ELITE CYBERSECURITY FOR BLOCKCHAIN ENTERPRISES

ANATHA ERC20 TOKEN SMART CONTRACT SECURITY AUDIT

Report Prepared by: Halborn
Date of Engagement: September 27-30, 2020

Table of Contents

DOCUMENT REVISION HISTORY 3

CONTACTS 3

1 - EXECUTIVE SUMMARY 4

1.1 INTRODUCTION 4
1.2 TEST APPROACH AND METHODOLOGY 5
1.3 SCOPE 6

2 - ASSESSMENT SUMMARY AND FINDINGS OVERVIEW 7

3 - FINDINGS AND TECHNICAL DETAILS 7

3.1 FLOATING PRAGMA – LOW 7

DESCRIPTION 7
CODE LOCATION 8
RECOMMENDATION 8

3.2 STATIC ANALYSIS REPORT 8

DESCRIPTION 8
3.2.1. FLOATING PRAGMA - LOW 8
RESULTS 9
RECOMMENDATION 9
3.2.2. POSSIBLE MISUSE OF PUBLIC VARIABLES- INFORMATIONAL 9
RECOMMENDATION 10
3.2.3. ERC CONFORMAL CHECKER- INFORMATIONAL 11
RESULTS 11
3.2.4. SECURITY TESTING EXPLOITATION - INFORMATIONAL 12
RESULTS 12

3.3 AUTOMATED SECURITY SCAN - INFORMATIONAL 12

DESCRIPTION 12
RESULTS 12

Document Revision History

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	9/28/2020	Gabi Urrutia
0.2	Document Edits	9/30/2020	Gabi Urrutia
1.0	Document DRAFT		

Contacts

CONTACT	COMPANY	EMAIL	PHONE
STEVEN WALBROEHL	Halborn	Steven.Walbroehl@halborn.com	--
ROB BEHNKE	Halborn	Rob.behnke@halborn.com	--
GABI URRUTIA	Halborn	Gabi.Urrutia@halborn.com	--
NISHIT MAJITHIA	Halborn	Nishit.Majithia@halborn.com	--

1 - Executive Summary

1.1 Introduction

Anatha engaged Halborn to conduct a security assessment on their ERC20 Token smart contract beginning on September 27th, 2020 and ending September 30th, 2020. The security assessment was scoped to the contracts ERC20MinterPauser.sol initialized by UpgradableCoin.sol and an audit of the security risk and implications regarding the changes introduced by the development team at Anatha prior to its production release shortly following the assessments deadline.

UpgradableCoin initialize the ERC20MinterPauser without a constructor according to the role-based access control (RBAC). The main goal of ERC20MinterPauser is to exchange Ether (ETH) for Wrapped Anatha (wANATHA) token.

Overall, the smart contract code is extremely well documented, follows a high-quality software development standard, contains many utilities and automation scripts to support continuous deployment / testing / integration, and does NOT contain any obvious exploitation vectors that Halborn was able to leverage within the timeframe of testing allotted.

Though the outcome of this security audit is satisfactory; due to time and resource constraints, only testing and verification of essential properties related to the ERC20MinterPauser was performed to achieve objectives and deliverables set in the scope. It is important to remark the use of the best practices for secure smart contract development. Halborn recommends performing further testing to validate extended safety and correctness in context to the whole set of contracts. External threats, such as economic attacks, oracle attacks, and inter-contract functions and calls should be validated for expected logic and state.

1.2 Test Approach and Methodology

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of the smart contract audit. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of smart contracts and can quickly identify items that do not follow security best practices. The following phases and associated tools were used throughout the term of the audit:

- **Research into architecture, purpose, and use of ERC20 Token.**
- **Smart Contract manual code read and walkthrough.**
- **Graphing out functionality and contract logic/connectivity/functions (*solgraph*)**
- **Manual Assessment of use and safety for the critical solidity variables and functions in scope to identify any arithmetic related vulnerability classes.**
- **Scanning of solidity files for vulnerabilities, security hotspots, or bugs. (*MythX*)**
- **Static Analysis of security for scoped contract, and imported functions. (*Slither*)**
- **Testnet deployment (*Truffle*, *Ganache, Infura*)**
- **Smart Contract analysis and automatic exploitation (*teEther*)**
- **Symbolic Execution / EVM bytecode security assessment (limited-time)**

1.3 Scope

IN-SCOPE:

Code related to the ERC20MinterPauser smart contract.
Specific commit of contract: b6b9ecc443651981f42eaafe6ab1ce9368cd1755

OUT-OF-SCOPE:

External contracts, External Oracles, other smart contracts in the repository or imported by ERC20MinterPauser, economic attacks.

