# Polymath Core Audit

November 2019

By CoinFabrik

**Introduction** **3**

**Summary** **3**

**Severity Classification** **5**

**Detailed findings** **6**

Critical severity 6

Medium severity 6

Minor severity 6

Potential gas issue with transfers in Istanbul 6

Token granularity is not respected 8

Enhancements 9

Code does nothing in function finalize of USDTieredSTO 9

Unnecessary imports 10

Observations 10

**Conclusion** **12**

**Appendix - Modules** **13**

Organization 13

USDTieredSTO 14

CappedSTO 15

RestrictedPartialSaleTM 16

# Introduction

CoinFabrik was asked to audit the contracts for the Polymath project. Firstly, we will provide a summary of our discoveries and secondly, we will show the details of our findings.

# Summary

The contracts audited are from the Polymath core repository at https://github.com/PolymathNetwork/polymath-core/tree/dev-3.1.0. The audit is based on the commit bdb055769dd87a09327c6200aed5d994ab0237e8, and updated to reflect changes at d083b134f562108f83c852ba37e25c6c4ee6dc66.

The audit specified that we looked for the changes to three modules from dev-3.0 to dev-3.1 and the pull request for “Usage Cost” feature:

- **USDTieredSTO**

  In directory contracts/modules/STO/USDTiered:

  - USDTieredSTO.sol
  - USDTieredSTOFactory.sol
  - USDTieredSTOStorage.sol
  - USDTieredSTOProxy.sol

  There are two major changes in this version:

  - Allow pre-minting of tokens.
    Tokens can be minted before the start of the token sale. This feature can be changed only before the sale starts.
  - Allow fiat currencies other than USD.
    Setting a custom oracle allows the contract to support other fiat currencies.

- **CappedSTO**

  In directory contracts/modules/STO/Capped:

- CappedSTO.sol
- CappedSTOFactory.sol
- CappedSTOProxy.sol
- CappedSTOStorage.sol

There is one major change in this version:

- Allow pre-minting of tokens.

- **RestrictedPartialTransferManager**

  In directory contracts/modules/TransferManager/RPTM:

  - RestrictedPartialSaleTM.sol
  - RestrictedPartialSaleTMFactory.sol
  - RestrictedPartialSaleTMProxy.sol
  - RestrictedPartialSaleTMStorage.sol

  this is a new module migrated from experimental that blocks partial transfer of balances.

- **"Usage Cost" feature pull request #837**
  https://github.com/PolymathNetwork/polymath-core/pull/837

  This feature enables a module to charge a fee based on usage. Modules already supported a setup fee when they are created.

The following analyses were performed:

- Misuse of the different call methods: call.value(), send() and transfer().
- Integer rounding errors, overflow, underflow and related usage of SafeMath functions.

- Old compiler version pragmas.
- Race conditions such as reentrancy attacks or front running.
- Misuse of block timestamps, assuming anything other than them being strictly increasing.
- Contract softlocking attacks (DoS).
- Potential gas cost of functions being over the gas limit.
- Missing function qualifiers and their misuse.
- Fallback functions with a higher gas cost than the one that a transfer or send call allows.
- Fraudulent or erroneous code.
- Code and contract interaction complexity.
- Wrong or missing error handling.
- Overuse of transfers in a single transaction instead of using withdrawal patterns.
- Insufficient analysis of the function input requirements.

# Severity Classification

The security risk findings are evaluated according to the following classification:

- **Critical:** These are issues that we managed to exploit. They compromise the system seriously. We suggest to fix them **immediately**.
- **Medium:** These are potentially exploitable issues. We did not manage to exploit them and maybe they can not be exploited right now or the impact is not clear, but they represent a security risk that can arise problems in the near future. We suggest to fix them **as soon as possible**.
- **Minor:** These issues represent problems that are relatively small or difficult to exploit but can be used in combination with other issues. These kinds of issues do not block deployments. They should be taken into account and be fixed eventually.

- **Enhancement:** These kinds of findings do not represent a security risk. They are best practices that we suggest to implement.

