SHERLOCK

Security Review For Superfluid

Collaborative Audit Prepared For: **Superfluid**
Lead Security Expert(s): **0x73696d616f**
deadmanwalking
Date Audited: **January 13 - January 15, 2026**

Introduction

The scope of the private audit competition is the super token yield backend support and its Aaave and ERC4624 backend implementation. It is an incremental update to the superfluid protocol.

The goal is to generate yields from selected super tokens' TVL, including USDCx and ETHx, on networks supported by Superfluid.

An extensive write-up about the implementation, which also addresses some of the comments from the private audit, can be found here (https://github.com/superfluid-org/protocol-monorepo/wiki/Yield-Backend).

The rollout of this feature is expected during Q1 of 2026.

Scope

Repository: superfluid-org/protocol-monorepo

Audited Commit: d69e4b0394c38d2760b9b54f173a797b630c9ed1

Final Commit: a7ee24823580510e484e7c5b42941b4c9f919ed8

Files:

- packages/ethereum-contracts/contracts/libs/CallUtils.sol
- packages/ethereum-contracts/contracts/superfluid/AaveETHYieldBackend.sol
- packages/ethereum-contracts/contracts/superfluid/AaveYieldBackend.sol
- packages/ethereum-contracts/contracts/superfluid/ERC4626YieldBackend.sol
- packages/ethereum-contracts/contracts/superfluid/SuperfluidToken.sol
- packages/ethereum-contracts/contracts/superfluid/SuperToken.sol

Final Commit Hash

a7ee24823580510e484e7c5b42941b4c9f919ed8

Findings

Each issue has an assigned severity:

- High issues are directly exploitable security vulnerabilities that need to be fixed.
- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.

- Low/Info issues are non-exploitable, informational findings that do not pose a security risk or impact the system's integrity. These issues are typically cosmetic or related to compliance requirements, and are not considered a priority for remediation.

Issues Found

High	Medium	Low/Info
0	3	9

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Issue M-1: AaveETHYieldBackend uses wrong WETH address for Polygon (returns WETH instead of WPOL) [RESOLVED]

Source: https://github.com/sherlock-audit/2026-01-superfluid-update-jan-13th/issues/12

Summary

`AaveETHYieldBackend` uses the wrong address for Polygon's native token wrapper - it returns WETH instead of WPOL (the native token wrapper for POL/MATIC).

Vulnerability Detail

In `getWETHAddress()`, for chain ID 137 (Polygon), the contract returns the WETH address (`0x7ceB23fD6bC0adD59E62ac25578270cFf1b9f619`) which is a bridged ETH token, instead of the native token wrapper WPOL at `0x0d500B1d8E8eF31E21C99d1Db9A6444d3ADf1270`.

The contract is designed to wrap native ETH/tokens to their wrapped ERC20 equivalent and deposit them to Aave. On Polygon, native tokens (POL/MATIC) should be wrapped to WPOL, not WETH. WETH on Polygon is simply a bridged representation of Ethereum's ETH and is not the native token wrapper.

Impact

The `AaveETHYieldBackend` is unusable on Polygon. When users try to deposit native POL, the contract attempts to interact with the WETH contract which does not have a `deposit()` function that accepts native POL, causing all deposits to fail.

Code Snippet

https://github.com/sherlock-audit/2026-01-superfluid-update-jan-13th/blob/main/protocol-monorepo/packages/ethereum-contracts/contracts/superfluid/AaveETHYieldBackend.sol#L46-L48

```
if (block.chainid == 137) { // Polygon
    return 0x7ceB23fD6bC0adD59E62ac25578270cFf1b9f619; //@audit ISSUE WETH not
    ↪  WPOL, WPOL is 0x0d500B1d8E8eF31E21C99d1Db9A6444d3ADf1270
}
```

Tool Used

Manual Review

Recommendation

Return the correct WPOL address for Polygon:

```
if (block.chainid == 137) { // Polygon
    // Note this token has the symbol WPOL, wrapping the native token POL
    return 0x0d500B1d8E8eF31E21C99d1Db9A6444d3ADf1270;
}
```

Discussion

d10r

good catch, thanks! Fixed in https://github.com/superfluid-org/protocol-monorepo/pull/2125/changes/ac0b85693fa387108d8d7ae1237c28d240719846

Issue M-2: Aave rounding behavior allows malicious users to drain accumulated yield via small withdrawals [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2026-01-superfluid-update-jan-13th/issues/18

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

Aave's rounding behavior on deposits and withdrawals allows malicious users to drain accumulated yield by performing many small withdrawals, each burning more scaled balance than the actual asset value withdrawn.