2 - Assessment Summary and Findings Overview

CRITICAL	HIGH	MEDIUM	LOW
0	0	0	1

SECURITY ANALYSIS	RISK LEVEL
FLOATING PRAMA	LOW
POSSIBLE MISUSE OF PUBLIC VARIABLES	INFORMATIONAL
ERC CONFORMAL CHECKER	INFORMATIONAL
SECURITY TESTING EXPLOITATION	INFORMATIONAL
AUTOMATED SECURITY SCAN RESULTS	INFORMATIONAL

3 - Findings and Technical Details

3.1 FLOATING PRAGMA – LOW

Description

Anatha token contracts use the floating pragma ^0.6.2. Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively. At the time of this audit, the current version is already at 0.7 The newer versions provide features that provide checks and accounting, as well as prevent insecure use of code.

Reference: https://consensys.github.io/smart-contract-best-practices/recommendations/#lock-pragmas-to-specific-compiler-version

Code Location

ERC20MinterPauser.sol Line #1
UpgradableCoin.sol Line #1

```
pragma solidity ^0.6.2;
```

Recommendation

Consider lock the pragma version known bugs for the compiler version. When possible, do not use floating pragma in the final live deployment.

3.2 STATIC ANALYSIS REPORT

Description

Halborn used automated testing techniques to enhance coverage of certain areas of the scoped contract. Among the tools used was Slither, a Solidity static analysis framework and teEther, an analysis and automatic exploitation framework. After Halborn verified all the contracts in the repository and was able to compile them correctly into their abi and binary formats, Slither and teEther were run on the ERC20MinterPauser and UpgradableCoin contracts.

3.2.1. FLOATING PRAGMA - LOW

This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire codebase.

Results

First finding is the use of floating pragma which was previously discovered in the manual code review:

```
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#assembly-usage
INFO:Detectors:
Different versions of Solidity is used in :
        - Version used: ['>=0.4.24<0.7.0', '^0.6.0', '^0.6.2']
        - ^0.6.2 (contracts/ERC20MinterPauser.sol#1)
        - ^0.6.0 (contracts/GSN/Context.sol#1)
        - >=0.4.24<0.7.0 (contracts/Initializable.sol#1)
        - ^0.6.2 (contracts/UpgradableCoin.sol#2)
        - ^0.6.0 (contracts/access/AccessControl.sol#1)
        - ^0.6.0 (contracts/math/SafeMath.sol#1)
        - ^0.6.0 (contracts/token/ERC20/ERC20.sol#1)
        - ^0.6.0 (contracts/token/ERC20/ERC20Burnable.sol#1)
        - ^0.6.0 (contracts/token/ERC20/ERC20Pausable.sol#1)
        - ^0.6.0 (contracts/token/ERC20/IERC20.sol#1)
        - ^0.6.2 (contracts/utils/Address.sol#1)
        - ^0.6.0 (contracts/utils/EnumerableSet.sol#1)
        - ^0.6.0 (contracts/utils/Pausable.sol#1)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#different-pragma-directives-are-used
INFO:Detectors:
Pragma version^0.6.2 (contracts/ERC20MinterPauser.sol#1) necessitates versions too recent to be trusted. Consider deploying with 0.5.11
Pragma version^0.6.0 (contracts/GSN/Context.sol#1) necessitates versions too recent to be trusted. Consider deploying with 0.5.11
Pragma version>=0.4.24<0.7.0 (contracts/Initializable.sol#1) allows old versions
Pragma version^0.6.2 (contracts/UpgradableCoin.sol#2) necessitates versions too recent to be trusted. Consider deploying with 0.5.11
Pragma version^0.6.0 (contracts/access/AccessControl.sol#1) necessitates versions too recent to be trusted. Consider deploying with 0.5.11
Pragma version^0.6.0 (contracts/math/SafeMath.sol#1) necessitates versions too recent to be trusted. Consider deploying with 0.5.11
Pragma version^0.6.0 (contracts/token/ERC20/ERC20.sol#1) necessitates versions too recent to be trusted. Consider deploying with 0.5.11
Pragma version^0.6.0 (contracts/token/ERC20/ERC20Burnable.sol#1) necessitates versions too recent to be trusted. Consider deploying with 0.5.11
Pragma version^0.6.0 (contracts/token/ERC20/ERC20Pausable.sol#1) necessitates versions too recent to be trusted. Consider deploying with 0.5.11
Pragma version^0.6.0 (contracts/token/ERC20/IERC20.sol#1) necessitates versions too recent to be trusted. Consider deploying with 0.5.11
Pragma version^0.6.2 (contracts/utils/Address.sol#1) necessitates versions too recent to be trusted. Consider deploying with 0.5.11
Pragma version^0.6.0 (contracts/utils/EnumerableSet.sol#1) necessitates versions too recent to be trusted. Consider deploying with 0.5.11
Pragma version^0.6.0 (contracts/utils/Pausable.sol#1) necessitates versions too recent to be trusted. Consider deploying with 0.5.11
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity
```

Recommendation

Consider lock the pragma version known bugs for the compiler version. When possible, do not use floating pragma in the final live deployment.

