PoolTogether — Sushi and Yearn V2 Yield Sources

Date	May 2021
Auditors	Heiko Fisch, Sergii Kravchenko

1 Executive Summary

This report presents the results of our engagement with **PoolTogether** to review their **Sushi and Yearn V2 yield sources**.

The review was conducted over one week, from **May 24** to **May 28** by **Heiko Fisch** and **Sergii Kravchenko**. A total of 10 person-days were spent.

2 Scope

Our review focused on the commit hashes `ccaf9d73f8cf5c0c41f6e4d640d9b186c51bc3ce` for the Sushi yield source and `a34857f1555908a6263d2ebd189f0cb40e1858da` for the Yearn V2 yield source. The list of files in scope can be found in the Appendix.

As per the client's request, higher priority was given to the Sushi yield source. Due to the limited time available, the review of the Yearn V2 yield source had to remain superficial. It should also be noted that the contracts were reviewed in isolation, without a thorough understanding of the rest of the system.

3 Findings

Each issue has an assigned severity:

- Minor issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
- Medium issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- Major 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.
- Critical issues are directly exploitable security vulnerabilities that need to be fixed.

3.1 Yearn: Re-entrancy attack during deposit Critical

Description

During the deposit in the `supplyTokenTo` function, the token transfer is happening after the shares are minted and before tokens are deposited to the yearn vault:

code/pooltogether-yearnv2-yield-source/contracts/yield-source/YearnV2YieldSource.sol:L117-L128

```
function supplyTokenTo(uint256 _amount, address to) override external {
    uint256 shares = _tokenToShares(_amount);

    _mint(to, shares);

    // NOTE: we have to deposit after calculating shares to mint
    token.safeTransferFrom(msg.sender, address(this), _amount);

    _depositInVault();

    emit SuppliedTokenTo(msg.sender, shares, _amount, to);
}
```

If the token allows the re-entrancy (e.g., ERC-777), the attacker can do one more transaction during the token transfer and call the `supplyTokenTo` function again. This second call will be done with already modified shares from the first deposit but non-modified token balances. That will lead to an increased amount of shares minted during the `supplyTokenTo`. By using that technique, it's possible to steal funds from other users of the contract.

Recommendation

Have the re-entrancy guard on all the external functions.

3.2 Yearn: Partial deposits are not processed properly Major

Description

The deposit is usually made with all the token balance of the contract:

code/pooltogether-yearnv2-yield-source/contracts/yield-source/YearnV2YieldSource.sol:L171-L172

```
// this will deposit full balance (for cases like not enough room in Vault)
return v.deposit();
```

The Yearn vault contract has a limit of how many tokens can be deposited there. If the deposit hits the limit, only part of the tokens is deposited (not to exceed the limit). That case is not handled properly, the shares are minted as if all the tokens are accepted, and the "change" is not transferred back to the caller:

code/pooltogether-yearnv2-yield-source/contracts/yield-source/YearnV2YieldSource.sol:L117-L128

```
function supplyTokenTo(uint256 _amount, address to) override external {
    uint256 shares = _tokenToShares(_amount);

    _mint(to, shares);

    // NOTE: we have to deposit after calculating shares to mint
    token.safeTransferFrom(msg.sender, address(this), _amount);

    _depositInVault();

    emit SuppliedTokenTo(msg.sender, shares, _amount, to);
}
```

Recommendation

Handle the edge cases properly.

3.3 Sushi: `redeemToken` redeems less than it should Medium

Description

The `redeemToken` function takes as argument the amount of SUSHI to redeem. Because the `SushiBar`'s `leave` function – which has to be called to achieve this goal – takes an amount of xSUSHI that is to be burned in exchange for SUSHI, `redeemToken` has to compute the amount of xSUSHI that will result in a return of as many SUSHI tokens as were requested.

code/sushi-pooltogether/contracts/SushiYieldSource.sol:L62-L87

```
/// @notice Redeems tokens from the yield source from the msg.sender, it burn yield bearing tokens and return token to the sender.
/// @param amount The amount of `token()` to withdraw.  Denominated in `token()` as above.
/// @return The actual amount of tokens that were redeemed.
function redeemToken(uint256 amount) public override returns (uint256) {
    ISushiBar bar = ISushiBar(sushiBar);
    ISushi sushi = ISushi(sushiAddr);

    uint256 totalShares = bar.totalSupply();
    uint256 barSushiBalance = sushi.balanceOf(address(bar));
    uint256 requiredShares = amount.mul(totalShares).div(barSushiBalance);

    uint256 barBeforeBalance = bar.balanceOf(address(this));
    uint256 sushiBeforeBalance = sushi.balanceOf(address(this));

    bar.leave(requiredShares);

    uint256 barAfterBalance = bar.balanceOf(address(this));
    uint256 sushiAfterBalance = sushi.balanceOf(address(this));

    uint256 barBalanceDiff = barBeforeBalance.sub(barAfterBalance);
    uint256 sushiBalanceDiff = sushiAfterBalance.sub(sushiBeforeBalance);

    balances[msg.sender] = balances[msg.sender].sub(barBalanceDiff);
    sushi.transfer(msg.sender, sushiBalanceDiff);
    return (sushiBalanceDiff);
}
```

