental feature, and the risk can be mitigated by re-deploying a new one, and the reward of such grieving attack it is not clear neither, the team decides to leave as is.

3.5 Improved Superfluid App Registration

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Superfluid`
- Category: Business Logic [7]
- CWE subcategory: CWE-837 [4]

Description

As mentioned in Section 3.4, there is a central contract named `Superfluid Host` that connects together white-listed `agreements`, `apps`, and `Superfluid-friendly tokens`. To this end, it also provides helper routines to register `agreements` and `apps`. In the following, we examine the registration logic in the `Superfluid Host` contract.

To elaborate, we show below the code snippet of `registerApp()` who handles the registration of `Superfluid apps`. The logic is rather straightforward in validating the given `app` and its associated `configWord` and properly recording it in the internal array `_appManifests`.

```
    function registerApp(
        uint256 configWord
    )
        external override
    {
```

```
275         ISuperApp app = ISuperApp(msg.sender);
276         {
277             uint256 cs;
278             // solhint-disable-next-line no-inline-assembly
279             assembly { cs := extcodesize(app) }
280             require(cs == 0, "SF: app registration only in constructor");
281         }
282         require(
283             SuperAppDefinitions.getAppLevel(configWord) > 0 &&
284             (configWord & SuperAppDefinitions.APP_JAIL_BIT) == 0,
285             "SF: invalid config word");
286         require(_appManifests[ISuperApp(msg.sender)].configWord == 0 , "SF: app already
                registered");
287         _appManifests[ISuperApp(msg.sender)] = AppManifest(configWord);
288     }
```

Listing 3.9: Superfluid::registerApp()

However, we notice the validation of a `Superfluid app` needs to be performed in the app's constructor. The current method of discerning the constructor call is based on the `extcodesize(app)` (line 279). This may not be reliable as a regular EOA account can also be registered as a `Superfluid app`.

Recommendation Ensure the registered `Superfluid app` is a contract. An example revision is shown below.

```
270     function registerApp(
271         uint256 configWord
272     )
273         external override
274     {
275         ISuperApp app = ISuperApp(msg.sender);
276         {
277             uint256 cs;
278             // solhint-disable-next-line no-inline-assembly
279             assembly { cs := extcodesize(app) }
280             require(cs == 0 && msg.sender!= tx.origin , "SF: app registration only in
                    constructor");
281         }
282         require(
283             SuperAppDefinitions.getAppLevel(configWord) > 0 &&
284             (configWord & SuperAppDefinitions.APP_JAIL_BIT) == 0,
285             "SF: invalid config word");
286         require(_appManifests[ISuperApp(msg.sender)].configWord == 0 , "SF: app already
                registered");
287         _appManifests[ISuperApp(msg.sender)] = AppManifest(configWord);
288     }
```

Listing 3.10: Revised Superfluid::registerApp()

Status The issue has been fixed by this commit: `d9453d5`.

3.6 Permission-Less Event Generation

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact:Low
- Target: `SuperfluidToken`
- Category: Numeric Errors [8]
- CWE subcategory: CWE-190 [2]

Description

In order to take advantage of various benefits enabled by Superfluid, current ERC20 tokens need to be upgraded to be `Superfluid`-friendly. A `Superfluid`-friendly token has additional functionality in maintaining various `agreement-related` states and data. For example, the meta-data associated with each `agreement` and current account-specific state from each connected `agreement` are all saved in the token storage.

To elaborate, we show below various `setters` that update these states, including `updateAgreementData()`, `terminateAgreement()`, and `updateAgreementStateSlot()`.

```
    /// @dev ISuperfluidToken.updateAgreementData implementation
    function updateAgreementData(
        bytes32 id,
        bytes32[] calldata data
    )
        external override
    {
        address agreementClass = msg.sender;
        bytes32 slot = keccak256(abi.encode("AgreementData", agreementClass, id));
        FixedSizeData.storeData(slot, data);
        emit AgreementUpdated(msg.sender, id, data);
    }

    /// @dev ISuperfluidToken.terminateAgreement implementation
    function terminateAgreement(
        bytes32 id,
        uint dataLength
    )
        external override
    {
        address agreementClass = msg.sender;
        bytes32 slot = keccak256(abi.encode("AgreementData", agreementClass, id));
        require(FixedSizeData.hasData(slot,dataLength), "SuperfluidToken: agreement does
            not exist");
        FixedSizeData.eraseData(slot, dataLength);
        emit AgreementTerminated(msg.sender, id);
    }

    /// @dev ISuperfluidToken.updateAgreementState implementation
```

```
    function updateAgreementStateSlot(
        address account,
        uint256 slotId,
        bytes32[] calldata slotData
    )
        external override
    {
        bytes32 slot = keccak256(abi.encode("AgreementState", msg.sender, account,
            slotId));
        FixedSizeData.storeData(slot, slotData);
        // FIXME change how this is done
        //_addAgreementClass(msg.sender, account);
        emit AgreementStateUpdated(msg.sender, account, slotId);
    }
```

Listing 3.11: Various setters in SuperfluidToken

It comes to our attention that these `setters` are permission-less and they can be used to emit misleading events. This may cause unnecessary confusion to external analytics and reporting tools.