Vulnerability Detail

When depositing to Aave, the scaled balance received is rounded down (`scaledBalance = amount / index`). When withdrawing, Aave rounds up the scaled balance burned (`scaledBalanceBurned = ceil(amount / index)`). The SuperToken mints/burns tokens at a 1:1 ratio with the requested amount, ignoring these rounding discrepancies.

Consider this scenario with an Aave index of 2:

- Two users each have 50 SuperTokens, backed by 100 aTokens total (50 scaled balance)
- A malicious user withdraws their 50 SuperTokens in a loop, 1 wei at a time
- Each 1 wei withdrawal burns `ceil(1/2) = 1` scaled balance
- After 50 iterations, the user has withdrawn 50 wei but burned all 50 scaled balance
- The second user's 50 SuperTokens are now backed by 0 aTokens

The attack is particularly impactful for tokens like WBTC where 1 wei equals ~$0.001, making the gas cost worthwhile for the attacker.

Numerical Example:

1. Aave index = 1.1
2. User deposits 50 tokens → receives `50 / 1.1 = 45` scaled balance (rounded down)
3. Aave holds `45 * 1.1 = 49.5  49` aTokens (not 50)
4. Index grows to 1.15
5. User withdraws 48 aTokens → burns `ceil(48 / 1.15) = 42` scaled balance
6. Remaining: `45 - 42 = 3` scaled balance = `3 * 1.15 = 3.45` aTokens

7. Yield lost: `45 * 1.15 - 48 - 3.45 = 0.3` aTokens due to rounding

Impact

A malicious user can systematically drain all accumulated Aave yield by performing repeated small withdrawals. This results in loss of yield for all SuperToken holders and potential insolvency where the total SuperToken supply exceeds the backing aTokens.

Code Snippet

https://github.com/sherlock-audit/2026-01-superfluid-update-jan-13th/blob/main/protocol-monorepo/packages/ethereum-contracts/contracts/superfluid/SuperToken.sol#L778-L779

```
_mint(msg.sender, account, amount, userData.length != 0 /* invokeHook */,
    userData.length != 0 /* requireReceptionAck */, userData, new bytes(0));
```

https://github.com/sherlock-audit/2026-01-superfluid-update-jan-13th/blob/main/protocol-monorepo/packages/ethereum-contracts/contracts/superfluid/AaveYieldBackend.sol#L56-L59

```
function withdraw(uint256 amount) public virtual {
    // withdraw amount asset by redeeming the corresponding aTokens amount
    AAVE_POOL.withdraw(address(ASSET_TOKEN), amount, address(this));
}
```

Tool Used

Manual Review

Recommendation

Modify the withdrawal to request a slightly lower amount from Aave that accounts for rounding, while still burning the full SuperToken amount requested by the user:

```
function withdraw(uint256 requestedAmount) public virtual returns (uint256) {
    uint256 index = AAVE_POOL.getReserveNormalizedIncome(address(ASSET_TOKEN));
    uint256 scaledAmount = (requestedAmount * 1e27) / index;
    uint256 safeWithdrawAmount = (scaledAmount * index) / 1e27;
    AAVE_POOL.withdraw(address(ASSET_TOKEN), safeWithdrawAmount, address(this));
}
```

This ensures the scaled balance burned matches the expected value, and the user takes the rounding loss.

