SOLIDIFIED

Audit Report for USM - December 10, 2020

Summary

Audit Report prepared by Solidified covering the USM Minimalist USD Stable Coin smart contracts.

Process and Delivery

Three (3) independent Solidified experts performed an unbiased and isolated audit of the code below. The debrief took place on December 10, 2020, and the results are presented here.

Audited Files

The following contracts were covered during the audit:

```
contracts
├── Delegable.sol
├── FUM.sol
├── IUSM.sol
├── Migrations.sol
├── MinOut.sol
├── Proxy.sol
├── USM.sol
├── USMTemplate.sol
├── WadMath.sol
└── oracles
    ├── ChainlinkOracle.sol
    ├── CompoundOpenOracle.sol
    ├── MedianOracle.sol
    ├── Oracle.sol
    └── OurUniswapV2TWAPOracle.sol
```

Supplied in the following source code repositories:

https://github.com/usmfum/USM

The final commit commit number covered by this report including fixes to issues encountered provided is `5b450aea474bc07f38606851b9ef6f84d2be0b3f`

Intended Behavior

The smart contract implements an algorithmic stable coin system, which is over-collateralized by ETH provided by third parties (funders). Funders are issued a FUM token in exchange for the ETH provided.

SOLIDIFIED

Audit Report for USM - December 10, 2020

Executive Summary

Smart contract audits are an important step to improve the security of smart contracts and can find many issues. However, auditing complex codebases has its limits and a remaining risk is present (see disclaimer).

Users of a smart contract system should exercise caution. In order to help with the evaluation of the remaining risk, we provide a measure of the following key indicators: **code complexity**, **code readability**, **level of documentation**, and **test coverage**.

Note, that high complexity or lower test coverage does not necessarily equate to a higher risk, although certain bugs are more easily detected in unit testing than a security audit and vice versa.

Criteria	Status	Comment
Code complexity	Medium	-
Code readability and clarity	High	-
Level of Documentation	High	-
Test Coverage	High	-

SOLIDIFIED

Audit Report for USM - December 10, 2020

Issues Found

Solidified found that the USM contracts contain 2 critical issues, no major issue, and 1 minor issue, in addition to 5 informational notes.

We recommend all issues are amended, while the notes are up to the team's discretion, as it refers to best practices.

Issue #	Description	Severity	Status
1	Proxy.sol: Anyone can burn/defund someone else's USM/FUM	Critical	Resolved
2	Proxy.sol: Anyone can mint USM and FUM using someone else's approved WETH	Critical	Resolved
3	EIP712 signature domain leftover from another project	Minor	Resolved
4	Oracles might disappear or become unreliable	Note	-
5	Malleable signatures accepted	Note	-
6	Tautologies in WadMath.sol	Note	-
7	Variable shadowed in inherited contracts	Note	-
8	ETH can be forced into the USM contract bypassing mint()	Note	-

SOLIDIFIED

Audit Report for USM - December 10, 2020

Critical Issues

1. Proxy.sol: Anyone can burn/defund someone else's USM/FUM

The `Proxy.sol` functions `burn()` and `defund()` don't check `msg.sender`. Thus, anyone can burn someone else's USM/FUM and get WETH to a desired destination as long as Proxy has Delegate permissions.

Recommendation
Implement access control mechanisms on the desired `msg.sender`.

Update
Fixed

2. Proxy.sol: Anyone can mint USM and FUM using someone else's approved WETH

When a victim approves certain amount of WETH to `Proxy.sol` contract, anyone can call:

```
mint(address from, address to, uint ethIn, uint minUsmOut)
fund(address from, address to, uint ethIn, uint minFumOut)
```

and use those WETH to mint USM/FUM for themselves.
While it is common to approve an infinite amount of funds for the DeFi contracts, it is not the only way this functionality could be abused.
In most cases, the approval would be executed in a separate preceding transaction. A malicious actor could front-run the mint or fund transactions and steal the approved WETH.

Recommendation
Implement access control mechanisms on the desired `msg.sender`.

Update
Fixed

Major Issues

No major issues have been found.

Minor Issues

3. EIP712 signature domain leftover from another project

The EIP712 signature domain in `Delegable.sol` seems to have been re-used from the Yield project.

Recommendation
Provide the project with a custom signature domain.

Update
Fixed

Notes

4. Oracles might disappear or become unreliable

Price Oracles introduce risks in projects since they have to strike a fine balance between being up to date, resistant to price manipulation attacks, and decentralization. The project tries to reduce the risk by using the median of three oracles. However, hardcoded oracles may become unreliable or disappear. For instance, the Uniswap v2 oracle may become inaccurate once Uniswap deploys v3 as a side-effect of low liquidity. There is currently no facility to update or replace the oracles used.

Recommendation
There is no best solution for this problem. The note is merely intended to highlight the trad-offs involved. Since stable coins are very long-lived, potential mitigation might be a facility to replace or update oracles. However, this would introduce centralization or complexity due to a governance system.

5. Malleable signatures accepted

The `addDelegateBySignature()` function in `Delegable.sol` uses the built-in `ecrecover()`. This function still allows malleable signatures for backward compatibility reasons. Signatures that have an `s` value larger than `0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF5D576E7357A4501DDFE92F46681B20A0` are usually rejected for Ethereum address post EIP-2.

Recommendation
Consider rejecting signatures with `s` values in the upper ranges, even though it may not be a security issue in this case.
For an example solution see https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/cryptography/ECDSA.sol

6. Tautologies in WadMath.sol

In `WadMath.sol`, function `wadHalfExp()` performs two checks over unsigned integer variables that will `always` resolve to `true`, since unsigned integers are always `>=0`:

```
require(power >= 0, "power must be positive");
```

```
require(powerInTenths >= 0, "powerInTenths must be positive");
```

Recommendation
Remove the unnecessary require statements.

7. Variable shadowed in inherited contracts

`USM.sol` declares shadows the constant `NUM_UNISWAP_PAIRS`, which is already declared in the imported and inherited `MedianOracle.sol`. Whilst the values are the same in this case, the code becomes less maintainable and changes in one of the contracts might lead to inconsistencies.

Recommendation
Remove the second declaration.

8. ETH can be forced into the USM contract bypassing mint()

There are some scenarios where a contract can receive ETH without triggering the execution of a receive function, for example, when another contract self-destructs and passes the address as beneficiary.

Since the function `ethPool()` relies on the balance of the USM contract, if it's forcibly sent ETH it might return incorrect values for view functions, eg: `debtRatio()`. There's no security implication, as the amount will be included in the next call to mint. This issue is present to raise awareness of the possibility.

Recommendation
It's a better practice to manually count the incoming ETH and rely on that number instead of `address(this).balance`.

Audit Report for USM - December 10, 2020

Disclaimer

Solidified audit is not a security warranty, investment advice, or an endorsement of USM or its products. This audit does not provide a security or correctness guarantee of the audited smart contract. Securing smart contracts is a multistep process, therefore running a bug bounty program as a complement to this audit is strongly recommended.

The individual audit reports are anonymized and combined during a debrief process, in order to provide an unbiased delivery and protect the auditors of Solidified platform from legal and financial liability.

Solidified Technologies Inc.