Recommendation Apply necessary restrictions on the caller when these `setters` update internal storage states and avoid firing unnecessary events.

Status The issue has been confirmed. Considering that these functions are permissionless for gas efficiency and the storage of agreements behind the token is to make `agreement` contracts to be pure logic contract. the team decides to leave as is.

3.7 Inaccurate rewardAmount In AgreementLiquidated Events

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `SuperfluidToken`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [5]

Description

In `Ethereum`, the `event` is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an `event` is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. `Events` can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

In the following, we use the `SuperfluidToken` contract as an example. This contract is designed to add necessary functionality to suit the `Superfluid` needs. we notice the emitted `AgreementLiquidated` event (line 368 – 373) contains incorrect information. Specifically, the event is defined as `event AgreementLiquidated(address indexed agreementClass, bytes32 id, address indexed penaltyAccount, address indexed rewardAccount, uint256 rewardAmount)` with a number of parameters: the first parameter `agreementClass` encodes the involved `agreement`; the second parameter represents the `Agreement ID`; the third parameter `penaltyAccount` shows the account to be penalized; the fourth parameter `rewardAccount` shows the account that collects the reward; while the last parameter `rewardAmount` indicates the amount of liquidation reward. The emitted event contains an incorrect `rewardAmount` information, which should not be `bailoutAmount`. Instead, it should be `rewardAmount`, the fourth function argument to `makeLiquidationPayouts()`.

```
320     /// @dev ISuperfluidToken.makeLiquidationPayouts implementation
321     function makeLiquidationPayouts
322     (
323         bytes32 id,
324         address liquidator,
325         address penaltyAccount,
326         uint256 rewardAmount,
327         uint256 bailoutAmount
328     )
329         external override
330         onlyAgreement
331     {
332         ISuperfluidGovernance gov = _host.getGovernance();
333         address rewardAccount = gov.getConfigAsAddress(_host, this,
                _REWARD_ADDRESS_CONFIG_KEY);
334         // reward go to liquidator if reward address is null
335         if (rewardAccount == address(0)) {
336             rewardAccount = liquidator;
337         }
338
339         int256 signedRewardAmount = rewardAmount.toInt256();
340
341         if (bailoutAmount == 0) {
342             // if account is in critical state
343             // - reward account takes the reward
344             _balances[rewardAccount] = _balances[rewardAccount]
345                 .add(signedRewardAmount);
346             // - penalty applies
347             _balances[penaltyAccount] = _balances[penaltyAccount]
348                 .sub(signedRewardAmount);
349             emit AgreementLiquidated(
350                 msg.sender, id,
351                 penaltyAccount,
352                 rewardAccount /* rewardAccount */,
353                 rewardAmount
354             );
355         } else {
356             int256 signedBailoutAmount = bailoutAmount.toInt256();
357             // if account is in insolvent state
```

```
            // - liquidator takes the reward
            _balances[liquidator] = _balances[liquidator]
                .add(signedRewardAmount);
            // - reward account becomes bailout account
            _balances[rewardAccount] = _balances[rewardAccount]
                .sub(signedRewardAmount)
                .sub(signedBailoutAmount);
            // - penalty applies (excluding the bailout)
            _balances[penaltyAccount] = _balances[penaltyAccount]
                .add(signedBailoutAmount);
            emit AgreementLiquidated(
                msg.sender, id,
                penaltyAccount,
                liquidator /* rewardAccount */,
                bailoutAmount
            );
            emit Bailout(
                rewardAccount,
                bailoutAmount
            );
        }
    }
```

Listing 3.12: SuperfluidToken::makeLiquidationPayouts()

Recommendation Properly emit the `AgreementLiquidated` event with accurate information to timely reflect state changes. This is very helpful for external analytics and reporting tools.

Status The issue has been fixed by this commit: `811c803`.

3.8 RoundingDown Consistency Of appAllowance in _changeFlow()

- ID: PVE-008
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `ConstantFlowAgreementV1`
- Category: Business Logic [7]
- CWE subcategory: CWE-708 [3]

Description

The Superfluid protocol has the built-in support of two `agreements`, i.e., `Constant Flow Agreement (CFA)` and `Instant Distribution Agreement (IDA)`: The first `agreement` enables the transfer of value from the sender to the receiver at a constant `flowRate` of amount per second; and the second one allows for the distribution of value from the distributor to any number of subscribers with instant solvency validation in one transaction.

