SmolRefuel Security Review

Pashov Audit Group

Conducted by: Peakbolt, pontifex, ast3ros, juancito

May 24th 2024 - May 25th 2024

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our 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 we try to find as many vulnerabilities as possible. We 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 **smolrefuel/contracts** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About SmolRefuel

Smolrefuel allows token-to-ETH exchanges without holding native token to pay for gas.

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 - 706cb99aa54d139d3ab9e0f2459b67fa834e006f

fixes review commit hash - 11e4c54aafcce99af4af778e8458e5d8cd80bae4

Scope

The following smart contracts were in scope of the audit:

- `SmolRefuel`

7. Executive Summary

Over the course of the security review, Peakbolt, pontifex, ast3ros, juancito engaged with SmolRefuel to review SmolRefuel. In this period of time a total of **6** issues were uncovered.

Protocol Summary

Protocol Name	SmolRefuel
Repository	https://github.com/smolrefuel/contracts
Date	May 24th 2024 - May 25th 2024
Protocol Type	DEX

Findings Count

Severity	**Amount**
Medium	3
Low	3
Total Findings	**6**

Summary of Findings

ID	Title	Severity	Status
[M-01]	Tokens without true return are not supported	Medium	Resolved
[M-02]	The bot balance can be drained by reverting tx	Medium	Acknowledged
[M-03]	refuel() DoS by frontrunning with permit()	Medium	Resolved
[L-01]	Use Ownable2Step rather than Ownable	Low	Resolved
[L-02]	Consider bounding botTake, router, and contractToApprove	Low	Acknowledged
[L-03]	Some tokens do not work with the permit function	Low	Acknowledged

8. Findings

8.1. Medium Findings

[M-01] Tokens without `true` return are not supported

Severity

Impact: Medium

Likelihood: Medium

Description

Certain tokens, such as USDT, deviate from the IERC20 interface by not returning a boolean value upon transfer.

Therefore these tokens cannot be retrieved by the `retrieveToken` function in the `SmolRefuel` contract because the function will always revert.

```
function retrieveToken
      (IERC20 token, uint256 amount, address to) external onlyOwner {
        token.transfer(to, amount);
    }
```

In addition, If any of these tokens has `permit` function, the `refuel` wouldn't work for them since the transfer will also always revert.

```
function refuel(
        ERC20Permit token,
        address payable from,
        uint256 amount,
        uint256 deadline,
        uint8 v,
        bytes32 r,
        bytes32 s,
        address router,
        bytes calldata data,
        address contractToApprove,
        uint256 botTake
    ) external payable {
        ...
        token.permit(from, address(this), amount, deadline, v, r, s);
        token.transferFrom(from, address(this), amount); // @audit revert here
        ...
    }
```

Recommendations

Using `safeTransfer` instead of `transfer` to retrieve the tokens.

```
function retrieveToken
      (IERC20 token, uint256 amount, address to) external onlyOwner {
-       token.transfer(to, amount);
+       token.safeTransfer(to, amount);
    }
```

Consider using `safeTransferFrom` in `refuel`:

```
function refuel(...) {
        ...
-     token.transferFrom(from, address(this), amount);
+     token.safeTransferFrom(from, address(this), amount);
       ...
    }
```

[M-02] The `bot` balance can be drained by reverting tx

Severity

Impact: Medium

Likelihood: Medium

Description

The protocol sends ETH to untrusted addresses with the `sendETH` function. The function throws `EthTransferFailed` in case of an unsuccessful transaction. This fact lets an attacker drain the `bot` balance.

```
function sendETH(address payable to, uint256 amount) internal {
        (bool sent,) = to.call{value: amount}("");
>>      if (!sent) revert EthTransferFailed();
    }

    function refuelWithoutPermit(
        IERC20 token,
        address payable from,
        uint256 amount,
        address router,
        bytes calldata data,
        address contractToApprove,
        uint256 botTake
    ) external payable {
        if (msg.sender != bot) revert AuthFailed();

        // fetch token from user
        token.safeTransferFrom(from, address(this), amount);

        // @note give infinite approval to the contract
        // added to save gas
        // @dev if contractToApprove is 0x0, it means the contract have enough
        // allowance, computed offchain
        if (contractToApprove != address(0)) token.safeApprove
          (contractToApprove, type(uint256).max);

        (bool sent,) = router.call(data);

        if (!sent) revert FailedRouterCall();

        sendETH(bot, botTake);
>>      sendETH(from, address(this).balance);
    }
```