3.2.2. POSSIBLE MISUSE OF PUBLIC VARIABLES- INFORMATIONAL

The other finding by Slither is involved with declaring some variables external instead of public. In public functions, array arguments are immediately copied array to memory, while external functions can read directly from calldata. Reading calldata is cheaper than memory allocation.

Public functions need to write the arguments to memory because public functions may be called internally. Internal calls are passed internally by pointers to memory. Thus, function expects its arguments being located in memory when the compiler generates the code for an internal function.

```
INFO:Detectors:
mint(address,uint256) should be declared external:
        - ERC20MinterPauser.mint(address,uint256) (contracts/ERC20MinterPauser.sol#69-72)
pause() should be declared external:
        - ERC20MinterPauser.pause() (contracts/ERC20MinterPauser.sol#83-86)
unpause() should be declared external:
        - ERC20MinterPauser.unpause() (contracts/ERC20MinterPauser.sol#97-100)
initialize(address) should be declared external:
        - UpgradableCoin.initialize(address) (contracts/UpgradableCoin.sol#7-17)
getRoleMemberCount(bytes32) should be declared external:
        - AccessControlUpgradeSafe.getRoleMemberCount(bytes32) (contracts/access/AccessControl.sol#90-92)
getRoleMember(bytes32,uint256) should be declared external:
        - AccessControlUpgradeSafe.getRoleMember(bytes32,uint256) (contracts/access/AccessControl.sol#106-108)
getRoleAdmin(bytes32) should be declared external:
        - AccessControlUpgradeSafe.getRoleAdmin(bytes32) (contracts/access/AccessControl.sol#116-118)
grantRole(bytes32,address) should be declared external:
        - AccessControlUpgradeSafe.grantRole(bytes32,address) (contracts/access/AccessControl.sol#130-134)
revokeRole(bytes32,address) should be declared external:
        - AccessControlUpgradeSafe.revokeRole(bytes32,address) (contracts/access/AccessControl.sol#145-149)
renounceRole(bytes32,address) should be declared external:
        - AccessControlUpgradeSafe.renounceRole(bytes32,address) (contracts/access/AccessControl.sol#165-169)
name() should be declared external:
        - ERC20UpgradeSafe.name() (contracts/token/ERC20/ERC20.sol#75-77)
symbol() should be declared external:
        - ERC20UpgradeSafe.symbol() (contracts/token/ERC20/ERC20.sol#83-85)
decimals() should be declared external:
        - ERC20UpgradeSafe.decimals() (contracts/token/ERC20/ERC20.sol#100-102)
totalSupply() should be declared external:
        - ERC20UpgradeSafe.totalSupply() (contracts/token/ERC20/ERC20.sol#107-109)
balanceOf(address) should be declared external:
        - ERC20UpgradeSafe.balanceOf(address) (contracts/token/ERC20/ERC20.sol#114-116)
transfer(address,uint256) should be declared external:
        - ERC20UpgradeSafe.transfer(address,uint256) (contracts/token/ERC20/ERC20.sol#126-129)
approve(address,uint256) should be declared external:
        - ERC20UpgradeSafe.approve(address,uint256) (contracts/token/ERC20/ERC20.sol#145-148)
transferFrom(address,address,uint256) should be declared external:
        - ERC20UpgradeSafe.transferFrom(address,address,uint256) (contracts/token/ERC20/ERC20.sol#162-166)
increaseAllowance(address,uint256) should be declared external:
        - ERC20UpgradeSafe.increaseAllowance(address,uint256) (contracts/token/ERC20/ERC20.sol#180-183)
decreaseAllowance(address,uint256) should be declared external:
        - ERC20UpgradeSafe.decreaseAllowance(address,uint256) (contracts/token/ERC20/ERC20.sol#199-202)
burn(uint256) should be declared external:
        - ERC20BurnableUpgradeSafe.burn(uint256) (contracts/token/ERC20/ERC20Burnable.sol#28-30)
burnFrom(address,uint256) should be declared external:
        - ERC20BurnableUpgradeSafe.burnFrom(address,uint256) (contracts/token/ERC20/ERC20Burnable.sol#43-48)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#public-function-that-could-be-declared-as-external
```

Recommendation

Consider as much as possible declaring external variables instead of public variables. As for best practices, you should use external if you expect that the function will only ever be called externally and use public if you need to call the function internally. To sum up, all can access to public functions while external functions only can be accessed externally.