This classification is summarized in the following table:

| Severity | Exploitable | Blocks the deployment | We suggest to fix it... |
|---|---|---|---|
| Critical | Yes | Yes | Immediately |
| Medium | In the near future | Yes | As soon as possible |
| Minor | Unlikely | No | Eventually |
| Enhancement | No | No | Optionally |

# Detailed findings

## Critical severity

No issues have been found.

## Medium severity

No issues have been found.

## Minor severity

### Potential gas issue with transfers in Istanbul

Previously it was considered safe to use Solidity's *transfer* and *send* to transfer Ether from a contract. These primitives provide protection against reentrancy attacks by limiting the execution of the recipient's fallback function to a maximum gas stipend of 2,300 gas.

The activation of the Istanbul fork of Ethereum's mainnet on early December 2019 includes EIP 1884 which will increase the cost of several common opcodes. This EIP will possibly affect all contracts that use a fixed amount of gas when calling another contract because the changes in prices will modify the total gas used.

An important class of affected contracts are the ones that use Solidity's transfer or send because of the fixed gas stipend. If after the fork the recipient's fallback function requires more gas than the gas stipend it will cause an "*Out of gas*" error when called.

In the function *buyWithETHRateLimited* of *USDTieredSTO* there are two transfers of Ether: one to the token sale wallet and another to return the excess to the user.

```
        // Forward ETH to issuer wallet
        wallet.transfer(spentValue);

        // Refund excess ETH to investor wallet
        msg.sender.transfer(msg.value.sub(spentValue));
```

The function will stop working if either *wallet* or *msg.sender* is affected. For a wallet to be affected it has to meet two conditions:

1. **There has to be a contract.** This is not unusual because multisig wallets are smart contracts and they are being used.
2. **It has to be affected by EIP 1884.** A recent study analyzing the impact has found some notable examples like OpenZepplin SDK, but no popular wallet has been found to be affected.

The owner of the security token can always change the wallet used by calling *modifyAddresses*. Having an affected wallet can be fixed by the owner.

When a user tries to buy tokens with an affected contract wallet the transaction will fail. We consider this is an unlikely case since no popular wallet has been found to be affected yet.

Considering the future proof of the contracts it is not in the best interest to continue using Solidity's transfer. Since pricing of opcodes is not fixed; it will be adjusted periodically and the gas stipend will remain fixed at 2,300 gas.

We recommend using OpenZeppelin's sendValue:

```
function sendValue(address payable recipient, uint256 amount) internal {
    require(address(this).balance >= amount, "Address: insufficient balance");

    // solhint-disable-next-line avoid-call-value
    (bool success, ) = recipient.call.value(amount)("");
    require(success, "Address: unable to send value, recipient may have
reverted");
}
```

**We understand the potential risk is low and it is easier to suggest to the users to switch to an unaffected wallet. To implement a solution like OpenZeppelin's sendValue without care might have some consequences like reentrancy attacks. During our audit the contracts examined follow the checks-effects-interactions pattern which should eliminate possible reentrancy attacks.**

### Token granularity is not respected

A security token's granularity is the minimum unit in which a token can be divided. Any operation that does not respect the granularity will generate an error making the transaction fail.

The function *_modifyTiers* of USDTieredSTO receives an array containing the amount of available tokens per tier and another array with the amount of tokens available with discount. The code never verifies if these amounts respect the granularity.

```
    function _modifyTiers(
        uint256[] memory _ratePerTier,
        uint256[] memory _ratePerTierDiscountPoly,
        uint256[] memory _tokensPerTierTotal,
        uint256[] memory _tokensPerTierDiscountPoly
    )
```

If the amount of tokens available in a tier does not respect the granularity, it would prevent the tier from being fully sold since amounts lower than the granularity cannot be minted or transferred.

Additionally if the sum of all available tokens returned by *_getTotalTokensCap* does not respect the granularity, it will fail while trying to enable the *pre-minting* feature.

```
    function allowPreMinting() external withPerm(ADMIN) {
        _allowPreMinting(_getTotalTokensCap());
    }
```

The owner can always fix the error by submitting new tiers with the correct granularity through the function *modifyTiers*.

We suggest to add checks to *_modifyTiers* to enforce token's granularity or to round the tier total to a valid amount.

Similarly in CappedSTO the cap is never checked against the token's granularity.