Discussion

d10r

After considerable deliberation we decided to accept this risk for the sake of a simpler implementation, making the SuperToken admin responsible for quantitatively assessing the risk before enabling. The reasoning behind that is:

- allowing the downgrade amount to differ from what was requested would be inconvenient, possibly also breaking some integrations not expecting that
- there is no economic incentive for an attacker (it's even negative considering there's a tx cost)
- the maximum loss which could be incurred doesn't scale with the deposited amount and is bound by sufficiently predictable constraints: the asset decimals, Aave index value and the chain's throughput. None of those is expected to suddenly change by orders of magnitude
- we may eventually switch to an implementation which decouples deposit/withdraw from upgrade/downgrade

A risk assessment for USDC on Base was made in https://github.com/superfluid-org/protocol-monorepo/pull/2125 allowbreak #issuecomment-3661329674, concluding that with the current constraints it wouldn't be possible with such an attack to burn deposits faster than yield is generated.

The risk will be highlighted in the documentation, and for asset tokens with reduced decimals (like USDC) monitoring of downgrade activity will be recommended in order to ensure early detection of any such activity.

Issue M-3: Underlying token donations can temporarily DOS surplus claims from yield backends [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2026-01-superfluid-update-jan-13th/issues/19

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

`withdrawSurplus` from the yield backends can be DoS'd for the protocol as a donor can increase `ASSET_TOKEN.balanceOf(address(this))`, inflating `surplusAmount`, while the function attempts to realize the full amount via `AAVE_POOL.withdraw`, which can revert if that amount is not redeemable from Aave at the time.

Vulnerability Detail

`surplusAmount` is computed as `A_TOKEN.balanceOf(this) + ASSET_TOKEN.balanceOf(this) - normalizedTotalSupply - 100`, but the backend withdraws surplusAmount only from Aave. If idle underlying exists on the SuperToken (primarily via direct donations), surplusAmount can exceed the redeemable Aave portion / available liquidity, causing withdrawSurplusFromYieldBackend() to revert.

Impact

Temporary DOS of `withdrawSurplusFromYieldBackend`. This comes at a cost to the attacker and is only temporary as any increase in backing via new upgrades will mitigate this. However, if new backing in AAVE is added and then `withdrawSurplusFromYieldBackend` is called, withdrawing from AAVE, it could decrease the portion of assets earning yield causing the intended recipient to miss out

Code Snippet

-

Tool Used

Manual Review

Recommendation

If the yield backend is enabled, then `ASSET_TOKEN.balanceOf(address(this))` should be 0 in all times except for donations. It would be useful to add an admin function on the SuperToken to deposit all idle balances (`ASSET_TOKEN.balanceOf(address(this))`) in the backend , if configured.

Discussion

d10r

The report is valid, but I wouldn't agree with the impact level.

1. In practice, the excess amounts of underlying will be tiny compared to the deposited amounts (unless an attacker is very generous)
2. yield withdrawal is not a critical operation worth DOSing, it will likely be an operation periodically executed by an admin multisig

Because of that we find it acceptable to trade this limitation for less code, but will mention it in the documentation.

Issue L-1: Yield backend cannot be enabled on a freshly deployed SuperToken due to zero-amount deposit [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2026-01-superfluid-update-jan-13th/issues/8

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

The requirement that yield backend `deposit` must be non-zero will cause an inability to enable yield for a freshly deployed SuperToken as the admin will call `enableYieldBackend()` on a token with zero underlying balance and the backend `deposit(0)` will revert.

Vulnerability Detail

In `SuperToken.enableYieldBackend` the contract always delegatecalls `IYieldBackend.deposit(depositAmount)` even when `depositAmount` is `0`, while the backend implementations require `amount > 0`.

```
function enableYieldBackend(IYieldBackend newYieldBackend) external onlyAdmin {
    require(address(_yieldBackend) == address(0), "yield backend already set");
    _yieldBackend = newYieldBackend;
    delegateCallChecked(address(_yieldBackend),
    ↪  abi.encodeCall(IYieldBackend.enable, ()));
    ...
    delegateCallChecked(address(_yieldBackend),
    ↪  abi.encodeCall(IYieldBackend.deposit, (depositAmount)));
    emit YieldBackendEnabled(address(_yieldBackend), depositAmount);
}
```

Impact

Intuitively, the admin should be able to enable a backend on a token even without an underlying balance. Otherwise, multiple transactions would be needed to deploy a token from scratch.

