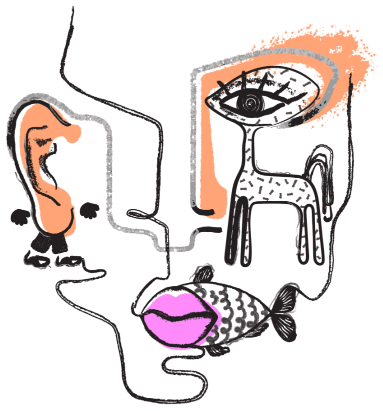
Parachain auction strategies

Audit remediations



Executive Summary

The **Polkastrategies** repository comprises yield-farming strategies into which users can deposit tokens (ETH or stable coins) to earn rewards and raise funds for Polkadot parachain auctions. More information can be found in the **Parachain auction strategies readme**.

Trail of Bits reviewed these contracts and the audit report can be found **on Github**.

Automated Testing and Verification

During their review, Trail of Bits also used automated tested techniques to enhance the coverage of certain areas of the contracts, including Slither and Echidna, augmenting their manual review.

One of the recommendations made based on the Slither output is related to functions that have the onlyOwner modifier.

Slither Scripts

ID	Property	Script
1	In the Polkastrategies codebase, all functions have `onlyOwner` modifiers when necessary.	APPENDIX C

We took their recommendation and we now use the script provided below (extracted from Trail of Bits audit report) to make sure all the methods that require the onlyOwner modifier have the right method signature.

```
from slither import Slither
from slither.core.declarations import Contract
from typing import List

contracts = Slither(".", ignore_compile=True)
```

```python
def _check_access_controls(
      contract: Contract, modifiers_access_controls: List[str],
ACCEPTED_LIST: List[str]
):
      print(f"### Check {contract} access controls")
      no_bug_found = True
      for function in contract.functions_entry_points:
            if function.is_constructor:
                  continue
            if function.view:
                  continue
            if not function.modifiers or (
                  not any((str(x) in modifiers_access_controls) for x in
function.modifiers)
            ):
                  if not function.name in ACCEPTED_LIST:
                        print(f"\t- {function.canonical_name} should
have {modifiers_access_controls} modifier")
                        no_bug_found = False
      if no_bug_found:
            print("\t- No bug found")

safe_functions = ["deposit","withdraw","receive"]

_check_access_controls( .
      contracts.get_contract_from_name("ForceSC"),
      ["onlyOwner"],
      safe_functions
)
_check_access_controls( .
      contracts.get_contract_from_name("ForceSLP"),
      ["onlyOwner"],
      safe_functions
)
_check_access_controls( .
      contracts.get_contract_from_name("HarvestDAI"),
      ["onlyOwner"],
      safe_functions
)
_check_access_controls( .
      contracts.get_contract_from_name("HarvestDAIStableCoin"),
      ["onlyOwner"],
      safe_functions
)
_check_access_controls( .
      contracts.get_contract_from_name("SushiETHSLP"),
      ["onlyOwner"],
      safe_functions
)


_check_access_controls( .
      contracts.get_contract_from_name("SushiSLP"),
      ["onlyOwner"],
      safe_functions
)
```

Findings Summary and Remediations

#	Title	Type	Severity
1	Reward share calculations do not reflect balance changes	Data Validation	High
2	Receipt tokens are not burned when a user withdraws all tokens	Data Validation	Informational
3	Lack of `lockTime` check enables withdrawals during vesting period	Data Validation	Medium

Reward share calculations do not reflect balance changes

To withdraw ETH from a strategy, the contract calls getPendingRewards to calculate the amount of rewards owed to the user. This calculation is based on the current value of the user's net deposits and the time since the user's first deposit.

```
uint256 rewardPerBlockUser =
    rewardPerBlock.mul(block.timestamp.sub(user.joinTimestamp));
uint256 ratio = _getRatio(user.amountfDai, totalDeposits, 18);
```

This method of calculating the reward share could provide an honest user who makes multiple deposits into a strategy with more rewards than the user actually earned, and a malicious user could drain a contract of all rewards.

Exploit scenario

An attacker deposits a small number of tokens into the HarvestDAI strategy. After the lockup period, the attacker deposits a much larger amount of ETH before withdrawing all of his tokens. The attacker receives more rewards than his deposit entitles him to and then repeatedly makes deposits and executes withdrawals to drain all of the rewards from the contract.

Remediation

We have completely redesigned the reward system. We have identified more issues with the old system that was coded and decided to completely change it to a robust implementation.

In the old system, if we take the following example:

- 1 user stakes X amount for 9 blocks
- another user stakes 10*X for 1 block

Imagine the Harvest reward is 1 Farm per X stake per block. After these actions and 10 blocks of time in total, each user should accumulate 10 rewards. Total rewards will be 20.
In the old system, we took the difference between first ever deposit timestamp and current block into account, which in this case is 10 and results in a reward per block of 2. Multiplying this with each user's investment will result in a rewardPerBlock for the first user of 20 and for the second user 2.