Since the cap cannot be modified after the initial configuration, it is not possible to fix it and it will require a new deployment of the module because the sale will not be finalized.

We recommend to enforce the token's granularity before accepting the cap. Besides, Polymath might provide a method that allows the owner to modify the cap before the token sale starts, if it was configured incorrectly.

> **The Polymath team has added the checks for granularity at pull request https://github.com/PolymathNetwork/polymath-core/pull/862.**

# Enhancements

## Code does nothing in function finalize of USDTieredSTO

The function *finalize* has to mint/transfer unsold security tokens to the Treasury Wallet of the token sale.

At line 359 the variable *tempReturn* represents the total amount of unsold tokens in every tier.

```
        if (preMintAllowed) {
            if (tempReturned == securityToken.balanceOf(address(this)))
                tempReturned = securityToken.balanceOf(address(this));
        }
```

When the condition is satisfied *tempReturned* will be assigned the same value it already contains.

These values should always be equal. It is an invariant of the contract that minted tokens are accumulated. If they don't match, it means that the token balance was manipulated.

If the code is there to check for invalid conditions we recommend to use a *require* statement instead, so that detection during testing is easier.

```
require(tempReturned = securityToken.balanceOf(address(this)), "Invalid balance");
```

> **The Polymath team has removed the if at pull request https://github.com/PolymathNetwork/polymath-core/pull/862 and the balance is assigned inconditionally to *tempReturned*.**

### Unnecessary imports

The library SafeMath.sol is imported in the file RestrictedPartialSaleTM.sol.

```
import "openzeppelin-solidity/contracts/math/SafeMath.sol";
```

Similarly *IPolymathRegistry* is included in Module.sol.

```
import "../interfaces/IPolymathRegistry.sol";
```

Nevertheless, neither of them is used in any of the functions defined in the file. This will not use extra resources in running time but it will increase compilation time.

We recommend to remove them to keep a minimal set of dependencies.

> **The Polymath team has removed the unneccessary imports at https://github.com/PolymathNetwork/polymath-core/pull/862.**

## Observations

1. In USDTieredSTO the owner can configure a custom oracle. Since the oracle is not vetted by anyone, a malicious owner can use an oracle that allows him to manipulate the price.
   Polymath team told us that the functionality is there to allow the owner to change the oracle if it is providing erroneous prices or malfunctioning.
   We understand that investors must have a high degree of trust on the owner to participate in a token sale. It is the owner's responsibility to adequately configure the token sale including the oracle's address and to quickly react in case of oracle misbehavior.

**We agree that no change is necessary and the oracles functions as it is expected.**

2. In function *_modifyTiers* when pre-minting is enabled and tiers are modified it might cause new tokens to be minted or deleted but no events are generated. It might be useful to generate an event when it changes.
   **The Polymath team has communicated to us that due to the contract size being too close to the 24 Kbytes limit it will not implement this suggestion. Whenever tiers are updated with the function modifyTiers the event *SetTiers* is emitted with the data from tiers. If pre-minting is active and tokens are issued or redeemed the corresponding event *Issued* or *Redeemed* is emitted by the security token contract.**

3. Function *getCustomOracleAddress* in USDTieredSTO implicitly returns the zero address when called with *FundRaiseType.SC*. We recommend to document such behavior.
   **The behavior has been documented at pull request https://github.com/PolymathNetwork/polymath-core/pull/862.**

4. The magic constant *uint256(10) ** 18* is used in function *_getTokenAmount* from CappedSTO. It is better to define it as a constant. The functions *div* and *mul* from the library DecimalMath have a similar functionality. Polymath might consider using them instead.
   **Now the contract uses the functions provided by DecimalMath https://github.com/PolymathNetwork/polymath-core/pull/862.**

5. The constructor in *DummySTOFactory* calls *UpgradableModuleFactory* with version 3.0.0.
   **It was updated to version 3.1.0 at pull request https://github.com/PolymathNetwork/polymath-core/pull/862.**

6. Some functions have been declared as **public** but they were declared previously in a base contract as **external**. For example *changeUsageCost* in ModuleFactory.sol and *changeAllowBeneficialInvestments* in CappedSTO.sol. We recommend to always respect the declaration to minimize possible problems in a future refactorization.
   **It was fixed so the functions have the proper access qualifier pull request https://github.com/PolymathNetwork/polymath-core/pull/862.**

7. Some variables don't have an access qualifier like *initialVersion*, *typesData* and *tagsData* in ModuleFactory.sol, *oracleKeys* in USDTieredSTOStorage.sol. We recommend to always use the correct access qualifier.