Code Snippet

-

Tool Used

Manual Review

Recommendation

Add a zero balance check in `enableYieldBackend` and skip the deposit if the balance is 0

Discussion

d10r

This is a limitation of the mechanism we prefer to keep for the sake of slightly simpler logic, but will be added to the documentation.

Issue L-2: Use `forceApprove` for enabling and disabling yield backends [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2026-01-superfluid-update-jan-13th/issues/9

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

Use SafeERC20::forceApprove since you are dealing with a range of potential tokens instead of standard IERC20 approve.

Vulnerability Detail

The strategies use simple ERC20 approvals for enabling and disabling yield backends. While this works for most tokens, even for USDT that disallows non-zero to non-zero approvals, it is safer to use `forceApprove` from Openzeppelin which also checks the return value and ensures correctness in the process.

```
function enable() external {
    // approve Aave pool to fetch asset
    ASSET_TOKEN.approve(address(AAVE_POOL), type(uint256).max);
}
```

Impact

Potential issues with approvals on non-standard tokens and unchecked return value

Code Snippet

-

Tool Used

Manual Review

Recommendation

Use `forceApprove` from OZ `SafeERC20` to approve tokens. See here for the implementation.

Discussion

d10r

Should not be an issue with USDT as the approval would go from zero to non-zero and back. The failure mode here would be that enabling a yield backend fails or that an enabled yield backend can't take underlying from the SuperToken. Both are not safety hazards. I'd rather stick to plain `approve` and leave the responsibility to vet and understand the underlying token to the SuperToken admin, with the option of adjusting the yield backend code if necessary for a peculiar underlying token.

Issue L-3: Withdrawals from AAVE can fail due to liquidity shortages, which will disallow the yield back-end from being disabled [ACKNOWLEDGED]

Source:
https://github.com/sherlock-audit/2026-01-superfluid-update-jan-13th/issues/10

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

Withdrawing from AAVE is subject to liquidity constraints due to potential high utilization (all underlying tokens are lent out). This means downgrades and disabling of the `AAVEYieldBackend` can be DoS'd by market conditions.

Vulnerability Detail

Suppose a liquidity shortage event makes AAVE USDC utilization go to 100% (due to bad collateral, sudden lack of trust in the protocol). In this case, users will be unable to withdraw and the yield backend can not be disabled since disabling also attempts a full withdrawal. The bigger the attempted withdrawal amount (if the SuperToken backing grows by a lot), the bigger the likelihood of this happening. Although this scenario is highly unlikely, it is not impossible.

Impact

DOS of AAVE yield backend disable and withdrawals. In the case of the external protocol getting hacked, it would be better to attempt a partial withdrawal and disable the backend than a full one that reverts.

Code Snippet

Tool Used

Manual Review

Recommendation

Document and surface this risk to users as downgrades are not guaranteed to be instantly available under extreme Aave utilization. For disabling the backend, consider allowing partial withdrawals (specified by admin) as a separate function to make sure

funds can be recovered incrementally before disabling the backend instead of an all-or-nothing transaction

Discussion

d10r

This observation is correct and will be added to the documentation. We decided to not preemptively add logic for dealing with such potential scenarios, but keep the logic simple while making sure the SuperToken admin can't get stuck in such a situation, thanks to

1. the stateless nature of the yield backend (doesn't take custody of any assets, but is merely a swappable logic contract)
2. the SuperToken contract itself being upgradable, e.g. in case `disableYieldBackend` would fail with the current implementation

Issue L-4: `delegateCallChecked` reverts without surfacing the error, limiting visibility into yield backend issues [RESOLVED]

Source:
https://github.com/sherlock-audit/2026-01-superfluid-update-jan-13th/issues/11

Summary

When the delegatecall fails, it reverts with the generic error message. This limits visibility into the cause of the issue, making debugging and troubleshooting challenging when adding more strategies.

