AfterDark

Glo Foundation

GLO DOLLAR V3 SECURITY AUDIT

Prepared by **AfterDark**
June 7, 2023
afterdarklabs.xyz

Table of Contents

1. Executive Summary

Project Overview

The Glo Foundation contracted AfterDark to perform a security audit of their Glo Dollar upgrade to version three. Between May 29, 2023 and June 3, 2023 AfterDark auditors inspected the overall design of the application, performed static and dynamic analysis on its codebase, and manually reviewed each line of the in-scope contracts. The goal of this audit was to identify potential vulnerabilities within the provided smart contracts and recommend remediation tactics. An in-depth overview of the audit process and identified vulnerabilities can be seen in the **Audit Overview** and **Findings** sections below.

Finding Analysis

During the audit, AfterDark identified several minor security issues. A full count of the findings and can be seen below:

SEVERITY	COUNT
Critical	**0**
High	**0**
Medium	**0**
Low	**2**
Informational	**3**

Key Areas for Improvement

Based on the results of this audit, AfterDark identified a single area that the Glo Foundation could improve:

1. *Access Control and Role Management*: Both of the low findings in this report are related to access control and role management. One of the privileged roles in the system could benefit from further separation of its duties. Additionally, we recommend the Glo Foundation increase the signing threshold for a couple of multisigs that control privileged roles.

2. Audit Overview

2.1 Scope

The scope was limited to the `contracts/` directory of the https://gitlab.com/global-income-coin/usdglo repository and the commit used for the initial audit was `a7e2c2b8409866fee8b09c9018524a8cd10d29ae`. The finalized commit that includes changes made during the fix-review process is `b5d93f59d7fc533b6a8d422dbe4e1237be0cd1de`. The list of files included in the audit scope can be found below:

```
contracts
├── v2
│   ├── ERC20Upgradeable_V2.sol
└── v3
    ├── ERC20PermitUpgradeable_V3.sol
    └── USDGLO_V3.sol
```

2.2 Testing Methodology

Each component of the in-scope protocol was reviewed in a multi-part process to holistically assess the security of the system and attempt to uncover complex security issues. AfterDark reviewed the design of the protocol, manually reviewed each line of the in-scope codebase, and used automated testing techniques to augment the security audit.

Design Review

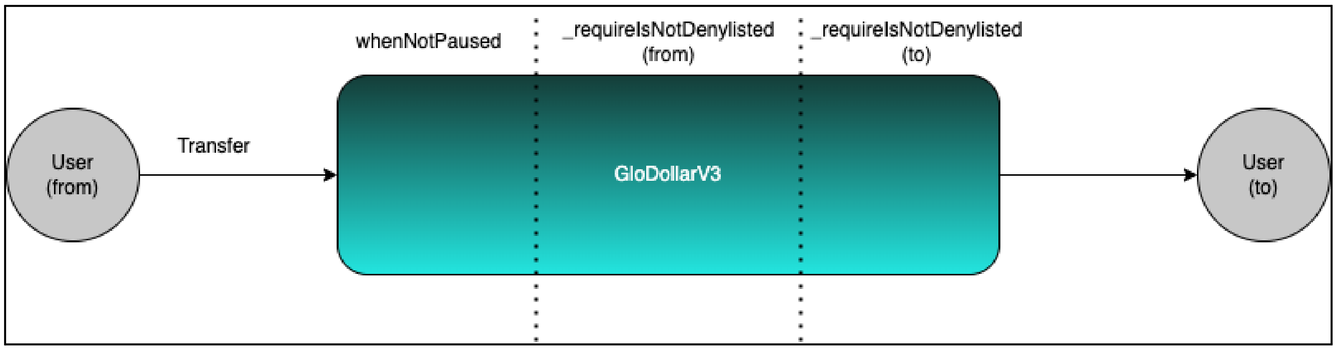
The design review took place throughout the entire audit. As AfterDark auditors made themselves familiar with the protocol, they assessed the protocol's architectural design. Doing so often helps auditors uncover subtle security issues that would otherwise be difficult to spot.