Because the necessary calculations involve division and amounts have to be integral values, it is usually not possible to get the *exact* amount of SUSHI tokens that were requested. More precisely, let `a` denote the total supply of xSUSHI and `b` the `SushiBar`'s balance of SUSHI at a certain point in time. If the `SushiBar`'s `leave` function is supplied with `x` xSUSHI, then it will transfer `floor(x * b / a)` SUSHI. (We assume throughout this discussion that the numbers involved are small enough such that no overflow occurs and that `a` and `b` are not zero.)
Hence, if `y` is the amount of SUSHI requested, it would make sense to call `leave` with the biggest number `x` that satisfies `floor(x * b / a) <= y` or the smallest number `x` that satisfies `floor(x * b / a) >= y`. Which of the two is "better" or "correct" needs to be specified, based on the requirements of the caller of `redeemToken`. It seems plausible, though, that the first variant is the one that makes more sense in this context, and the current implementation of `redeemToken` supports this hypothesis. It calls `leave` with `x1 := floor(y * a / b)`, which gives us `floor(x1 * b / a) <= y`. However, `x1` is not necessarily the *biggest* number that satisfies the relation, so the caller of `redeemToken` might end up with less SUSHI than they could have gotten while still not exceeding `y`.

The correct amount to call `leave` with is `x2 := floor((y * a + a - 1) / b) = max { x | floor(x * b / a) <= y }`. Since `|x2 - x1| <= 1`, the difference in SUSHI is at most `floor(b / a)`. Nevertheless, even this small difference might subvert fairly reasonable expectations. For example, if someone queries `balanceOfToken` and immediately after that feeds the result into `redeemToken`, they might very well expect to redeem exactly the given amount and not less; it's their current balance, after all. However, that's not always the case with the current implementation.

Recommendation

Calculate `requiredShares` based on the formula above (`x2`). We also recommend dealing in a clean way with the special cases `totalShares == 0` and `barSushiBalance == 0`.

3.4 Sushi: `balanceOfToken` underestimates balance Medium

Description

The `balanceOfToken` computation is too pessimistic, i.e., it can underestimate the current balance slightly.

code/sushi-pooltogether/contracts/SushiYieldSource.sol:L29-L45

```
/// @notice Returns the total balance (in asset tokens).  This includes the deposits and interest.
/// @return The underlying balance of asset tokens
function balanceOfToken(address addr) public override returns (uint256) {
    if (balances[addr] == 0) return 0;
    ISushiBar bar = ISushiBar(sushiBar);

    uint256 shares = bar.balanceOf(address(this));
    uint256 totalShares = bar.totalSupply();

    uint256 sushiBalance =
        shares.mul(ISushi(sushiAddr).balanceOf(address(sushiBar))).div(
            totalShares
        );
    uint256 sourceShares = bar.balanceOf(address(this));

    return (balances[addr].mul(sushiBalance).div(sourceShares));
}
```

First, it calculates the amount of SUSHI that "belongs to" the yield source contract (`sushiBalance`), and then it determines the fraction of *that amount* that would be owed to the address in question. However, the "belongs to" above is a purely theoretical concept; it never happens that the yield source contract as a whole redeems and then distributes that amount among its shareholders; instead, if a shareholder redeems tokens, their request is passed through to the `SushiBar`. So in reality, there's no reason for this two-step process, and the holder's balance of SUSHI is more accurately computed as `balances[addr].mul(ISushi(sushiAddr).balanceOf(address(sushiBar))).div(totalShares)`, which can be greater than what `balanceOfToken` currently returns. Note that this is the amount of SUSHI that `addr` could withdraw directly from the `SushiBar`, based on their amount of shares. Observe also that if we sum these numbers up over all holders in the yield source contract, the result is smaller than or equal to `sushiBalance`. So the sum still doesn't exceed what "belongs to" the yield source contract.

Recommendation

The `balanceOfToken` function should use the formula above.

3.5 Yearn: Redundant `approve` call Minor

Description

The approval for token transfer is done in the following way:

code/pooltogether-yearnv2-yield-source/contracts/yield-source/YearnV2YieldSource.sol:L167-L170

```
if(token.allowance(address(this), address(v)) < token.balanceOf(address(this))) {
    token.safeApprove(address(v), 0);
    token.safeApprove(address(v), type(uint256).max);
}
```

Since the approval will be equal to the maximum value, there's no need to make zero-value approval first.

Recommendation

Change two `safeApprove` to one regular `approve` with the maximum value.