Vulnerability Detail

In `CallUtils` the `delegateCallChecked` function is defined as follows:

```
function delegateCallChecked(address target, bytes memory callData) {
    // free function, implicitly internal
    // solhint-disable-next-line avoid-low-level-calls
    (bool success,) = target.delegatecall(callData);
    require(success, "CallUtils: delegatecall failed");
}
```

When the delegatecall is unsucessful, it reverts with a generic error message. The return data however are already copied into memory so it would be useful to surface them to give both users (in the ui) and devs (when adding/testing more strategies) visibility into what went wrong.

Impact

Lack of visibility into delegatecall reverts making debugging and troubleshooting challenging when adding more strategies.

Code Snippet

-

Tool Used

Manual Review

Recommendation

Consider surfacing the return data upon an unsuccessful delegatecall

Discussion

d10r

changed as recommended in https://github.com/superfluid-org/protocol-monorepo/pull/2125/commits/2a9428b64dfcc4ff5782480a6e79fb7d4b8bcb5a

Issue L-5: AaveYieldBackend.deposit() DoS when Aave pool is paused, frozen, or at supply cap [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2026-01-superfluid-update-jan-13th/issues/13

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

`AaveYieldBackend.deposit()` reverts when the Aave pool is paused, frozen, or the supply cap is reached, causing a DoS for all SuperToken upgrade/mint operations.

Vulnerability Detail

The `deposit()` function calls `AAVE_POOL.supply()` directly without handling cases where Aave may reject the deposit. Aave pools can be paused, frozen, or have supply caps that prevent new deposits. When any of these conditions occur, all SuperToken operations that trigger deposits (such as `upgrade()`, `upgradeTo()`, and `selfMint()`) revert.

The underlying assets could simply be kept in the SuperToken contract when Aave is unavailable, but currently there is no fallback mechanism.

Impact

Users are unable to mint or upgrade SuperTokens when the underlying Aave pool is paused, frozen, or at supply capacity. This DoSes the core functionality of the SuperToken for an indeterminate period of time until Aave governance re-enables the pool.

Code Snippet

https://github.com/sherlock-audit/2026-01-superfluid-update-jan-13th/blob/main/protocol-monorepo/packages/ethereum-contracts/contracts/superfluid/AaveYieldBackend.sol#L49-L54

```
function deposit(uint256 amount) public virtual {
    // TODO: can this constraint break anything?
    require(amount > 0, "amount must be greater than 0");
    // Deposit asset and get back aTokens
    AAVE_POOL.supply(address(ASSET_TOKEN), amount, address(this), 0);
}
```

Tool Used

Manual Review

Recommendation

Implement a try-catch mechanism to handle Aave deposit failures gracefully. When the deposit fails, keep the underlying assets in the SuperToken contract:

```
function deposit(uint256 amount) public virtual {
    require(amount > 0, "amount must be greater than 0");
    try AAVE_POOL.supply(address(ASSET_TOKEN), amount, address(this), 0) {
        // Deposit succeeded
    } catch {
        // Aave deposit failed (paused, frozen, supply cap reached)
        // Keep the underlying assets in the contract
    }
}
```

Discussion

d10r

In general this is in line with the design choice to have a simple implementation, accepting that extraordinary circumstances like the ones described require intervention of the SuperToken admin.

One possible concern here would be if there's a possibility for an attacker to sandwich upgrade operations such that they fail, e.g. by using a flashloan to temporarily bring the pool to its supply cap. But I don't see any possibility to do so cheaply. Do you? Assuming there's not, I'd prefer to stick to the current implementation, document this risks as a responsibility for the SuperToken admin to be aware of, and highlight that the implementation may not be suited for assets likely to hit the supply cap.

Issue L-6: ERC4626YieldBackend does not support vaults with deposit/withdrawal fees or slashing [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2026-01-superfluid-update-jan-13th/issues/14

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

`ERC4626YieldBackend` does not support ERC4626 vaults with deposit/withdrawal fees or slashing mechanisms, leading to loss of funds for other users or potential insolvency.