Protocol Summary

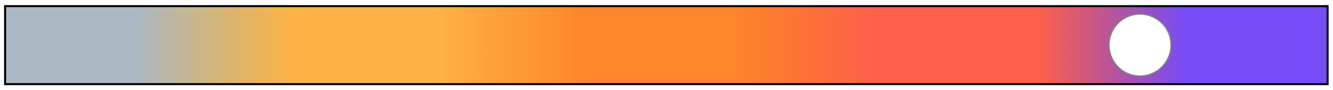
The Glo Dollar is a stablecoin created by the nonprofit Glo Foundation, which states that it will be fully pegged to the U.S. Dollar. The goal of the Glo Dollar project is to reduce extreme poverty by donating 100% of the yield generated by the Glo Dollar reserves to those in need. According to the Glo Foundation, yield will be generated similarly to other stablecoins: through low risk investments such as short-term U.S. treasury bills.

This audit covered the upgrade from version two to version three of the Glo Dollar smart contracts. Both of these contracts are upgradeable ERC20 tokens with access controls for roles that can perform privileged actions such as pausing the contracts, minting new tokens, denylisting addresses, and burning denylisted funds. Version three builds upon the foundation of the previous Glo Dollar contracts by adding off-chain signature functionality in accordance with EIP-2612.

A simplified diagram of a user interacting with the Glo Dollar contract to transfer a token can be seen below:



Centralization Analysis



Decentralized **Somewhat Centralized** **Very Centralized**

Similar to other known stablecoins, the Glo Dollar contracts allow certain parties to perform highly privileged operations that could lead to lost or confiscated funds for users of the protocol. As a result, the project scores highly on the centralization scale, although some efforts have been made by the Glo Foundation to reduce centralization by using multisigs for privileged roles. The full list of roles, their capabilities, and the controls in place to limit their use can be found below:

ROLE	CAPABILITY	CONTROLS
PAUSER	The pauser may elect to halt core functionality of the contract at any time. This would prevent every user from performing a transfer, mint, or burn operation until the pauser lifts the pause.	2 of 3 Multisig Mainnet: 0xeBE0ef4cF72e9ACf37F4337355b56427a09C8F89 Polygon: 0x87f8a6F1743DBD7C7afDe4

		Cf3B5a5Cefb4f9DAAD All 3 members are co-founders of the Glo Foundation.
MINTER	The minter has the permission to mint new funds to arbitrary accounts as well as to burn funds from its own account.	1 of 3 Multisig on Mainnet 2 of 3 Multisig on Polygon Mainnet: 0x18216283a5045E8719D0F8B3617Ab6B14fC4d479 Polygon: 0xf8D6c2162136Fe646A4a9f0BeC40EbCac96E4AA4 All 3 members are co-founders of the Glo Foundation.
DENYLISTER	The denylister can add individual users to a denylist, preventing them from transferring or receiving tokens in any way. The denylister also has the ability to burn the entire balance of a denylisted user and to remove a user from the denylist.	1 of 3 Multisig on Mainnet 2 of 3 Multisig on Polygon Mainnet: 0x1135A985908639b5a218B351d16A61e084CFF7a1 Polygon: 0xfE6E709BAC23af91DAB0ecF7e7290D71D9A1ed23 All 3 members are co-founders of the Glo Foundation.
UPGRADER	The upgrader is able to upgrade the contract to any implementation of its choice. The implementation contract could be created such that the upgrader could perform any number of sensitive operations such as stealing user tokens.	2 of 3 Multisig Mainnet: 0xDf04400bBE27Cbe51F53737AFC446c04F355cC5B Polygon: 0xAf6916a2564619F017BE365848cB078177e55726

		All 3 members are co-founders of the Glo Foundation.
ADMIN	The admin role is able to grant or revoke any of the above roles and their associated capabilities to or from other users including itself.	2 of 3 Multisig Mainnet: 0x284797b8dA4909755FCA06Fa02BF81c0dae9a0E3 Polygon: 0x03BcC9362F97Ec8027BA495cfFb21dd36b3c610D All 3 members are co-founders of the Glo Foundation.

