Universal Dollar Protocol

Security Assessment

February 4th, 2021

For :
Universal Dollar Protocol

By :

Owan Li @ CertiK
guilong.li@certik.org

Bryan Xu @ CertiK
buyun.xu@certik.org

Disclaimer

CertiK reports are not, nor should be considered, an “endorsement” or “disapproval” of any particular project or team. These reports are not, nor should be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts CertiK to perform a security review.

CertiK Reports do not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

CertiK Reports should not be used in any way to make decisions around investment or involvement with any particular project. These reports in no way provide investment advice, nor should be leveraged as investment advice of any sort.

CertiK Reports represent an extensive auditing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. CertiK’s position is that each company and individual are responsible for their own due diligence and continuous security. CertiK’s goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

What is a CertiK report?

- A document describing in detail an in depth analysis of a particular piece(s) of source code provided to CertiK by a Client.
- An organized collection of testing results, analysis and inferences made about the structure, implementation and overall best practices of a particular piece of source code.
- Representation that a Client of CertiK has indeed completed a round of auditing with the intention to increase the quality of the company/product’s IT infrastructure and or source code.

Overview

Project Summary

Project Name	Universal Dollar
Description	an algorithmic open-source stablecoin with reserve asset mechanism built on Ethereum
Platform	Ethereum; Solidity; Yul
Codebase	https://github.com/8quad/u8d-protocol
Commit	7633dee313d846c670e19f01395920516ddc3c0f

Audit Summary

Delivery Date	Feb. 4th, 2021
Method of Audit	Static Analysis, Manual Review
Consultants Engaged	2
Timeline	Jan. 21, 2021 - Jan. 25, 2021, Jan. 30, 2021, Feb. 3, 2021

Vulnerability Summary

Total Issues	8
Total Critical	0
Total Major	0
Total Minor	1
Total Informational	7

Executive Summary

This report has been prepared for **Universal Dollar** smart contract to discover issues and vulnerabilities in the source code of their Smart Contract as well as any contract dependencies that were not part of an officially recognized library. A comprehensive examination has been performed, utilizing Dynamic Analysis, Static Analysis, and Manual Review techniques.

The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts.

This protocal is a fork of Empty Set Dollar.
The audit scope is the differences between this repo and the Empty Set Dollar repo.

Additionally, to bridge the trust gap between project team and community, project team should consider to express a sincere attitude. Project team has the responsibility to notify community with the following capability of the deployment:

- Governance proposal dao can transfer assets in this contract `Pool` under unpredicted cases via 'emergencyWithdraw' method.

The Universal Dollar protocol aims to apply all changes and upgrades to Pool contracts, and core contracts on the DAO, which is managed by on chain governance mechanisms.

