PERIMETER

Tapioca DAO YieldBox Strategies

Testing Reinforcement Report

06/18/2024

Supervised By:
Rappie | Lead Fuzzing Specialist
rappie@perimetersec.io

0xScourgedev | Lead Fuzzing Specialist
0xscourgedev@perimetersec.io

Prepared By:
nican0r | Junior Fuzzing Specialist
https://x.com/nican0r

Table of Contents

Services Provided

Perimeter has successfully delivered a comprehensive suite of services that include:

- **Test Suite Development:**
    - Contracts in scope had unit tests created to achieve > 70% coverage to ensure proper functionality.
    - Stateless fuzz tests were then added on top of the created unit tests to provide more thorough testing of possible edge cases.
- **Documentation of Branches:**
    - Each covered branch was thoroughly documented with clear diagrams and concise in-code comments.
- **Creation of a Final Report:**
    - Created this final report, which includes our methodology, with all findings and their corresponding PoCs, providing a comprehensive overview of the engagement's outcomes.

Files in Scope

The engagement focuses on the files listed below, acquired from commit 78adb9dcb41e3e6e7a911658636045dcee26db24.

File	nSLOC
contracts/sdai/sDaiStrategy.sol	98
contracts/glp/GlpStrategy.sol	137
Total	**235**

Files Out of Scope

Files outside the scope were not directly considered in achieving the target. However, since many of these files are utilized by those within the scope, a significant portion was indirectly covered.

Methodology

For both contracts within scope a unit testing suite was developed. This required outlining the possible branches of code paths that could be executed and which are described in the Specifications section. Additionally, high level specifications of expected system behavior from a user perspective was outlined to guide the creation of more specific unit tests.

After achieving coverage over all meaningful branches with unit tests, some unit tests were abstracted to allow fuzzing their input values to offer greater certainty of correct behavior.

The test structure used for allowing these fuzzed values in unit tests consisted of wrappers, fuzz and implementation functions.

- Wrappers evaluate a single value for a unit test implementation(suffixed with _wrapper)
- Fuzz tests take a random input value to evaluate the implementation (prefixed with testFuzz)
- Implementations hold the actual test logic and assertions (prefixed with test_, no suffix)

NOTE: some tests which would not benefit from fuzz testing only have an implementation and no fuzz or wrapper functions.

Specifications

The following specifications were defined and tested on the contracts.

GlpStrategy

1. tsGLP passed in on deposit is staked for GlpStrategy
2. GLP bought with WETH rewards is staked for GlpStrategy
3. Harvesting uses all the WETH rewards balance if it's nonzero
4. Only YieldBox can withdraw and deposit
5. Depositing sGLP directly to strategy should fail
6. Calling harvest with 0 rewards accumulated doesn't revert
7. User balance of sGLP increases by amount on call to withdraw
8. GlpStrategy balance of sGLP decreases on withdrawal
9. User can always withdraw up to the full amount of GLP + WETH rewards in the GlpStrategy
10. User can only withdraw yield accumulated for their shares

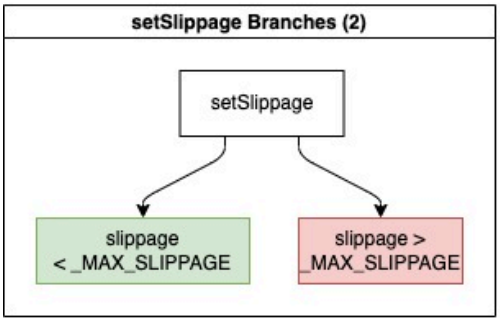
sDaiStrategy

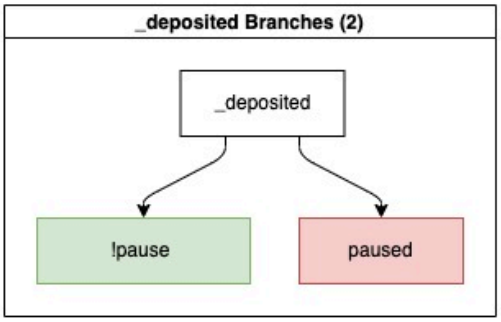
1. tDAI passed in on deposit is deposited for sDaiStrategy
2. User can always withdraw as much as they deposited
3. Only YieldBox can withdraw and deposit into strategy
4. Depositing sDAI directly to strategy should not result in user getting shares
5. Withdrawing with 0 savings accumulated doesn't revert
6. User balance of tDAI increases by amount on call to withdraw
7. sDaiStrategy balance of sDAI decreases on withdrawal
8. User can always withdraw up to the full amount of sDAI in the GlpStrategy
9. User can only withdraw share + yield accumulated for their shares
10. User withdrawing their share doesn't affect other's ability to withdraw
11. Tokens are added to deposit queue if threshold isn't met when depositing
12. Deposit queue gets fully deposited, no dust remains
13. User can withdraw if their assets remain in queue