Vulnerability Detail

The `ERC4626YieldBackend` mints/burns SuperTokens at a 1:1 ratio with the underlying asset amount, but many ERC4626 vaults charge deposit and/or withdrawal fees. When a user deposits into a vault with fees, fewer shares are received than expected. When withdrawing, more shares are burned than expected. Since the SuperToken mints the exact requested amount regardless of fees, other users absorb the loss.

Additionally, vaults with slashing mechanisms (e.g., staking vaults where validators can be slashed) are problematic because losses are never socialized among SuperToken users. The vault's balance decreases, but the SuperToken's total supply remains unchanged, leading to insolvency where the last users to withdraw cannot redeem their tokens.

Impact

1. Users who deposit/withdraw cause losses to other SuperToken holders, as fees are effectively paid from the shared pool.
2. A slashing event causes the SuperToken to become insolvent - the total supply of SuperTokens exceeds the actual underlying assets available, and some users are unable to redeem their tokens.

Code Snippet

https://github.com/sherlock-audit/2026-01-superfluid-update-jan-13th/blob/main/protocol-monorepo/packages/ethereum-contracts/contracts/superfluid/ERC4626YieldBackend.sol#L10-L19

```
/// @title A SuperToken yield backend for ERC4626 compliant vaults.
contract ERC4626YieldBackend is IYieldBackend {
```

```
    IERC20 public immutable ASSET_TOKEN;
    IERC4626 public immutable VAULT;
    address public immutable SURPLUS_RECEIVER;

    constructor(IERC4626 vault, address surplusReceiver) {
        VAULT = vault;
        ASSET_TOKEN = IERC20(vault.asset());
        SURPLUS_RECEIVER = surplusReceiver;
    }
```

https://github.com/sherlock-audit/2026-01-superfluid-update-jan-13th/blob/main/protocol-monorepo/packages/ethereum-contracts/contracts/superfluid/ERC4626YieldBackend.sol#L29-L36

```
function deposit(uint256 amount) external {
    require(amount > 0, "amount must be greater than 0");
    VAULT.deposit(amount, address(this));
}

function withdraw(uint256 amount) external {
    VAULT.withdraw(amount, address(this), address(this));
}
```

Tool Used

Manual Review

Recommendation

1. Document that `ERC4626YieldBackend` only supports well-behaved vaults without fees or slashing (e.g., sDAI).
2. Consider adding validation in the constructor or `enable()` to check if the vault charges fees by comparing `previewDeposit()`/`previewWithdraw()` with actual amounts.
3. For broader vault support, track actual shares received/burned and adjust SuperToken minting/burning accordingly, or implement a mechanism to socialize losses among SuperToken holders.

Discussion

d10r

This limitation will be documented, leaving the responsibility to only use compatible vaults to the SuperToken admin.

Issue L-7: ERC4626YieldBackend.withdrawMax() silently succeeds when vault is paused, leaving funds stuck [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2026-01-superfluid-update-jan-13th/issues/15

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

`ERC4626YieldBackend.withdrawMax()` silently succeeds when the vault is paused, leaving funds stuck after `disableYieldBackend()` is called.

Vulnerability Detail

Per EIP-4626, when a vault is paused, `maxWithdraw()` should return 0 rather than revert. The `withdrawMax()` function checks if `balance > 0` before attempting withdrawal, so when a vault is paused, it simply does nothing and returns successfully.

The problem is that `SuperToken.disableYieldBackend()` calls `withdrawMax()` and then clears the yield backend reference. If the vault is paused at the time of disabling, the call succeeds but funds remain locked in the vault. Since the yield backend reference is cleared, there is no way to recover the funds afterward.

```
function disableYieldBackend() external onlyAdmin {
    require(address(_yieldBackend) != address(0), "yield backend not set");
    address oldYieldBackend = address(_yieldBackend);

    _skipSelfMint = true;
    delegateCallChecked(address(_yieldBackend),
    ↪  abi.encodeCall(IYieldBackend.withdrawMax, ()));
    _skipSelfMint = false;

    delegateCallChecked(address(_yieldBackend),
    ↪  abi.encodeCall(IYieldBackend.disable, ()));
    _yieldBackend = IYieldBackend(address(0)); // Backend cleared, funds stuck!
    emit YieldBackendDisabled(oldYieldBackend);
}
```

Impact

When `disableYieldBackend()` is called while the ERC4626 vault is paused, all funds deposited in the vault become permanently stuck. The yield backend is disabled and

there is no mechanism to recover the funds.