3.6 Sushi: Some state variables should be `immutable` and have more specific types Minor

Description

The state variables `sushiBar` and `sushiAddr` are initialized in the contract's constructor and never changed afterward.

code/sushi-pooltogether/contracts/SushiYieldSource.sol:L12-L21

```
contract SushiYieldSource is IYieldSource {
    using SafeMath for uint256;
    address public sushiBar;
    address public sushiAddr;
    mapping(address => uint256) public balances;

    constructor(address _sushiBar, address _sushiAddr) public {
        sushiBar = _sushiBar;
        sushiAddr = _sushiAddr;
    }
```

They should be `immutable`; that would save some gas and make it clear that they won't (and can't) be changed once the contract has been deployed.
Moreover, they would better have more specific interface types than `address`, i.e., `ISushiBar` for `sushiBar` and `ISushi` for `sushiAddr`. That would be safer and make the code more readable.

Recommendation

Make these two state variables `immutable` and change their types as indicated above. Remove the corresponding explicit type conversions in the rest of the contract, and add explicit conversions to type `address` where necessary.

3.7 Sushi: Unnecessary balance queries Minor

Description

In function `redeemToken`, `barBalanceDiff` is always the same as `requiredShares` because the `SushiBar`'s `leave` function burns exactly `requiredShares` xSUSHI.

code/sushi-pooltogether/contracts/SushiYieldSource.sol:L73-L84

```
uint256 barBeforeBalance = bar.balanceOf(address(this));
uint256 sushiBeforeBalance = sushi.balanceOf(address(this));

bar.leave(requiredShares);

uint256 barAfterBalance = bar.balanceOf(address(this));
uint256 sushiAfterBalance = sushi.balanceOf(address(this));

uint256 barBalanceDiff = barBeforeBalance.sub(barAfterBalance);
uint256 sushiBalanceDiff = sushiAfterBalance.sub(sushiBeforeBalance);

balances[msg.sender] = balances[msg.sender].sub(barBalanceDiff);
```

Recommendation

Use `requiredShares` instead of `barBalanceDiff`, and remove the unnecessary queries and variables.

3.8 Sushi: Unnecessary function declaration in interface Minor

Description

The `ISushiBar` interface declares a `transfer` function.

code/sushi-pooltogether/contracts/ISushiBar.sol:L5-L17

```
interface ISushiBar {
    function enter(uint256 _amount) external;

    function leave(uint256 _share) external;

    function totalSupply() external view returns (uint256);

    function balanceOf(address account) external view returns (uint256);

    function transfer(address recipient, uint256 amount)
        external
        returns (bool);
}
```

However, this function is never used, so it could be removed from the interface. Other functions that the `SushiBar` provides but are not used (`approve` , for example) aren't part of the interface either.

Recommendation

Remove the `transfer` declaration from the `ISushiBar` interface.

Appendix 1 - Files in Scope

This audit covered the following files:

A.1.1 Sushi Yield Source

File	SHA-1 hash
contracts/SushiYieldSource.sol	afa8b6083c6956c82ffe3b16c2f67ad86929eb75
contracts/ISushiBar.sol	86e92592551dd788e9936f1383f0920aee0501c8
contracts/ISushi.sol	3e4001370481f4f3fd1b235edca4873e47973647

A.1.2 Yearn V2 Yield Source

File	SHA-1 hash
contracts/yield-source/YearnV2YieldSource.sol	33f041474bdf367549f51c6d759e73918ea8ff7c
contracts/yield-source/YearnV2YieldSourceProxyFactory.sol	8a69d3c9d035a15b2d5f751f976f832162eb37c1

Appendix 2 - Disclosure

areas beyond specified code that could present security risks. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. In some instances, we may perform penetration testing or infrastructure assessments depending on the scope of the particular engagement.

CD makes the Reports available to parties other than the Clients (i.e., “third parties”) – on its website. CD hopes that by making these analyses publicly available, it can help the blockchain ecosystem develop technical best practices in this rapidly evolving area of innovation.

LINKS TO OTHER WEB SITES FROM THIS WEB SITE You may, through hypertext or other computer links, gain access to web sites operated by persons other than ConsenSys and CD. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites’ owners. You agree that ConsenSys and CD are not responsible for the content or operation of such Web sites, and that ConsenSys and CD shall have no liability to you or any other person or entity for the use of third party Web sites. Except as described below, a hyperlink from this web Site to another web site does not imply or mean that ConsenSys and CD endorses the content on that Web site or the operator or operations of that site. You are solely responsible for determining the extent to which you may use any content at any other web sites to which you link from the Reports. ConsenSys and CD assumes no responsibility for the use of third party software on the Web Site and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.

TIMELINESS OF CONTENT The content contained in the Reports is current as of the date appearing on the Report and is subject to change without notice. Unless indicated otherwise, by ConsenSys and CD.