File in Scope

ID	Contract	SHA-256 Checksum
CST	**Constants.sol**	62b57907211bdcc9dd743e306cbf57b1ebeb5b66c1bedf5028f0ca0860330c67
BDG	**dao/Bonding.sol**	46a4054507dbf03abe870a549aa162300337b1e93f1efb42218445ea0a2d8edd
CTR	**dao/Comptroller.sol**	c7b2e72311458b6a84d9c9d6bb2d70d1a064cc1f674a362b5d424a0e2953f1f1
CRV	**dao/Curve.sol**	a921a913abf8422ba672a9463e41ad237f8ee22a432d917f6dbcf009190b283c
GET	**dao/Getters.sol**	6939d287ab61320200a8c328db605aa269b7b50d0eceda14cb64a39feb7ff770
GVN	**dao/Govern.sol**	6224679faad11d9efc0e0dd815883a7cbf377c65aaf0d910111ed223df7ace9a
IMP	**dao/Implementation.sol**	d33b6eba805a224678fd4238654cc67f63c66470e51e35078dc2b43db5281e5a
MKT	**dao/Market.sol**	35f3c52426fcca82a12fc19170e82439367cfe3d57daaa87a3824dab6d167530
PMS	**dao/Permission.sol**	5b47f59f2abe6918a8ce827cf9a35771ac0ef65823dbf4e4e9a3cd208ab20b79
RGT	**dao/Regulator.sol**	ab24bbf066bf5a1e6c6cd4ab845fdd118ade674e3730e15bdb6b2c254f651fd5
ROT	**dao/Root.sol**	6d517073ddc344424514bb5a88186821e704e5dbc192aa2c28f0a8ee3df6f8dd
SET	**dao/Setters.sol**	8c3ddeacb7efdae661ab35a1254ec64533be7d8c4079c7147e7cc94f785653b5
STA	**dao/State.sol**	575537a9bf0a8f28931a3b9f0ae6a147075cf09784eb7d1f56815392bd535c6e
UPG	**dao/Upgradeable.sol**	3aa89e2e57a70c12343f4ec592d5f6946a5dcafcc094fea818194e6df5cdbfb8
DPY	**deployment/Deployer.sol**	4e05805602368426e21dbee83c873d4dd028aa4c3f238429a00a5a610ee93d32
DCM	**external/Decimal.sol**	619c19cc264b6521f58c938efcb42bc5854b0c367e5509e2d3fbb19df3e32324
LQD	**oracle/Liquidity.sol**	782db1b05bf472489b78f82b83dd3efd77a9ad32ab90e695a9697490c5b52548
ORA	**oracle/Oracle.sol**	e5374ca09b5dd40d281972785f4283e7756f374ba0ab11d6275ac09ddbd8932d
POL	**oracle/Pool.sol**	6f967bf9eafed13462d8eee496f8f65e1b3e6b4760f62de7f863ca918b59971d
PGT	**oracle/PoolGetters.sol**	eacab8702ebd9fa8aff45036a9f7fedf590faf7c2534d8c260708d353982ae75
PSR	**oracle/PoolSetters.sol**	3e709021cf7bdc121bbfe02a1d7a1785ce49436045c1c84027feb6d294ae5f36
PST	**oracle/PoolState.sol**	d09c9c7a90980c2a97f71c3feae1a4970af73b69312dc462c2a2a83c835447fb
PUP	**oracle/PoolUpgradable.sol**	4dbac7281d3f2c554a640d7f70a7a2d7991fba2e3464f305d1df02378a747e1c
STR	**streaming/Stream.sol**	75fd181574cd8d118192db86f9ec6efcbf05b93476be83892f9428cf257fc7c8
STG	**streaming/StreamingGetters.sol**	c1aa64a7155be1f8ae623d95a029f83cfbb5bb4a0aa5c8f3b83d0ccae173b943
STS	**streaming/StreamingSetters.sol**	81dbf04c333c8286c144ea6e1216e4c2c985d9777d901d7c601d950e8ae4fdbd
DOL	**token/Dollar.sol**	ff7dc6828df73027daeb3251b0878c17b691b3cde7570acd17afa793929d79ad
PMT	**token/Permittable.sol**	1e23527d8be747a5f94be6ca2807da425472f2a6c938ef552e343c78e15ef8cf

Documentation

The sources of truth regarding the operation of the contracts in scope were lackluster and are something we advise to be enriched to aid in the legibility of the codebase as well as project. To help aid our understanding of each contract's functionality we referred to in-line comments and naming conventions.

These were considered the specification, and when discrepancies arose with the actual code behaviour, we consulted with the **Universal Dollar** team or reported an issue.

Review Notes

Certain optimization steps that we pinpointed in the source code mostly referred to coding standards and inefficiencies, however 3 minor vulnerabilities were identified during our audit that solely concerns the specification.

Certain discrepancies between the expected specification and the implementation of it were identified and were relayed to the team, however they pose no type of vulnerability and concern an optional code path that was unaccounted for.

The project has adequate doumentation and specification outside of the source files, also the code comment coverage is detailed.

Recommendations

Overall, the codebase of the contracts should be refactored to assimilate the findings of this report, enforce linters and / or coding styles as well as correct any spelling errors and mistakes that appear throughout the code to achieve a high standard of code quality and security.

Findings

ID	Title	Type	Severity
PMT-01	Incorrect Naming Convention Utilization	Coding Style	Informational
CTR-01	Proper Usage of Function State	Gas Optimization	Informational
CTR-02	`Checks-effects-pattern` Not Used	Implementation	Informational
GET-01	Unlocked Compiler Version Declaration	Language Sepcific	Informational
GET-02	Proper Usage of “public” and “external” type	Gas Optimization	Informational
POL-01	Tautology or Contradiction	Language Sepcific	Informational
IMP-01	Suppression Attack on `advance()` Functionality	Implementation	Minor
BDS-01	Lack of Natspec Comments	Language Sepcific	Informational

PMT-01: Incorrect Naming Convention Utilization

Type	Severity	Location
Coding Style	Informational	Permittable.sol L32

Description:

Solidity defines a naming convention that should be followed. In general, parameters should use mixedCase, refer to: https://solidity.readthedocs.io/en/v0.5.17/style-guide.html#naming-conventions

Variables shoud use mixedCase.
Examples:
Variables like: `EIP712_DOMAIN_SEPARATOR` in `Permittable` contract

Recommendation:

The recommendations outlined here are intended to improve the readability, and thus they are not rules, but rather guidelines to try and help convey the most information through the names of things.