Code Snippet

https://github.com/sherlock-audit/2026-01-superfluid-update-jan-13th/blob/main/protocol-monorepo/packages/ethereum-contracts/contracts/superfluid/ERC4626YieldBackend.sol#L38-L43

```
function withdrawMax() external {
    uint256 balance = VAULT.maxWithdraw(address(this));
    if (balance > 0) {
        VAULT.withdraw(balance, address(this), address(this));
    }
}
```

Tool Used

Manual Review

Recommendation

Add a check in `disableYieldBackend()` to verify that funds were actually withdrawn, or add a minimum expected amount parameter:

```
function withdrawMax() external returns (uint256) {
    uint256 balance = VAULT.maxWithdraw(address(this));
    if (balance > 0) {
        VAULT.withdraw(balance, address(this), address(this));
    }
    return balance;
}
```

Then in `SuperToken.disableYieldBackend()`:

```
function disableYieldBackend(uint256 minExpectedAmount) external onlyAdmin {
    // ... existing code ...
    uint256 withdrawn = delegateCallChecked(address(_yieldBackend),
    ↪  abi.encodeCall(IYieldBackend.withdrawMax, ()));
    require(withdrawn >= minExpectedAmount, "insufficient withdrawal");
    // ... rest of function ...
}
```

Discussion

d10r

No funds would get stuck/lost. The design choice here is to keep the yield backend contracts simple and stateless (never own any assets) and leave the SuperToken admin with the option to switch to a different implementation if necessary for dealing with extraordinary circumstances like paused vaults. Due to that, silent failure is ok. This will be added to the documentation.

Issue L-8: ERC4626YieldBackend.withdrawSurplus() uses convertToAssets() which doesn't account for fees [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2026-01-superfluid-update-jan-13th/issues/16

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

`ERC4626YieldBackend.withdrawSurplus()` uses `convertToAssets()` to calculate the surplus, which does not account for withdrawal fees, potentially causing the function to revert or withdraw more than the actual surplus.

Vulnerability Detail

The `withdrawSurplus()` function calculates the vault's asset value using `convertToAssets()`:

```
uint256 vaultAssets = VAULT.convertToAssets(
    VAULT.balanceOf(address(this))
);
```

Per EIP-4626, `convertToAssets()` returns the amount of assets that would be exchanged for the given shares, but it does not account for withdrawal fees. For vaults that charge withdrawal fees, the actual redeemable amount is less than what `convertToAssets()` reports.

This causes the calculated `surplusAmount` to be higher than the actual withdrawable surplus. When `VAULT.withdraw(surplusAmount, ...)` is called, it may:

1. Revert if the vault cannot fulfill the inflated withdrawal amount
2. Withdraw assets that belong to SuperToken holders (not actual surplus)

Impact

For ERC4626 vaults with withdrawal fees, `withdrawSurplus()` either reverts (DoSing surplus withdrawal) or incorrectly withdraws funds that belong to SuperToken holders, causing a loss for users.

Code Snippet

https://github.com/sherlock-audit/2026-01-superfluid-update-jan-13th/blob/main/protocol-monorepo/packages/ethereum-contracts/contracts/superfluid/ERC4626YieldBackend.sol#L45-L55

```
function withdrawSurplus(uint256 totalSupply) external {
    (uint256 normalizedTotalSupply, ) = ISuperToken(address(this))
        .toUnderlyingAmount(totalSupply);

    uint256 vaultAssets = VAULT.convertToAssets(
        VAULT.balanceOf(address(this))
    );

    uint256 surplusAmount = vaultAssets + ASSET_TOKEN.balanceOf(address(this)) -
    ↪  normalizedTotalSupply;
    VAULT.withdraw(surplusAmount, SURPLUS_RECEIVER, address(this));
}
```