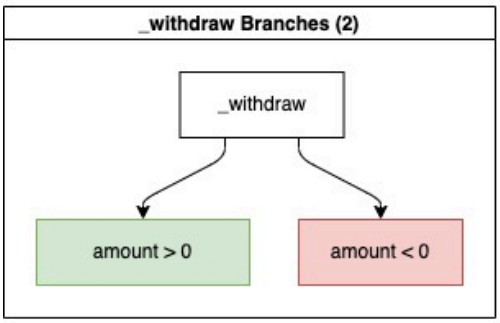
NOTE: any tests that make assertions with initialUserBalance - 1 are taking into account that the user loses a dust amount when withdrawing.

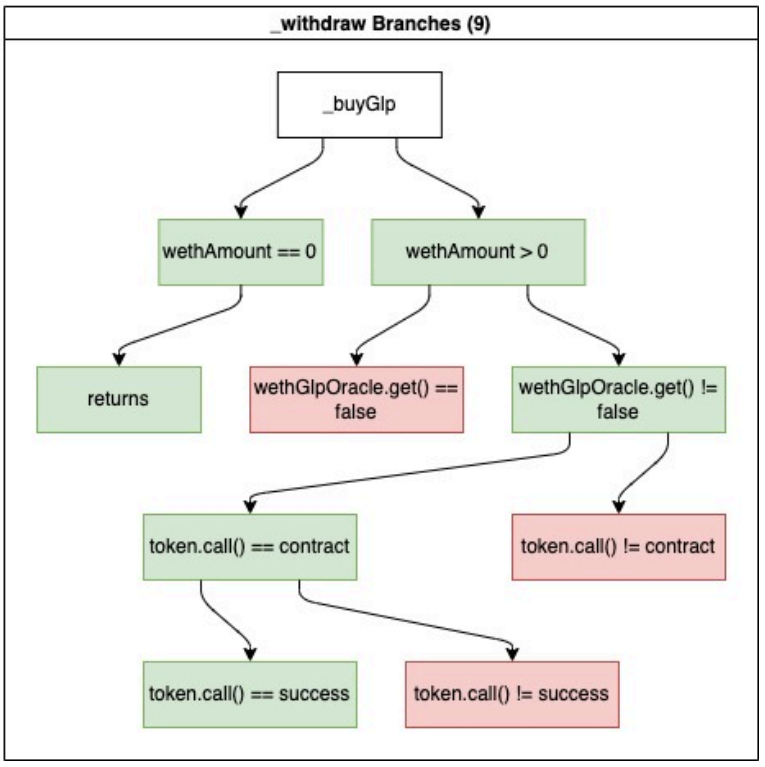
The branches covered using the above specifications are shown in the following diagrams, where red blocks indicate revert paths and green blocks indicate successful execution.