**It was fixed so variables have the correct qualifier at pull request https://github.com/PolymathNetwork/polymath-core/pull/862.**

# Conclusion

The audited contracts are simple but they have complex dependencies, making interactions with other contracts more complicated. The contracts are well documented.

We found two minor issues:

- The next Ethereum fork “Istanbul” changes opcodes pricing making some transfers to fail with an out of gas error in an affected wallet.
- In USDTieredSTO it is possible to configure tokens amounts that do not respect the token’s granularity.

**Disclaimer: This audit report is not a security warranty, investment advice, or an approval of the Polymath Core project since CoinFabrik has not reviewed its platform. Moreover, it does not provide a smart contract code faultlessness guarantee.**

# Appendix - Modules

## Organization

The modules are split into four contracts:

1. **Logic component.** It contains the logic of the module which need to be deployed just once per version.
2. **Storage component.** It contains the storage of the module. Every instance of the module will contain one storage component.
3. **Factory component.** It is responsible for deploying an instance of the component. It is also responsible for upgrades to the contract logic.
4. **Proxy component.** It is the component that will link the logic and storage component.

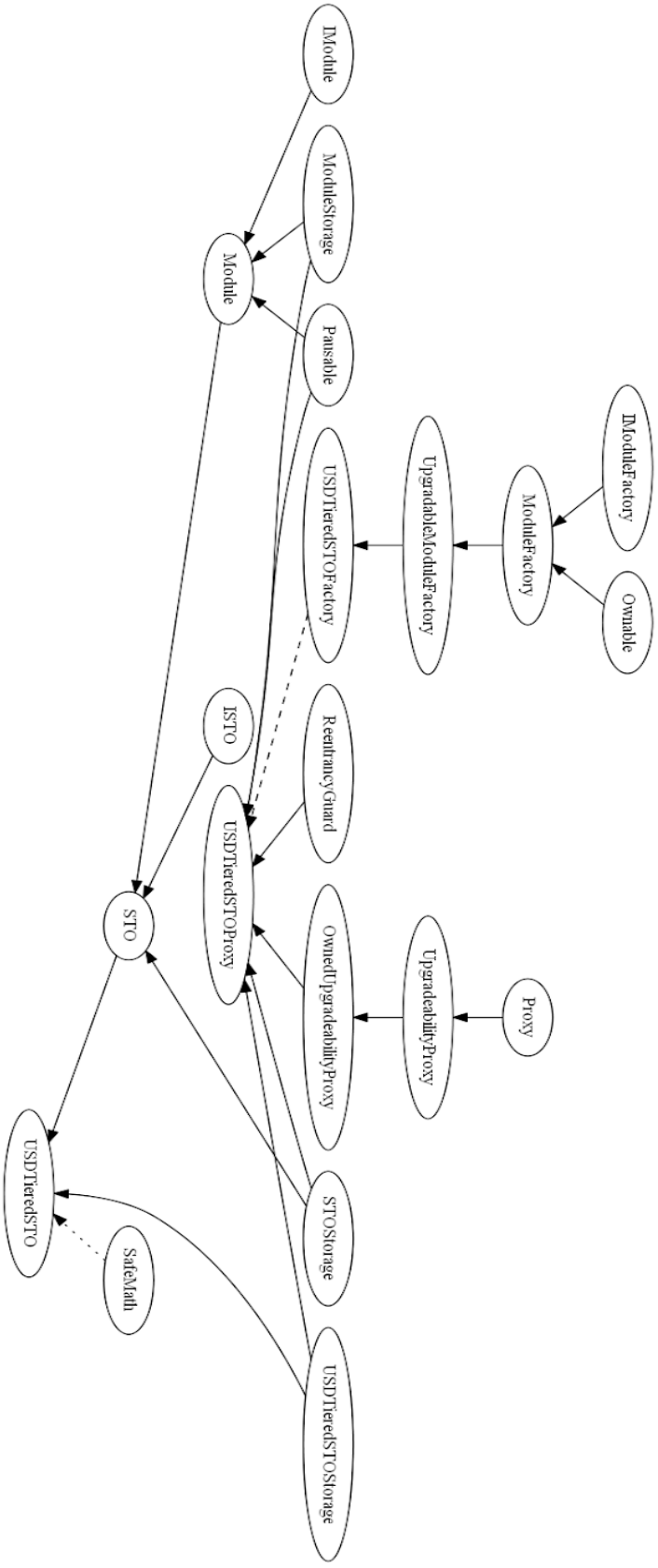
This division allows to deploy a lighter component per instance that will contain only the storage. The heavier logic is deployed only once and, using the proxy, it will be shared by all instances.