Tool Used

Manual Review

Recommendation

Use `previewRedeem()` instead of `convertToAssets()` to get an accurate estimate that includes fees, or use `maxWithdraw()` to get the maximum withdrawable amount:

```
function withdrawSurplus(uint256 totalSupply) external {
    (uint256 normalizedTotalSupply, ) = ISuperToken(address(this))
        .toUnderlyingAmount(totalSupply);

    // Use maxWithdraw to account for fees and other constraints
    uint256 vaultAssets = VAULT.maxWithdraw(address(this));

    uint256 surplusAmount = vaultAssets + ASSET_TOKEN.balanceOf(address(this)) -
    ↪  normalizedTotalSupply;
    VAULT.withdraw(surplusAmount, SURPLUS_RECEIVER, address(this));
}
```

Discussion

d10r

This implementation is not considered as suitable for vaults with fee. This limitation will be added to the documentation as a responsibility for the SuperToken admin to verify.

Issue L-9: SuperToken._skipSelfMint silently skips minting, breaking custom SuperToken implementations [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2026-01-superfluid-update-jan-13th/issues/17

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

The `_skipSelfMint` transient flag silently skips minting in `selfMint()`, which breaks contracts that extend SuperToken and expect `selfMint()` to always mint tokens when called.

Vulnerability Detail

The `_skipSelfMint` flag is used internally to prevent a mint/burn loop when withdrawing ETH from the yield backend. When set to `true`, `selfMint()` silently returns without minting any tokens:

```
function selfMint(
    address account,
    uint256 amount,
    bytes memory userData
)
    external virtual override
    onlySelf
{
    if (!_skipSelfMint) {
        // ... actual minting logic ...
    }
}
```

The problem is that contracts extending SuperToken (e.g., custom wrapper contracts) may implement a `receive()` function that calls `selfMint()` expecting tokens to be minted. If `_skipSelfMint` is set, the call succeeds but no tokens are minted, and the accounting gets corrupted.

For example, a custom SuperToken with fee logic:

```
receive() external payable {
    uint256 fees = msg.value * 1 / 10000; // 1 BPS fee
    uint256 amount = msg.value - fees;
    this.selfMint(msg.sender, amount, "");
```

```
    totalFees += fees;
}
```

When `_skipSelfMint` is true, this code collects fees but mints nothing, causing users to lose funds.

Impact

Custom SuperToken implementations that have other logic in their `receive()` function, such as the example above, will become broken. The current SETH already has this problem, since it emits the event `emit TokenUpgraded(msg.sender, msg.value);` on `receive()`, but no tokens are actually minted.

Code Snippet

https://github.com/sherlock-audit/2026-01-superfluid-update-jan-13th/blob/main/protocol-monorepo/packages/ethereum-contracts/contracts/superfluid/SuperToken.sol#L765-L781

```
function selfMint(
    address account,
    uint256 amount,
    bytes memory userData
)
    external virtual override
    onlySelf
{
    if (!_skipSelfMint) {
        if (address(_yieldBackend) != address(0)) {
            delegateCallChecked(address(_yieldBackend),
            ↪  abi.encodeCall(IYieldBackend.deposit, (amount)));
        }

        _mint(msg.sender, account, amount, userData.length != 0 /* invokeHook */,
            userData.length != 0 /* requireReceptionAck */, userData, new bytes(0));
    }
}
```

Tool Used

Manual Review

Recommendation

Should be flagged to any SuperToken implementation, so it makes sure when the skipSelfMint flag is true, all logic in the receive function should be skipped, not just selfMint().

Discussion

d10r

> The current SETH already has this problem, since it emits the event emit TokenUpgraded(msg.sender, msg.value); on receive(), but no tokens are actually minted.

This is a great observation we missed so far. It doesn't affect onchain accounting, but it does affect offchain data (e.g. via subgraph), thus before enabling on any currently deployed native token wrapper an impact assessment will be done.

Also, this will be added to the documentation as a responsibility for the SuperToken admin to verify.

d10r

PS: I don't currently see any way to redeem ETH to the SETH contract (as is deployed now) without having this unwanted `TokenUpgraded` event emitted. If you do, please let me know!

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.