Pino Security Review

Pashov Audit Group

Conducted by: pashov

October 3rd, 2023

Contents

1. About pashov

Krum Pashov, or **pashov**, is an independent smart contract security researcher. Having found numerous security vulnerabilities in various protocols, he does his best to contribute to the blockchain ecosystem and its protocols by putting time and effort into security research & reviews. Check his previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where I try to find as many vulnerabilities as possible. I can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **Pino** protocol was done by **pashov**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Pino

The Pino protocol is one that serves as a proxy between the user and popular DeFi protocols (Aave, Curve, Compound, Uniswap etc). By adding a `Multicall`, `Permit` and other functionalities on top of integrations with protocols, the interaction of users with them is simplified.

More docs

Observations

Anybody can make any of the protocol's contracts approve him as a spender of tokens held by it.

Anybody can sweep any ERC20 from any contract.

The protocol shouldn't hold any balances between transactions apart from ETH balance from fees, which only the `owner` can withdraw.

Privileged Roles & Actors

The only privileged role here is the `BaseProtocolProxy` owner, it holds the `owner` role for all contracts and can withdraw ETH balances or update addresses in `Aave` and `Swap` contracts.

5. Risk Classification

Severity	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash **-** **e11214c8eb52fd967d496888999b0327a8f28a93**

fixes review commit hash **-** **7596a1dab98f712c5fdf2499d091ddc2c8a6bed7**

Scope

The following smart contracts were in scope of the audit:

- `base/BaseProtocolProxy`
- `base/Multicall`
- `base/Permit`
- `protocols/v2/Balancer`
- `protocols/v2/Curve`
- `protocols/v2/Uniswap`
- `protocols/Aave`
- `protocols/Compound`
- `protocols/Invest`
- `protocols/Swap`
- `interfaces/**`

7. Executive Summary

Over the course of the security review, pashov engaged with Pino to review Pino. In this period of time a total of **7** issues were uncovered.

Protocol Summary

Protocol Name	Pino
Date	October 3rd, 2023

Findings Count

Severity	**Amount**
Critical	2
Medium	3
Low	2
Total Findings	**7**

Summary of Findings

ID	Title	Severity	Status
[C-01]	Calling Curve::withdraw will likely result in users losing ETH	Critical	Resolved
[C-02]	The fee mechanism is not enforced	Critical	Resolved
[M-01]	Integration with Curve is flawed	Medium	Resolved
[M-02]	Calls to methods with nonETHReuse modifier can be force reverted	Medium	Resolved
[M-03]	The contracts owner can maliciously front-run users	Medium	Resolved
[L-01]	The Uniswap functionality does not have a deadline parameter	Low	Resolved
[L-02]	Fake Curve pools and Compound tokens can be used	Low	Resolved

8. Findings

8.1. Critical Findings

[C-01] Calling `Curve::withdraw` will likely result in users losing ETH

Severity

Impact: High, as users can lose ETH value

Likelihood: High, as all >75 Curve pools that have ETH are problematic

Description

The `Curve::withdraw` method removes liquidity from a pool to burn LP tokens and receive the underlying assets, after which a user can sweep them to his wallet. The problem with this is that some pools use ETH as an underlying asset and the `remove_liquidity` method will send back ETH to the caller. When this is the case the ETH will be stuck in the contract (the `owner` can withdraw it for himself) since the user has no way to sweep the ETH balance of the contracts. This is handled in `withdrawOneCoinI` and `withdrawOneCoinU` but not in `withdraw`.

Below you can see a runnable Proof of Concept unit test, add this to the `curve_pool.test.ts` file in its `Remove Liquidity` suite as a last test to run it:

```
it('User can not claim his ETH liquidity', async () => {
      const { contract, sign } = await loadFixture(deploy);

      const POOL = '0xdc24316b9ae028f1497c275eb9192a3ea0f67022';
      const POOL_TOKEN = '0x06325440d014e39736583c165c2963ba99faf14e';

      const proxyFee = 5n;
      const ethAmount = 1n * 10n ** 18n;

      const amounts = [ethAmount, 0]; // eth - steth

      const poolToken = await ethers.getContractAt(
        'IERC20',
        POOL_TOKEN,
      );

      const poolTokenBalanceBefore = await poolToken.balanceOf(
        account.address,
      );

      const addTx = await contract.populateTransaction.deposit(
        amounts,
        0,
        POOL,
        proxyFee,
      );
      const sweepTx = await contract.populateTransaction.sweepToken(
        POOL_TOKEN,
        account.address,
      );

      await contract.multicall([addTx.data, sweepTx.data], 0, {
        value: ethAmount + proxyFee,
      });

      const poolTokenBalanceAfter = await poolToken.balanceOf(
        account.address,
      );

      expect(poolTokenBalanceAfter).to.gt(poolTokenBalanceBefore);

      await poolToken.approve(PERMIT2_ADDRESS, constants.MaxUint256);

      const { permit, signature } = await sign(
        {
          amount: poolTokenBalanceAfter,
          token: POOL_TOKEN,
        },
        contract.address,
      );

      const approveTx =
        await contract.populateTransaction.approveToken(POOL_TOKEN, [
          POOL,
        ]);
      const permitTx =
        await contract.populateTransaction.permitTransferFrom(
          permit,
          signature,
        );

      const removeLiquidityTx =
        await contract.populateTransaction.withdraw(
          poolTokenBalanceAfter,
          [0, 0],
          POOL,
        );
      // remove liquidity then try with unwrapping, as if the contract wrapped
      // the received ETH in WETH
      const unwrapTx = await contract.populateTransaction.unwrapWETH9(
        account.address,
      );

      let ethBalanceBefore = await account.getBalance();
      await contract.multicall(
        [
          approveTx.data,
          permitTx.data,
          removeLiquidityTx.data,
          unwrapTx.data,
        ],
        0,
      );

      let ethBalanceAfter = await account.getBalance();
      // balance before is actually more than current because of paying gas - it
      // should have been ~1 ETH more, because of `ethAmount`
      expect(ethBalanceBefore).to.gt(ethBalanceAfter);
    });
  });
```

Recommendations

Use the handling as in `withdrawOneCoinI` and `withdrawOneCoinU`, namely wrapping the ETH balance to `WETH`, which the user can sweep to his wallet.

