YearnV2: our gotcha notes

Martinet Lee (https://twitter.com/martinetlee) & Jun-You Liu (https://twitter.com/orbxball)

TL;DR

- We have notified Andre before publishing this analysis.
- **Your WAIFU is SAFU:** None of the existing contracts are subject to the exploits mentioned below.
- The analysis is based on the set of contracts related to the yCurve yVault.
- “Poisoning Baby Vault” is valid but it only affects new vaults.
- The implementation in yCurve yVault/Strategy pair does not consider the case if `Vault Token` is not `Want Token`. Specifically, the functions that are related to balance.
- Future developers should be aware of these issues when implementing Strategies.
- As it is not guaranteed to be correct by the interface/protocol, whenever a new strategy is being proposed, we recommend that the governance check:
  - if `Vault Token` and `Want Token` are the same
  - how the functions related to balance are being implemented.

Intro

Yearn is a fascinating system that aims to provide users with excellent yields. To stay on top of the game, YearnV2 is extensible and allows new strategies to be deployed and applied.

We were fascinated by the concept and decided to do some research into the contracts.

In the `yVault / Controller / Strategy`, there are three kinds of tokens:

1. Vault Token: the kind of token that users would deposit in `yVault`. This will be converted into `Want Token`
2. Want Token: during deposit, this is the token that the `Strategy` uses to obtain the target yield. During harvest, `Strategy` will sell all the pool reward tokens into want tokens.
3. Pool Reward Token: the token that is being harvested.

One of the most important things in a protocol is how to distribute profits to users. In YearnV2, when users deposit funds with `vault tokens`, they would get some shares. As users withdraw, they would burn the shares and supposedly get their tokens and profits back. Thus, we looked

into the functions that are related to share calculation.

We identified 4 issues. One of which is valid but it only affects new vaults. Others are related to the fact that the implementation of the yCurve yVault/Strategy pair does not consider the case if `Vault Token` is not `Want Token`.

According to Andre, this is a design choice and different strategies should implement those functions differently.

However, we still wanted to leave the analysis here to show that what would happen **IF** the `Vault Token` and `Want Token` are different in value and the functions are not modified.

Issue 1: Poisoning the Baby Vault

Let's take a look at the `deposit()` below. According to our understanding, the `_pool = balance()` should have represented the amount of token this sytem holds. (p.s. we would later show that `balance()` here is also incorrect.)

So, from the line `if(_pool == 0)`, we can see that the contract assumes that:

> if there is some balance in the system, then it is not a new vault anymore.

```
function deposit(uint _amount) external {
    uint _pool = balance();
    token.safeTransferFrom(msg.sender, address(this), _amount);
    uint shares = 0;
    if (_pool == 0) {
      shares = _amount;
    } else {
      shares = (_amount.mul(totalSupply())).div(_pool);
    }
    _mint(msg.sender, shares);
}
```

However, this assumption can be easily broken.

As soon as a new vault is being deployed, an attacker can poison this baby vault by either sending some `Want Token` to the underlying `Strategy` or sending some `Vault Token` to the `yVault`. Later when a user comes in and tries to `deposit()`, it will execute the `else` branch:

- `shares = (_amount.mul(totalSupply())).div(_pool);`
  Note that the `totalSupply()` here is the number of shares minted before and would be `0`. Thus, the `shares` would be `0` as well.

A poisoned vault will not give users any share and consequently the user's funds will be locked inside forever

Andre's Response:

> I agree on the exploit, but the exploit is probability wise very unlikely, since;
>
> 1. The strategy needs to be deployed
> 2. The strategy needs to be linked to the 'want' in controller
> 3. The vault needs to be deployed
> 4. There needs to be no deposits in the vault
> 5. The vault needs to be added in the controller
>
> And then sending something to either vault or controller directly will brick the totalSupply calculation. This is more to do with the rollout system than anything else. Currently it can't actually do any damage if you send funds to the contract

The apples and oranges issues

It is very important to align the units of the interface together. NASA once lost a spacecraft, only because some parts of the system used English measurement and other parts used metric system. We have identified several ways of how the system would fail if the `Vault Token` were to be different from the `Want Token` .

We want to stress again that the issues described here are NOT present in the current yCurve Vault (as of `August 4th 12:00pm EST`). The `Vault Token` and `Want Token` are the same in the present version.

Andre's Response:

> Yes, they can be different, but if they are different, that specific strategy reports `balanceOf` differently. Balance of isn't copy/paste for all strategies

Still, we think it would be beneficial for the community to highlight these issues for future developers. Be extra careful when two tokens are different and make sure you have tweaked the functions.