Alleviation:

The development team think it is immutable variable. For constants this style is correct.
They use solidity `0.5.17` which doesn't have immutable keyword.
(Certik - response) The team uses `^0.5.17`, which means any version greater than `0.5.17` (including `0.6.5`) can be utilized. Consider to lock to `0.5.17` just as we recommended in GET-01.

CTR-01: Proper Usage of Function State

Type	Severity	Location
Gas Optimization	Informational	Comptroller.sol L125

Description:

Function `balanceCheck()` can be declared `view` since it does not modify the state.

```
1      function balanceCheck() private {
2          Require.that(
3              dollar().balanceOf(address(this)) >=
   totalBonded().add(totalStaged()).add(totalRedeemable()),
4              FILE,
5              "Inconsistent balances"
6          );
7      }
```

Recommendation:

Consider using the "view" attribute for functions which promise not to modify the state.

Alleviation:

The development team think it's private function, it's not allowed to save a lot of gas. They will keep this in mind but use this function from ESD sources.
(Certik - response) This is a recommendation for best practice.
refer to : https://docs.soliditylang.org/en/v0.5.17/contracts.html?highlight=view#view-functions

CTR-02: `Checks-effects-pattern` Not Used

Type	Severity	Location
Implementation	Informational	Bondings.sol L53 Comptroller.sol L38,L46 Pool.sol L60

Description:

During `deposit()`, `burnFromAccount()`, `redeemToAccount()` function calls state variables for balance are changed after transfers are done. This will lead to reentrancy issue. Although functions from Comptroller.sol contract are internal, it's always good to follow the said pattern.

Recommendation:

It is recommended to follow checks-effects-interactions pattern for cases like this. It shields public functions from re-entrancy attacks. It's always a good practice to follow this pattern. `checks-effects-interactions` pattern also applies to ERC20 tokens as they can inform the recipient of a transfer in certain implementations.

Alleviation:

The development team think they do not use ETH in this function and reentrancy attack is not possible in their case. And also in this function there are checks – `balanceCheck()` function call.
(Certik -response) We agree the development team's opinions, balanceCheck() will help prevent issues. Here what we provide is a best practice of coding style.

ESD team has alleviated this issue. Refer to: https://github.com/emptysetsquad/dollar/blob/2e4d49a24aa70993dfdb19ca4371fb53ed76e445/protocol/contracts/dao/Comptroller.sol

GET-01: Unlocked Compiler Version Declaration

Type	Severity	Location
Language Sepcific	Informational	Getters.sol

Description:

The compiler version utilized throughout the project uses the “^” prefix specifier, denoting that a compiler version which is greater than the version will be used to compile the contracts. Recommend the compiler version should be consistent throughout the codebase.

Recommendation:

It is a general practice to instead lock the compiler at a specific version rather than allow a range of compiler versions to be utilized to avoid compiler-specific bugs and be able to identify ones more easily. We recommend locking the compiler at the lowest possible version that supports all the capabilities wished by the codebase. This will ensure that the project utilizes a compiler version that has been in use for the longest time and as such is less likely to contain yet-undiscovered bugs.

Alleviation:

The development team think it's no matter besause `0.5.17` solc version is the last `0.5` version. And they can not use a next version of Solc.

GET-02: Proper Usage of "public" and "external" type

Type	Severity	Location
Gas Optimization	Informational	Getters.sol, Bonding.sol L92, Dollar.sol L35 Oracle.sol L47,L52, PoolUpgradable.sol L41, PoolGetters.sol, Pool.sol L31

Description:

"public" functions that are never called by the contract could be declared "external" . When the inputs are arrays "external" functions are more efficient than "public" functions.
Examples:
Functions `name()` , `symbol()` , `decimals()` , `oracle()` , `totalNet()` , `balanceOfStaged()` , `balanceOfBonded()` , `balanceOfCoupons()` , `statusOf()` , `allowanceCoupons()` , `streamedFrom()` , `streamedUntil()` , `streamDuration()` ...etc in contract `Getters` .
Function `mint()` in contract `Dollar` .
Function `boostStream()` in contract `Bonding` .
Function `capture()` , `setup()` in contract `Oracle` .
Function `initialize()` in contract `Pool` .
Function `balanceOfClaimable()` , `streamedLpFrom()` , `streamLpDuration()` , `streamLpReleased()` , `streamedRewardFrom()` , `streamRewardDuration()` , `streamRewardTimeleft()` , `streamRewardReleased()` in contract `PoolGetters` .
Function `initialize()` in contract `PoolUpgradable` .