Though the protocol emulates a tx with gas estimation it can't prevent front-running the tx with the `from` contract state changing which causes the tx reverting. The possible attack can consist of the next steps. 1. The attacker's contract (`from`) approves the `SmolRefuel` contract and then sends a request. Depending on if the reasonable gas limit exists in the backend logic the attacker can include in the transaction a gas-costly callback to maximize the protocol losses. 2. The attacker frontruns the refuel tx to change the initial state of the `from` contract to cause the refuel tx to reverting. This can include just reverting in a fallback or increasing the necessary gas which will throw an out-of-gas error.

The attack cost can be relatively low compared with the protocol losses.

Recommendations

There is no simple mitigation for the issue. Consider separating the interaction with untrusted addresses from the swap logic.

[M-03] `refuel()` DoS by frontrunning with `permit()`

Severity

Impact: Medium

Likelihood: Medium

Description

`refuel()` allows `bot` to use `ERC20Permit` for supported tokens to transfer in the tokens for swapping to ETH.

```
function refuel(
     ...
    ) external payable {
        if (msg.sender != bot) revert AuthFailed();

        token.permit(from, address(this), amount, deadline, v, r, s);
        token.transferFrom(from, address(this), amount);
```

However, `permit()` can be frontrun causing the bot refuel transactions to be griefed. As `bot` pays for the gas upfront, this can cause `bot` to be drained of its ETH, since it will not be able to recoup it from the reverted transaction.

1. Bot calls `refuel()` with a valid permit signature to transfer in WBTC tokens.
2. The attacker sees the tx in mempool, and proceeds to frontrun `refuel()` with a `permit()` using the valid permit signature.
3. Bot's tx will fail due to incorrect nonce as the Attacker tx will increment the nonce as the signature will be consumed.

Recommendations

Use a try/catch as recommended in OZ docs so that it will proceed and not revert when the permit signature has been consumed.

```
try token.permit(from, address
          (this), amount, deadline, v, r, s) {} catch {}
        token.transferFrom(from, address(this), amount);
```

8.2. Low Findings

[L-01] Use Ownable2Step rather than Ownable

`Ownable2Step` and `Ownable2StepUpgradeable` prevent the contract ownership from mistakenly being transferred to an address that cannot handle it (e.g. due to a typo in the address), by requiring that the recipient of the owner permissions actively accept via a contract call of its own.

```
contract SmolRefuel is Ownable {
```

[L-02] Consider bounding `botTake`, `router`, and `contractToApprove`

Both `refuel()` and `refuelWithoutPermit()` allow `botTake`, `router`, and `contractToApprove` to be passed in as parameters. As they are not bounded to a range or value, it is possible for these values to be arbitrary, which can be exploited if `bot` is compromised.

It is recommended to set `botTake` to a predefined percentage and `router`/`contractToApprove` to a pre-defined swap contract. Both of them should be stored and configured by the contract owner, to provide additional safety in the event of a `bot` compromise.

[L-03] Some tokens do not work with the permit function

DAI and other tokens use a different version of the `permit()` function than the one used in the protocol (EIP-2612).

```
function permit(
  addressowner,
  addressspender,
  uintvalue,
  uintdeadline,
  uint8v,
  bytes32r,
  bytes32s
) external;

function permit(
  addressowner,
  addressspender,
  uint256nonce,
  uint256deadline,
  boolallowed,
  uint8v,
  bytes32r,
  bytes32s
) external;
```

In those cases, users won't be able to use the `refuel()` function with permits.

It would be recommended to implement another function that allows users to sign permits with the DAI-like permit interface.