Issue 2: Lost in translation

When `Vault Token` is different from `Want Token` , it is possible that the user will not receive anything when he withdraws.

Looking at the `withdraw()` in `yVault.sol`, we see in the `if (b < r) { ... }` that the contract attempts to withdraw from the underlying strategy if it does not have enough `Vault Token` in the current Vault contract.

```
function withdraw(uint _shares) external {
    uint r = (balance().mul(_shares)).div(totalSupply());
    _burn(msg.sender, _shares);

    // Check balance
    uint b = token.balanceOf(address(this));
    if (b < r) {
        uint _withdraw = r.sub(b);
        Controller(controller).withdraw(address(token), _withdraw);
        uint _after = token.balanceOf(address(this));
        uint _diff = _after.sub(b);
        if (_diff < _withdraw) {
            r = b.add(_diff);
        }
    }

    token.safeTransfer(msg.sender, r);
}
```

Now let's take a closer look into how this line works:

- `Controller(controller).withdraw(address(token), _withdraw);`

This calls the `withdraw()` in `Controller` with the `token` being the `Vault Token`.

```
// In Controller.sol

function withdraw(address _token, uint _amount) public {
    require(msg.sender == vaults[_token], "!vault");
    Strategy(strategies[_token]).withdraw(_amount);
}
```

and passes on to `withdraw()` in the `Strategy`. Note that:

1. **The vault's address is being checked against vaults[`Vault Token`]**
2. **`_token` is not passed into `Strategy`.**

```
// In StrategyYfii.sol

// Withdraw partial funds, normally used with a vault withdrawal
function withdraw(uint _amount) external {
    require(msg.sender == controller, "!controller");
    uint _balance = IERC20(want).balanceOf(address(this));
    if (_balance < _amount) {
        _amount = _withdrawSome(_amount.sub(_balance));
        _amount = _amount.add(_balance);
    }

    address _vault = Controller(controller).vaults(address(want));
    require(_vault != address(0), "!vault"); // additional protection so we don't bur
    IERC20(want).safeTransfer(_vault, _amount);
}
```

If `Vault Token` is not the same as `Want Token`, There are at least two issues here:

1. `_vault` is obtained with **vaults[`Want Token`]**, which is different from **vaults[`Vault Token`]**.
2. the last line of this function transfers `Want Token` to the `_vault`.

Consequently, the original `yVault` would NOT receive any token and the wrong `yVault` would receive some `Want Tokens`.

Issue 3: Incorrect counting: Adding apples and oranges together

The function `balance()` in `yVault.sol` is used in both `deposit()` and `withdraw()` to calculate users' shares. This is how it looks like:

```
// In yVault.sol
function balance() public view returns (uint) {
    return token.balanceOf(address(this))
           .add(Controller(controller).balanceOf(address(token)));
}
```

We can see that it adds two balances together. But what tokens do they represent respectively? After tracing the code, we concluded that:

- `Controller(controller).balanceOf(address(token))` represents the amount of **`Want Token`** that the `Strategy` contract holds.
- `token.balanceOf(address(this))` represents the amount of **`Vault Token`** that this contract holds.

Ooops! Seems like here we added two different things here.

Issue 4: Dilution exploit

Based on the `Issue 3`, there is a more complicated and profitable exploit which we call the "dilution exploit". An attacker can transfer `Want Token` to the `Strategy` contract and incorrectly increase the `balance` of the `Vault` contract. **Whoever deposits after the transfer would get less shares than one is supposed to and thus "diluted".** While the victims would not lose their funds or shares, they would get less shares than expected and thus earn less profit in the future.

The profits that they missed are distributed evenly to previous share holders and thus the attacker would gain more profit.

If this extra profit is greater than the value of `Want token` the attacker deposits into strategy contract, the attacker is incentivized to perform the attack. Again, we stress that this exploit only works when the `Vault token:Want token != 1:1`. Thus, the current `StrategyYfii` and `StrategyYffi` are not vulnerable.

We provide a simple example with two scenarios to show this exploit. The two scenarios described below are based on the following settings:

- `Vault Token` is worth `1000` times the `Want Token`
- The functions related to balance are not updated accordingly.