Recommendation:

Consider using the "external" attribute for functions never called from the contract.

Alleviation:

The development team think use external and public for similar function is not convinient for understanding (by other devs).
Example: use streamLpReserved as public but streamLpReleased as external – it confuses the reading.
(Certik -response) ESD reply on this issue:
"These methods are explicitly public so that they can be overridden/mocked for testing".

POL-01: Tautology or Contradiction

Type	Severity	Location
Language Specific	Informational	Pool.sol L163

Description:

Variable `newLpReserved` is uint256, it will be always greater than zero.

```
1  L249 uint256 newLpReserved = unreleasedLpAmount(msg.sender).sub(amountToUnstream, "Pool:
   insufficient balance");
2  L250 if (newLpReserved >= 0) {
```

Recommendation:

We recommend to fix the incorrect comparison by changing the value type or the comparison.

Alleviation:

The development team heeded our advices and resolve this issue in commit 7633dee313d846c670e19f01395920516ddc3c0f.

IMP-01: Suppression attack on `advance()` functionality

Type	Severity	Location
Implementation	Minor	Implementation.sol L40

Description:

Epochs are advanced manually by sending an `advance()` transaction to the DAO, incentivizing users by minting reward U8D tokens to the sender upon successful advancement. As who is first is incentivized, it can lead to `Block stuffing` by an attacker, which can brute-force himself to win U8D reward. This is mostly exploitable where epochs are quicker (1h).

Recommendation:

https://consensys.github.io/smart-contract-best-practices/known_attacks/#mitigations

Alleviation:

The development team thinks the goal here is to get anyone to call the function as soon as an epoch is available not necessarily to guarantee the average user has a fair shot at the advance reward. In practice, they generally have been seeing a bot scoop this up with a decent margin in the first block after the timestamp which is exactly what we intended.

BDS-01: Lack of Natspec Comments

Type	Severity	Location
Language Sepcific	Informational	All internal contracts

Description:

Contract code is missing natspec comments, which helps understand the code and all the functions' parameters.

Recommendation:

Please follow these style guides for adding natspec comments.
https://docs.soliditylang.org/en/v0.5.17/style-guide.html?highlight=natspec%23natspec

Alleviation:

The development team will be fixing the issues in their timeframe.

Appendix

Finding Categories

Gas Optimization

Gas Optimization findings refer to exhibits that do not affect the functionality of the code but generate different, more optimal EVM opcodes resulting in a reduction on the total gas cost of a transaction.

Mathematical Operations

Mathematical Operation exhibits entail findings that relate to mishandling of math formulas, such as overflows, incorrect operations etc.

Logical Issue

Logical Issue findings are exhibits that detail a fault in the logic of the linked code, such as an incorrect notion on how `block.timestamp` works.

Control Flow

Control Flow findings concern the access control imposed on functions, such as owner-only functions being invoke-able by anyone under certain circumstances.

Volatile Code

Volatile Code findings refer to segments of code that behave unexpectedly on certain edge cases that may result in a vulnerability.

Data Flow

Data Flow findings describe faults in the way data is handled at rest and in memory, such as the result of a `struct` assignment operation affecting an in-memory `struct` rather than an instorage one.

Language Specific

Language Specific findings are issues that would only arise within Solidity, i.e. incorrect usage of `private` or `delete` .

Coding Style

Coding Style findings usually do not affect the generated byte-code and comment on how to make the codebase more legible and as a result easily maintainable.

Inconsistency

Inconsistency findings refer to functions that should seemingly behave similarly yet contain different code, such as a `constructor` assignment imposing different `require` statements on the input variables than a setter function.

Magic Numbers

Magic Number findings refer to numeric literals that are expressed in the codebase in their raw format and should otherwise be specified as `constant` contract variables aiding in their legibility and maintainability.

Compiler Error

Compiler Error findings refer to an error in the structure of the code that renders it impossible to compile using the specified version of the project.

Dead Code

Code that otherwise does not affect the functionality of the codebase and can be safely omitted.

Icons explanation

 : Issue resolved

 : Issue not resolved / Acknowledged. The team will be fixing the issues in the own timeframe.

 : Issue partially resolved. Not all instances of an issue was resolved.