For improved scalability, an account may settle each agreement with any number of other accounts. Also, the protocol maintains the aggregated flow information for each sender and receiver pair. It is important to note that the efficiency and scalability is partially supported by the compact encoding of the flow information. Specifically, the encoding, as elaborated in the following `_encodeFlowData()` helper routine, allows for gas-efficient encapsulation of all required flow information within a single 256-bits storage slot.

```
    //
    // Data packing:
    //
    // WORD A: timestamp  flowRate  deposit  owedDeposit //  32b        96b  64        64

    //
    // NOTE:
    // - flowRate has 96 bits length
    // - deposit has 96 bits length too, but 32 bits are clipped-off when storing

    function _encodeFlowData
    (
        FlowData memory flowData
    )
        internal pure
        returns(bytes32[] memory data)
    {
        // enable these for debugging
        // assert(flowData.deposit & type(uint32).max == 0);
        // assert(flowData.owedDeposit & type(uint32).max == 0);
        data = new bytes32[](1);
        data[0] = bytes32(
            ((uint256(flowData.timestamp)) << 224) ((uint256(uint96(flowData.flowRate)) « 128))
            (uint256(flowData.deposit) >> 32 << 64) (uint256(flowData.owedDeposit) » 32));
```

Listing 3.13: ConstantFlowAgreementV1::_encodeFlowData()

Meanwhile, we should note that the compact encoding makes an assumption that though deposit has 96 bits length too, the least-significant 32 bits are clipped-off for storing. With that, we further show below another helper routine, i.e., `_changeFlow()`, that is responsible for the actual update of the flow information between each sender and receiver pair. It comes to our attention that though it is documented (lines 649 – 650) that the internal `appAllowance` variable needs to be computed by rounding down the number. However, the current implementation takes a rounding-up approach (line 651), which brings unnecessary inconsistency.

```
618     /**
619      * @dev change flow between sender and receiver with new flow rate
620      *
621      * NOTE:
```

```
     * - leaving owed deposit unchanged for later adjustment
     * - depositDelta output is always clipped (see _clipDepositNumber)
     */
    function _changeFlow(
        uint256 currentTimestamp,
        ISuperfluidToken token,
        FlowParams memory flowParams,
        FlowData memory oldFlowData
    )
        private
        returns (
            int256 depositDelta,
            uint256 appAllowance,
            FlowData memory newFlowData
        )
    {
        { // ecnlosed block to avoid stack too deep error
            //int256 oldDeposit;

            // STEP 1: calculate old and new deposit required for the flow
            ISuperfluidGovernance gov = ISuperfluidGovernance(ISuperfluid(msg.sender).
                getGovernance());
            uint256 liquidationPeriod = gov.getConfigAsUint256(
                ISuperfluid(msg.sender), token, _LIQUIDATION_PERIOD_CONFIG_KEY);

            //oldDeposit = _calculateDeposit(oldFlowData.flowRate, liquidationPeriod,
                false).toInt256();
            depositDelta = _calculateDeposit(flowParams.flowRate, liquidationPeriod).
                toInt256();

            // for app allowance, rounding down the number instead,
            // in order not to give the downstream app chance to create larger flow rate
            appAllowance = _calculateDeposit(flowParams.flowRate, liquidationPeriod);

            // STEP 2: calculate deposit delta
            depositDelta = depositDelta
                .sub(oldFlowData.deposit.toInt256())
                .add(oldFlowData.owedDeposit.toInt256());

            // STEP 3: update current flow info
            newFlowData = FlowData(
                flowParams.flowRate > 0 ? currentTimestamp : 0,
                flowParams.flowRate,
                oldFlowData.deposit.toInt256().add(depositDelta).toUint256(),
                oldFlowData.owedDeposit // leaving it unchanged for later adjustment
            );
            token.updateAgreementData(flowParams.flowId, _encodeFlowData(newFlowData));
        }

        // STEP 4: update sender and receiver account flow state with the deltas
        int96 totalSenderFlowRate = _updateAccountFlowState(
            token,
```

```
            flowParams.sender,
            oldFlowData.flowRate.sub(flowParams.flowRate, "CFA: flowrate overflow"),
            depositDelta,
            0,
            currentTimestamp
        );
        int96 totalReceiverFlowRate = _updateAccountFlowState(
            token,
            flowParams.receiver,
            flowParams.flowRate.sub(oldFlowData.flowRate, "CFA: flowrate overflow"),
            0,
            0, // leaving owed deposit unchanged for later adjustment
            currentTimestamp
        );

        // STEP 5: emit the FlowUpdated Event
        emit FlowUpdated(
            token,
            flowParams.sender,
            flowParams.receiver,
            flowParams.flowRate,
            totalSenderFlowRate,
            totalReceiverFlowRate,
            flowParams.userData);
    }
```

Listing 3.14: ConstantFlowAgreementV1::_changeFlow()

Recommendation Resolve the inconsistency when computing `appAllowance` in the commented rounding-down manner or the actual rounding-up approach.

Status The issue has been fixed by this commit: `811c803`.

4 Conclusion

In this audit, we have analyzed the `Superfluid` design and implementation. `Superfluid` is a smart contract framework on layer 1 `Ethereum`, enabling the asset movement on-chain by following predefined rules – `agreements`. During the audit, we notice that the current code base is well organized and those identified issues are promptly confirmed and fixed.

References

[1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

[3] MITRE. CWE-708: Incorrect Ownership Assignment. https://cwe.mitre.org/data/definitions/708.html.

[4] MITRE. CWE-837: Improper Enforcement of a Single, Unique Action. https://cwe.mitre.org/data/definitions/837.html.

[5] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[6] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[7] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[8] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

[9] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[10] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[11] PeckShield. PeckShield Inc. https://www.peckshield.com.