Design Checklist

The below checklist covers several areas of design that AfterDark takes into account during the engagement:

CATEGORY	SUMMARY	RESULT
Libraries	The Glo Dollar project made extensive use of OpenZeppelin's upgradeable contracts including: `UUPSUpgradeable`, `PausableUpgradeable`, `AccessControlUpgradeable`, `Initializable` and implemented two custom contracts that drew heavily from `ERC20Upgradeable` and `ERC20PermitUpgradeable`.	**Strong**
Integrations	N/A	**N/A**
Documentation	In addition to a well-documented, public github repository, the Glo Foundation also provided AfterDark with a publicly accessible notion page and google doc outlining the system and the upgrades they have made since the previous version of Glo Dollar.	**Strong**
Testing	Both hardhat and foundry tests were available within the project. These tests had solid coverage of codebase functionality and included fuzzing for some of the unit tests.	**Strong**
Complexity	While much of the complexity is kept in check due to the use of well-known public libraries, some is introduced by the denylist mechanism. No direct vulnerabilities were discovered related to denylisting during this audit, however the risks of using the uppermost bit of a user's balance to determine inclusion in a denylist are high. Since ideas for a similar denylist mechanism have been conceived in another notable project, the rating is given as satisfactory.	**Satisfactory**

Storage Slot Analysis

Since this audit covered an upgrade between versions of the Glo Dollar, AfterDark checked the storage slots did not contain any collision issues. AfterDark used both Hardhat and Foundry to test the storage slots of each version of the Glo Dollar. No issues regarding storage slot collisions were discovered. A table depicting the variables at each slot and their inclusion in each version of the Glo Dollar can be seen below:

STORAGE SLOT	STATE VARIABLE	V3	V2	V1
0	_initialized	✅	✅	✅
0	_initializing	✅	✅	✅
1	__gap	✅	✅	✅
51	_balances	✅	✅	✅
52	_allowances	✅	✅	✅
53	_totalSupply	✅	✅	✅
54	_name	✅	✅	✅
55	_symbol	✅	✅	✅
56	__gap	✅	✅	✅
101	_paused	✅	✅	✅
102	__gap	✅	✅	✅
151	__gap	✅	✅	✅
201	_roles	✅	✅	✅
202	__gap	✅	✅	✅
251	__gap	✅	✅	✅
301	__gap	✅	✅	✅
351	_HASHED_NAME	✅		
352	_HASHED_VERSION	✅		
353	__gap	✅		
403	_nonces	✅		
404	_PERMIT_TYPEHASH_DEPRECATED_SLOT	✅		

405	__gap	✅		

Manual Review

A significant portion of testing occurred during this phase. During the manual review, the auditors looked at every single line of code within the in-scope contracts and considered language specific issues, issues related to the execution environment for the code, as well as the protocol's intended business logic. A majority of the findings in this report were discovered during the manual review process.

Automated Review

AfterDark also included static analysis and dynamic analysis as part of the security audit of the Glo Dollar V3 contracts.

Static Analysis

Static analysis was performed using a combination of Slither and simple regular-expression pattern matching searches on the codebase. Potential issues flagged from these tools were triaged and none were included in the report that were not already discovered during the manual review portion.

Dynamic Analysis

AfterDark focused on two separate areas during dynamic analysis of the project: expanding the existing test suite and testing the upgrade process.

Expanding the Existing Test Suite

Fuzz Testing

While testing certain edge cases within the protocol, AfterDark auditors added a few fuzz tests to the `USDGLO.t.sol` file. A snippet containing one of the added fuzz tests can be seen below:

```
    function testCanStillPerformActionsWhilePaused(uint256 mintAmount1, uint256
mintAmount2, uint256 mintAmount3) public {
        mintAmount1 = bound(mintAmount1, 0, MAX_ALLOWED_SUPPLY / 4);
        mintAmount2 = bound(mintAmount2, 0, MAX_ALLOWED_SUPPLY / 4);
        mintAmount3 = bound(mintAmount3, 0, MAX_ALLOWED_SUPPLY / 4);
        uint256 mintAmount4 = mintAmount1 + mintAmount2 + mintAmount3;

        //giving address(100) address(200) and the minter their mintAmounts
        vm.startPrank(minter);
        usdglo.mint(address(100), mintAmount1);
        usdglo.mint(address(200), mintAmount2);
        usdglo.mint(minter, mintAmount3);
        vm.stopPrank();

        //pause the contract and check it was paused
        assertEq(usdglo.paused(), false);
        vm.prank(pauser);
        usdglo.pause();
        assertEq(usdglo.paused(), true);

        //giving address(100) mintAmount usdglo (Mint)
        vm.startPrank(minter);
        vm.expectRevert("Pausable: paused");
        usdglo.mint(address(100), mintAmount4);

        //burning minter mintAmount usdglo (Burn)
        vm.expectRevert("Pausable: paused");
        usdglo.burn(mintAmount3);
        vm.stopPrank();

        //sending 100 usdglo to minter (Transfer)
        vm.startPrank(address(100));
        vm.expectRevert("Pausable: paused");
        usdglo.transfer(minter, mintAmount1);

        //setting up approval for transferfrom
        usdglo.approve(address(200), mintAmount1);
        vm.stopPrank();

        //sending 1e6 usdglo to minter (TransferFrom)
        vm.startPrank(address(200));
        vm.expectRevert("Pausable: paused");
        usdglo.transferFrom(address(100), minter, mintAmount1);
        vm.stopPrank();

        //destroying denylist funds !THIS SHOULD WORK!
        vm.startPrank(denylister);
        usdglo.denylist(address(100));
        usdglo.destroyDenylistedFunds(address(100));
    }
```