Following are simplified dependency graphs of the audited modules.

- Full lines indicate inheritance
- Dotted lines are library dependencies
- Dashed lines are factory relationship

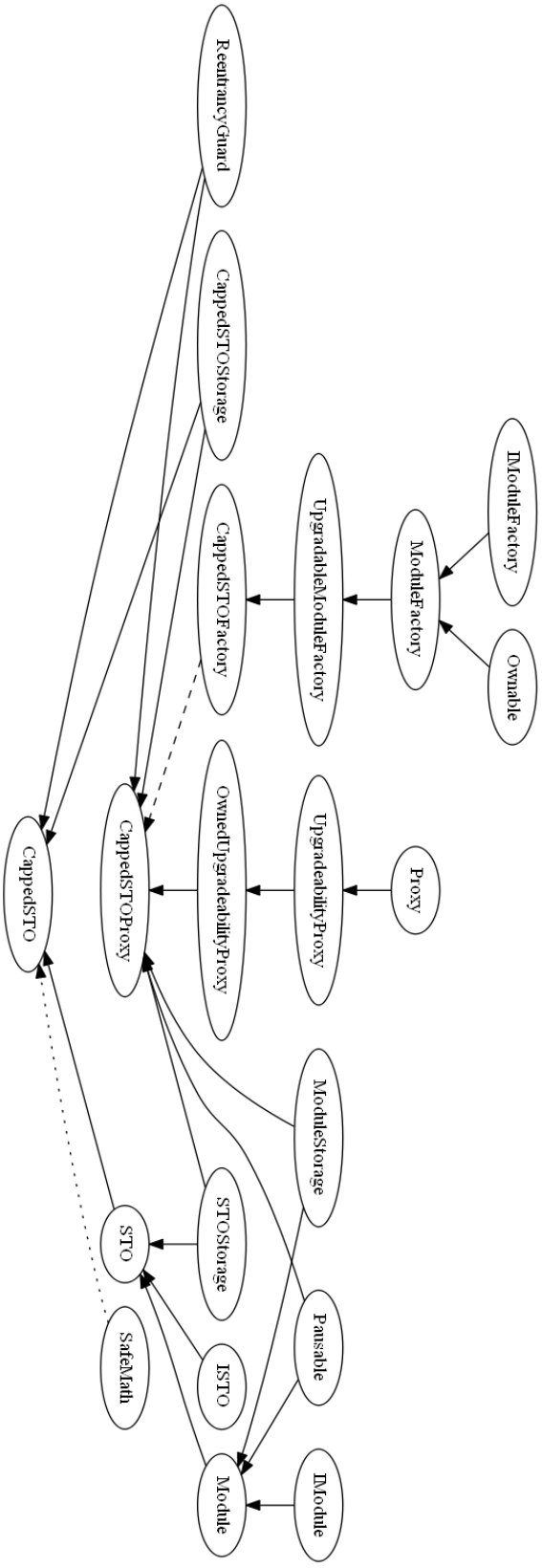
## USDTieredSTO

It provides a token offering to have several tiers each one with its own token price and available tokens, with the tier limit in a fiat currency.



## CappedSTO

This module provides a cap for the token offering. Purchases above the limit will be ignored.



### RestrictedPartialSaleTM

The module enables the owner to limit the partial sale of tokens, only the full balance can be transferred. The owner can allow certain addresses to make partial transfers.