3.2.3. ERC CONFORMAL CHECKER- INFORMATIONAL

In addition, another tool by Slither is able to test ERC Token functions. Thus, slither-check-erc20 was performed:

```
# Check UpgradableCoin

## Check functions
[✓] totalSupply() is present
        [✓] totalSupply() -> () (correct return value)
        [✓] totalSupply() is view
[✓] balanceOf(address) is present
        [✓] balanceOf(address) -> () (correct return value)
        [✓] balanceOf(address) is view
[✓] transfer(address,uint256) is present
        [✓] transfer(address,uint256) -> () (correct return value)
        [✓] Transfer(address,address,uint256) is emitted
[✓] transferFrom(address,address,uint256) is present
        [✓] transferFrom(address,address,uint256) -> () (correct return value)
        [✓] Transfer(address,address,uint256) is emitted
[✓] approve(address,uint256) is present
        [✓] approve(address,uint256) -> () (correct return value)
        [✓] Approval(address,address,uint256) is emitted
[✓] allowance(address,address) is present
        [✓] allowance(address,address) -> () (correct return value)
        [✓] allowance(address,address) is view
[✓] name() is present
        [✓] name() -> () (correct return value)
        [✓] name() is view
[✓] symbol() is present
        [✓] symbol() -> () (correct return value)
        [✓] symbol() is view
[✓] decimals() is present
        [✓] decimals() -> () (correct return value)
        [✓] decimals() is view

## Check events
[✓] Transfer(address,address,uint256) is present
        [✓] parameter 0 is indexed
        [✓] parameter 1 is indexed
[✓] Approval(address,address,uint256) is present
        [✓] parameter 0 is indexed
        [✓] parameter 1 is indexed


        [✓] UpgradableCoin has increaseAllowance(address,uint256)
```

Results

All tests were successfully passed.

3.2.4. SECURITY TESTING EXPLOITATION - INFORMATIONAL

TeEther is a tool to perform analysis and automatic exploitation over smart contracts. teEther try to exploit the most common vulnerabilities such as DELEGATE CALL and SELFDESTRUCT on Smart Contracts.

```
ethsec@594e298019ad:/share/teether$ python3 bin/gen_exploit.py test.contract.code 0x1234 0x1000 +1000
INFO:root:Finished all paths
INFO:root:No CALL instructions
INFO:root:No DELEGATECALL instructions
INFO:root:No CALLCODE instructions
INFO:root:No SELFDESTRUCT instructions
WARNING:root:No state-dependent critical path found, aborting
```

Results

All tests were successfully passed, no vulnerabilities were found.

3.3 AUTOMATED SECURITY SCAN - INFORMATIONAL

Description

Halborn used automated security scanners to assist with detection of well-known security issues, and to identify low-hanging fruit on the targets for this engagement. Among the tools used was **MythX**, a security analysis service for Ethereum smart contracts. **MythX** performed a scan on the testers machine and sent the compiled results to the analyzers to locate any vulnerabilities. Security Detections are only in scope, and the analysis was pointed towards issues with ERC20MinterPauser.sol and UpgradableCoin.sol.

Results

MythX detected 0 High findings, 0 Medium, and 6 Low.

0 High		0 Medium		6 Low
ID	SEVERITY	NAME	FILE	LOCATION
SWC-103	Low	A floating pragma is set.	ERC20MinterPauser.sol	L: 1 C: 0
SWC-131	Low	Unused function parameter "name".	ERC20MinterPauser.sol	L: 50 C: 48
SWC-131	Low	Unused function parameter "symbol".	ERC20MinterPauser.sol	L: 50 C: 68
SWC-131	Low	Unused function parameter "from".	ERC20.sol	L: 315 C: 34
SWC-131	Low	Unused function parameter "to".	ERC20.sol	L: 315 C: 48
SWC-131	Low	Unused function parameter "amount".	ERC20.sol	L: 315 C: 60

MythX detected 0 High findings, 0 Medium, and 3 Low.

0 High		0 Medium		6 Low
ID	SEVERITY	NAME	FILE	LOCATION
SWC-103	Low	A floating pragma is set.	UpgradableCoin.sol	L: 2 C: 0
SWC-131	Low	Unused function parameter "name".	ERC20MinterPauser.sol	L: 50 C: 48
SWC-131	Low	Unused function parameter "symbol".	ERC20MinterPauser.sol	L: 50 C: 68
SWC-131	Low	Unused function parameter "from".	ERC20.sol	L: 315 C: 34
SWC-131	Low	Unused function parameter "to".	ERC20.sol	L: 315 C: 48
SWC-131	Low	Unused function parameter "amount".	ERC20.sol	L: 315 C: 60