[C-02] The fee mechanism is not enforced

Severity

Impact: High, as the protocol can lose potential yield in the form of fees

Likelihood: High, as users can craft such transactions in a permissionless way

Description

The codebase is using a fee mechanism where the users pay a fee for using some functionality. An example where this is done is the `Compound::depositETHV2` method, as we can see here:

```
function depositETHV2(
      address_recipient,
      uint256_proxyFeeInWei
    ) external payable nonETHReuse {
        address _cEther = address(cEther);

        ICEther(_cEther).mint{value: msg.value - _proxyFeeInWei}();
    ....
    ....
```

The problem with this approach is that the value of the fee is controlled by the user through the `_proxyFeeInWei` argument, meaning he can always send 0 value to it so he doesn't pay any fees.

Recommendations

Rearchitecture the fees approach so that a fee can be enforced on users, for example by using a sensible admin set value for it.

8.2. Medium Findings

[M-01] Integration with `Curve` is flawed

Severity

Impact: Low, as users won't lose funds but the protocol's contract might need new implementation and redeployment

Likelihood: High, as users can't use a big part of Curve pools

Description

Currently, the `Curve` methods `deposit` and `withdraw` are hardcoding the number of underlying tokens in a Curve pool to be exactly two. This is incorrect, as some pools have three or more underlying tokens and with the current implementations users can't make proxy calls to them, which limits the functionality of the protocol.

Recommendations

Change the methods in `Curve` so that they can work for different counts of underlying tokens in a pool, make sure to do this with a proper validations.

[M-02] Calls to methods with `nonETHReuse` modifier can be force reverted

Severity

Impact: Medium, as the user will get its transaction reverted, but it can be replayed through a `Multicall` call

Likelihood: Medium, as it can only happen when there is a direct call to such methods, which isn't the usual way to use the app

Description

In the contracts under `protocols/` we see a good amount of their methods having the `nonETHReuse` modifier. The modifier code calls the following method:

```
function _nonReuseBefore() private {
        // On the first call to nonETHReuse, _status will be NOT_ENTERED
        if (_status == ENTERED) {
            revert EtherReuseGuardCall();
        }

        // Any calls to nonETHReuse after this point will fail
        _status = ENTERED;
    }
```

This code means that if a method with the modifier is called two times in a row, the second call would be reverted. The only way to "unlock" the contracts is through a `Multicall::multicall` call, which sets `_status = NOT_ENTERED;`. Because of this, the following attack can be executed:

1. Alice wants to directly (not through `Multicall`) call a method that has the `nonETHReuse` modifier, for example `Compound::depositWETHV2`
2. Bob sees Alice's transaction and front-runs it with another direct call to a method that has the `nonETHReuse` modifier, for example `Compound::depositETHV2`
3. Since Bob's transaction was executed first, now we have `_status == ENTERED`, which would revert Alice's transaction

Recommendations

Consider forbidding direct calls to methods and force them to be done through `Multicall`.

[M-03] The contracts owner can maliciously front-run users

Severity

Impact: High, as it can result in a loss of funds for users

Likelihood: Low, as it requires a malicious or compromised owners

Description

The `Swap` and `Aave` contracts have the `setNewAddresses` functionality, which can be only called by the contracts `owner`. If users send `Multicall` calls to the protocol and are using either the `Swap` or `Aave` contracts, the `owner` can front-run their call by updating the addresses to his own controlled malicious contracts, which can receive the user assets and give nothing back in return.

Recommendations

Remove the method from both contracts as it is not needed as the contracts shouldn't be holding any value or allowances anyway between transactions - if you wish to update the addresses in them you can just deploy new `Swap` or `Aave` contracts and make the front-end forward calls to them.

8.3. Low Findings

[L-01] The `Uniswap` functionality does not have a deadline parameter

The `Uniswap` proxy contract should use a transaction deadline mechanism, due to the following attack vector:

1. Alice wants to execute a swap, sets slippage to 10% and sends a transaction to the mempool, but with a very low gas fee
2. Miners/validators see the transaction but the fe is not attractive, so the transaction is stale and pending for a long time
3. After a week (let's say) the average gas fees drop low enough for the miners/validators to execute the transaction but the price of the assets has changed drastically
4. Now the value Alice receives is much lower and possibly close to the max slippage she set.

It's best to add a `deadline` property in the `IUniswap.MintParams` struct that you will forward to the calls to the `NonFungiblePositionManager` Uniswap contract to prevent this issue.

[L-02] Fake Curve pools and Compound tokens can be used

The methods in the `Curve` contract take in a `ICurvePool _pool` parameter which means the user can send an address that is a contract that he deployed, not one from the actual `Curve` protocol. The same situation is present with the `ICToken _cToken` parameter in the `Compound` contract. A malicious user can use this vulnerability to spam events that are not for actual usages of either protocol. Fixing this can be done with a predefined whitelist of pools and tokens.