GlpStrategy Branches

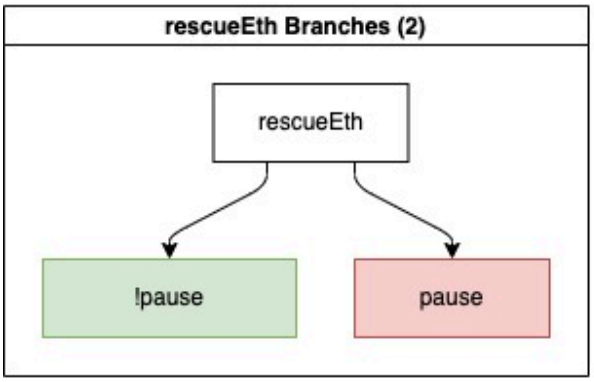


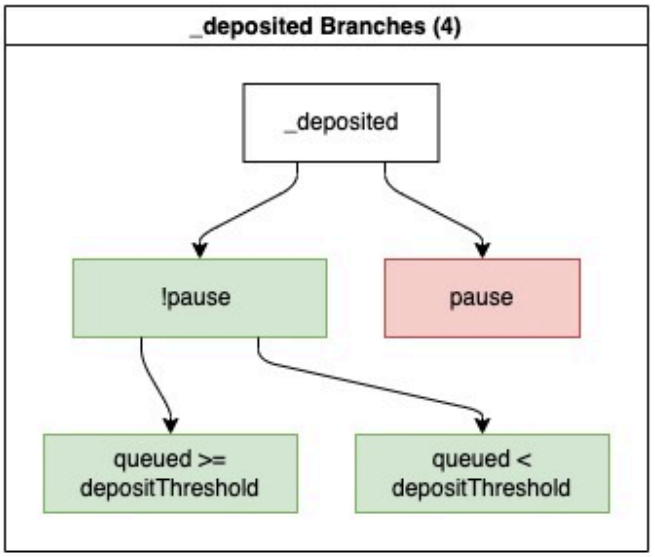


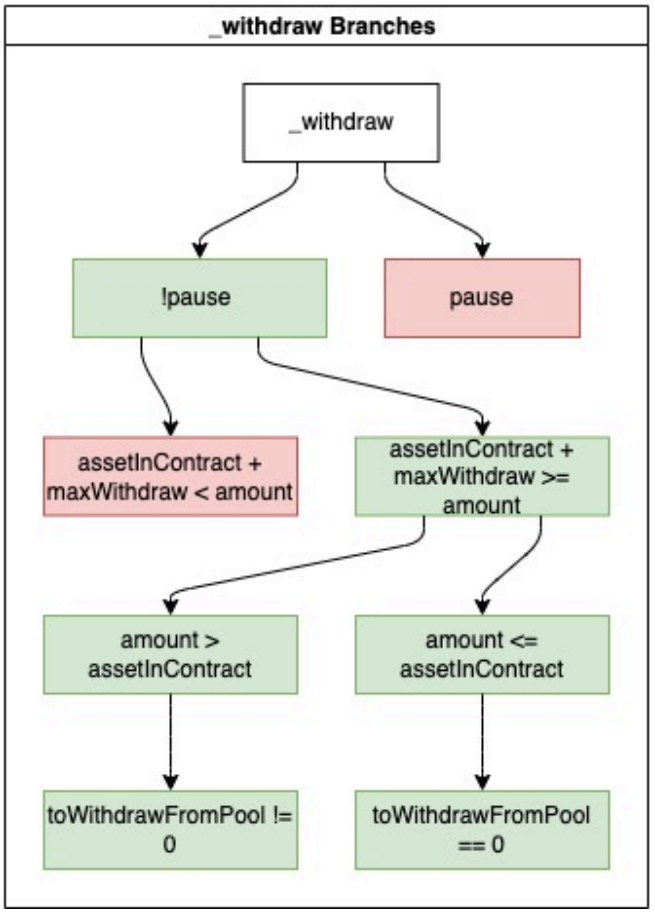




sDaiStrategy Branches







L-01: harvestable function missing return value

Severity

Low

Description

`sDaiStrategy::harvestable` always returns a 0 value for the accumulated savings. When calling the same `sDai::maxWithdraw` function directly from the test contract this returns the correct value.

This is because the harvestable function queries the `maxWithdraw` amount but doesn't return it:

```
function harvestable() external view returns (uint256 result) {
        // @audit need to return this value
        sDai.maxWithdraw(address(this));
}
```

Proof of concept

Demonstrated in `test_harvestable_with_accumulation`.

Recommendation

Return result from the `harvestable` function.

Response

INFO-01: Full reward amount can only be withdrawn with shares for first depositor

Severity

Informational

Description

The first depositor into `GlpStrategy` will have all shares allocated to them `(shares[depositor] == totalSupply)`, but due to rounding in `YieldBox`, if they have accumulated rewards on their deposit and try to withdraw by passing in the balance of the strategy (which they are owed), the conversion of the amount they are withdrawing is greater than the `totalSupply` of shares, they are therefore only able to withdraw their full amount by passing in the `totalSupply` of shares.