Then we did the ratio compared to the total current TVL (which is also wrong as some users might be luckier than others if they decide to withdraw when TVL is lower). In the example above, the first user has a ratio of 1/11 and the second one 10/11. The total final rewards for each user would have been 20 * 1/11 for the first one and 2 * 10/11 for the second one, which is not right.

In the new system, we give exactly the same amount of rewards as Harvest.

```
function _updateRewards(address account) internal {
    if (account != address(0)) {
        UserInfo storage user = userInfo[account];

        uint256 _stored = harvestRewardPool.rewardPerToken();

        user.rewards = _earned(
            user.amountfToken,
            user.userRewardPerTokenPaid,
            user.rewards
        );
        user.userRewardPerTokenPaid = _stored;
    }
}

function _earned(
    uint256 _amountfToken,
    uint256 _userRewardPerTokenPaid,
    uint256 _rewards
) internal view returns (uint256) {
    return
        _amountfToken
            .mul(
            harvestRewardPool.rewardPerToken().sub(_userRewardPerTokenPaid)
        )
            .div(1e18)
            .add(_rewards);
}
```

By updating the reward calculation, we covered Trail of Bits' short term recommendations. We've also added more complex tests in order to check their long term recommendation mentioned in the report at page 16. Besides the ones that check the onlyOwner methods, deposit and withdrawal unit tests were added for all the contracts. An example of such test can be found under test/strategies/harvestsc.ts

Situations covered in these tests are:

```
 - deposit when not blacklisted
 - deposit when blacklisted
 - deposit when cap
 - deposit and then change fee address
 - UI items (total earned, total deposited, eth amount and so on)
 - withdraw when not blacklisted
 - withdraw with receipt tokens
 - withdraw without receipt tokens
 - withdraw when blacklisted
 - withdraw when lock time
```

```
- withdraw without lock time
- multiple withdraw
- entire amount withdraw
- withdraw with fee
- withdraw without fee
- dust and treasury
```

Receipt tokens are not burned when a user withdraws all tokens

Because blacklisted users do not receive receipt tokens for their deposits, the contract checks whether a user is included on the blacklist when executing withdrawals, before attempting to burn the appropriate number of receipt tokens. However, the blacklist check is missing a negation and does not burn the tokens of users attempting to withdraw their entire token balances (or more). Because of the additional bookkeeping performed by the strategy, this flaw does not appear to be exploitable; however, it could be confusing to users.

```
if(user.wasUserBlacklisted){
    receiptToken.burn(msg.sender, user.amountReceiptToken);
    emit ReceiptBurned(msg.sender, user.amountReceiptToken);
    user.amountReceiptToken = 0;
}
```

Exploit Scenario

Alice has deposited tokens into the HarvestDAI strategy. She withdraws all of her tokens and rewards. Because of the missing negation, Alice's receipt tokens are not burned as part of the withdrawal. Alice notices that she still has receipt tokens and becomes confused when she attempts to make another withdrawal but does not receive any more ETH.

Remediation

It was an easy fix. By just adding a negation to that if statement if(!user.wasUserBlacklisted) the issue is solved.

As for the long term recommendation made by Trail of Bits, we already covered that in the previously mentioned issue.

Lack of lockTime check enables withdrawals during vesting period

The polkastrategies contracts are supposed to lock user deposits for 120 days. However, SushiETHSLP fails to check the lock time on withdrawal, allowing users to withdraw assets immediately after depositing them.

The ForceSLP contract checks that the lock time has passed before it allows a user to withdraw tokens and rewards.

The withdraw function in SushiETHSLP lacks such a check.

Instead, the lockTime check was erroneously included in the deposit function in the SushiETHSLP contract.

Exploit Scenario

Eve deposits ETH into the SushiETHSLP contract. As the lock time is not checked in the withdraw function, she can bypass the vesting period and withdraw her funds immediately.

Remediation

To clarify, the 120 days is the default lock time strategies are deployed with, but it can be set to 0 by the owner. To check what's the current lockTime please see our official documentation https://www.composable.finance/

The fix is again an easy one and it was just a mistake created by wrongly adding that part in the deposit method instead of withdraw.

Code maturity

During their review, Trail of Bits marked the solution with a low score for the code maturity section due to the fact that we were updating the code while they were looking at it and also because contracts had duplicated code.

Remediation

We've completely refactored the contracts, without changing the initial flow reviewed by them. Initially, there was no StrategyBase, HarvestBase, HarvestETHBase, HarvestSCBase, SushiBase, SushiETHBase, SushiSLPBase contracts and associated interfaces so you can imagine the amount of duplicate code there was in each contract.

We've also introduced the TimelockOwner contract to increase the confidence our community has in us. We won't be able to change contracts' properties right away. There's a minimum grace period of 24 hours before anything can be changed.