Normal Transaction

Vault Token Price **(VT)**	1000					
Want Token Price **(WT)**	1					
	totalSupply (shares)	**balance()**	**Alice Shares**	**Bob Shares**	**Alice can Withdraw**	**Bob can Withdraw**
start	0	0	0	0		
Alice deposits 100 VT	100.00	100.00	100.00	0.00	100.00	0.00
Alice deposits 100 VT	200.00	200.00	200.00	0.00	200.00	0.00
Bob deposits 10000 VT	10,200.00	10,200.00	200.00	**10,000.00**	200.00	10,000.00
gained 100000 in profit	10,200.00	110,200.00	200.00	10,000.00	2,160.78	108,039.22

Malicious transfer into the Strategy Contract

Vault Token Price **(VT)**	1000					
Want Token Price **(WT)**	1					
	totalSupply (shares)	**balance()**	**Alice Shares**	**Bob Shares**	**Alice can Withdraw**	**Bob can Withdraw**
start	0	0	0	0		
Alice deposits 100 VT	100.00	100.00	100.00	0.00	100.00	0.00
Alice transfers 100000 WT to Strategy (same value as 100 VT)	100.00	**100,100.00**	100.00	0.00	100,100.00	0.00
Bob deposits 10000 VT	109.99	110,100.00	100.00	**9.99**	100,100.00	10,000.00
gained 100000 in profit	109.99	210,100.00	100.00	9.99	**191,017.35**	**19,082.65**

We can see that Bob only gets `9.99` shares as opposed to the `10000` shares during the normal transaction. While Bob will not lose his original deposit, his profits are partially taken away by the attacker Alice.

Maths!

For a more generalized version of this exploit, formal mathematical calculation is provided in this link (https://www.overleaf.com/read/hxdrzfvjmvkb) for better latex support. It provides the precise equation to check if this type of exploit is incentivized to exist.

More scared than hurt : `balance = balanceOfCurve + balanceOfYfii` ?!

- This is not an issue, but it did scare us :P

When reading the contract `StrategyYfii` (https://etherscan.io/address/0xBE197E668D13746BB92E675dEa2868FF14dA0b73#code), we noticed something odd when it is calculating the balance:

```
function balanceOf() public view returns (uint) {
    return balanceOfCurve()
           .add(balanceOfYfii());
}
```

Adding balance from two different coins seems unintuitive at best, and is possibly an error. Here are the two functions:

```
function balanceOfCurve() public view returns (uint) {
    return IERC20(want).balanceOf(address(this));
}
```

`balanceOfCurve()` is pretty straghtforward and represents how much `yCRV` the strategy contract owns.

```
function balanceOfYfii() public view returns (uint) {
    return Yfii(pool).balanceOf(address(this));
}
```

At first sight, it appeared to us that it is an `ERC20` and reperesents how much `Yfii` that the strategy contract owns. We've contacted Andre to confirm about this issue, and thankfully, it is actually not the case.

So, `balanceOfYfii()` is less intuitive – The `balanceOf()` here is actually how much `yCRV` has the strategy contract staked into the yearnRewards pool (https://etherscan.io/address/0xb81D3cB2708530ea990a287142b82D058725C092#code).

```
function balanceOf(address account) public view returns (uint256) {
    return _balances[account];
}

function stake(uint256 amount) public {
    _totalSupply = _totalSupply.add(amount);
    _balances[msg.sender] = _balances[msg.sender].add(amount);
    y.safeTransferFrom(msg.sender, address(this), amount);
}
```

Thus, the `StrategyYfii` acts correctly and we are safe here.

References

- yVault yCRV: https://etherscan.io/address/0x5dbcf33d8c2e976c6b560249878e6f1491bca25c#code (https://etherscan.io/address/0x5dbcf33d8c2e976c6b560249878e6f1491bca25c#code)
- Controller: https://etherscan.io/address/0x31317f9a5e4cc1d231bdf07755c994015a96a37c#code (https://etherscan.io/address/0x31317f9a5e4cc1d231bdf07755c994015a96a37c#code)
- Yfii strategy: https://etherscan.io/address/0x382185F3ea9268E65Bb16f81de6b4e725134ED72#code

(https://etherscan.io/address/0x382185F3ea9268E65Bb16f81de6b4e725134ED72#code)

- Yffi strategy: https://etherscan.io/address/0xBE197E668D13746BB92E675dEa2868FF14dA0b73#code (https://etherscan.io/address/0xBE197E668D13746BB92E675dEa2868FF14dA0b73#code)
- Yfii Rewards (staking yCRV to earn Yfii): https://etherscan.io/address/0xb81D3cB2708530ea990a287142b82D058725C092#code (https://etherscan.io/address/0xb81D3cB2708530ea990a287142b82D058725C092#code)