Proof of concept

Adding the following console logs to the ERC1155 contract's `_burn` function:

```
function _burn(address from, uint256 id, uint256 value) internal {
        require(from != address(0), "No 0 address");

        console2.log(
            "total supply of shares less than redeemed amount: ",
            totalSupply[id] < value
        );
        console2.log("total supply of shares: %e", totalSupply[id]);

        balanceOf[from][id] -= value;
        totalSupply[id] -= value;
```

Demonstrates that when running the `test_rewards_always_withdrawable` test where a user tries to withdraw their entire initial deposit + yield earned, the following line from the test triggers a revert in the `_burn` function:

```
uint256 totalSupplyOfShares = yieldBox.totalSupply(glpStratAssetId);
        yieldBox.withdraw(
            glpStratAssetId,
            binanceWalletAddr,
            binanceWalletAddr,
            0,
            totalSupplyOfShares
        );
```

Because the rounding up of shares in `YieldBox::_withdrawFungible` which allocates more shares to the user than the `totalSupply`:

```
function _withdrawFungible(
        Asset storage asset,
        uint256 assetId,
        address from,
        address to,
        uint256 amount,
        uint256 share
    ) internal returns (uint256 amountOut, uint256 shareOut) {
        // Effects
        uint256 totalAmount = _tokenBalanceOf(asset);
        if (share == 0) {
            // value of the share paid could be lower than the amount
paid due to rounding, in that case, add a share (Always round up)
            share = amount._toShares(totalSupply[assetId], totalAmount,
true);
           ...
           }
     }
```

This same issue is demonstrated in `test_rewards_always_withdrawable_multiple` where if multiple users deposit and withdraw, if the last user attempts to withdraw the remaining balance of `sGLP` in the strategy (which should correspond to their amount of shares), it also triggers an underflow revert in the ERC1155 `_burn` function due to the following line:

```
// Bob tries to withdraw his amount which should be the remaining balance
of the strategy
        vm.startPrank(bob);
        uint256 amountRemainingInStrategy = sGLP.balanceOf(
            address(glpStrategy)
        );

        yieldBox.withdraw(glpStratAssetId, bob, bob,
amountRemainingInStrategy, 0);
        vm.stopPrank();
```

Recommendation

This issue can be mitigated by ensuring front-end logic prevents this edge case or only allowing sole depositors to withdraw by passing in shares.

Response

INFO-02: Comments for emergencyWithdraw incorrect

Severity

Informational

Description

Comments for `sDaiStrategy::emergencyWithdraw` function states that it "withdraws everything from the strategy" but it actually withdraws everything from sDai to the strategy.

Recommendation

Refactor comments to properly describe function behavior.

Response

INFO-03: Unclear NatSpec for ITapiocaOracle

Severity

Informational

Description

The `ITapiocaOracle` NatSpec states that the get function:

```
@return success if no valid (recent) rate is available, return false else true.
```

Impact

This would imply that calls to get in `_buyGLP` would revert if there IS a valid recent rate returned by the oracle, and only pass if there is NOT a valid recent rate due to the following lines:

```
(success, glpPrice) = wethGlpOracle.get(wethGlpOracleData);
    if (!success) revert Failed();
```

The NatSpec makes understanding the effect in the resulting implementation difficult to discern.

Recommendation

Rephrase NatSpec for clearer definition of function or change return variable name.

Response

INFO-04: More descriptive message for user trying to overdraw their deposit

Severity

Informational

Description

In test `test_user_cant_overdraw` it reverts due to underflow when the user tries to withdraw more than their balance.

Recommendation

Throwing a more descriptive error could allow for better error handling.

Response

INFO-05: Users can withdraw from GlpStrategy when paused

Severity

Informational

Description

`_withdraw` function is missing a check for paused, while `_deposited` contains a pause check.

```
 function _withdraw(address to, uint256 amount) internal override {
        if (amount == 0) revert NotValid();
        _claimRewards(); // Claim rewards before withdrawing
        _buyGlp(); // Buy GLP with WETH rewards

        sGLP.safeApprove(contractAddress, amount);
        ITOFT(contractAddress).wrap(address(this), to, amount); // wrap
the sGLP to tsGLP to `to`, as a transfer
        sGLP.safeApprove(contractAddress, 0);
    }
the sDaiStrategy has a check that prevents withdrawals if the system is
paused, which makes the two strategies inconsistent with each other:

/// @dev burns sDai in exchange of Dai and wraps it into tDai
    function _withdraw(
        address to,
        uint256 amount
    ) internal override nonReentrant {
        if (paused) revert Paused();
            ...
     }
```

Recommendation

Verify if the intended behavior for the `_withdraw` function is to be not pausable.

Response