The entire modified test file will be provided along with this report should the Glo Foundation find the added tests of use.

Invariant Testing

After learning from the Glo Foundation that a critical vulnerability affecting balances had gone unnoticed in the first version of the Glo Dollar, AfterDark created a couple of invariant tests to reduce the likelihood of such a vulnerability affecting the codebase in the future. A snippet of the invariant tests can be seen below:

```
    function invariant_sumOfBalancesIsAlwaysTotalSupply() public {
        address[] memory actors = usdgHandler.actors();
        uint256 actorsLen = actors.length;

        uint256 sumOfBalances;

        for(uint256 i; i < actorsLen; ++i) {
            sumOfBalances = sumOfBalances + usdglo.balanceOf(actors[i]);
        }

        assertEq(sumOfBalances, usdglo.totalSupply());
    }

    function invariant_sumOfBalancesIsNeverMoreThanMaxAllowed() public {
        address[] memory actors = usdgHandler.actors();
        uint256 actorsLen = actors.length;

        uint256 sumOfBalances;

        for(uint256 i; i < actorsLen; ++i) {
            sumOfBalances = sumOfBalances + usdglo.balanceOf(actors[i]);
        }

        assertLe(sumOfBalances, MAX_ALLOWED_SUPPLY);
    }
```

The full test file and all of the other necessary files and imports for running these invariant tests will be provided along with this report. With this setup, more invariants can be added as desired by the Glo Foundation to improve the robustness of their test suite.

Testing the Upgrade Process

Deployment Script Testing

The Glo Foundation expressed interest in deploying and upgrading to version three of the Glo Dollar following this audit. To help make this a smooth upgrade for the Glo Foundation, AfterDark reviewed their deployment script and then later used a modified version of it to perform fork testing of the Glo Dollar before and after upgrading.

AfterDark was unable to complete every step of the upgrade process using the existing deployment scripts as they integrated with OpenZeppelin Defender and required the associated permissions to manually create and execute a proposal through Defender. Instead, AfterDark created a modified version of the deployment script to simulate the upgrade on a forked instance of the Ethereum mainnet. The modified script can be seen below:

```
import * as dotenv from "dotenv";

import { ethers, upgrades } from "hardhat";
const hre = require("hardhat");
import {
  DefenderRelayProvider,
  DefenderRelaySigner,
} from "defender-relay-client/lib/ethers";

async function main() {
  const upgradeContractName = process.env.UPGRADE_CONTRACT_NAME as string;
  const proxyAddress = process.env.PROXY_ADDRESS as string;

  const UPGRADER_ACCOUNT = "0xDf04400bBE27Cbe51F53737AFC446c04F355cC5B";

  const [alice] = await ethers.getSigners();

  await alice.sendTransaction({
    to: UPGRADER_ACCOUNT,
    value: ethers.utils.parseEther("10.0"), // Sends exactly 1.0 ether
  });

  const credentials = {
    apiKey: process.env.RELAYER_DEPLOYER_KEY as string,
    apiSecret: process.env.RELAYER_DEPLOYER_SECRET as string,
  };
  const provider = new DefenderRelayProvider(credentials);
  const relaySigner = new DefenderRelaySigner(credentials, provider, {
    speed: "fast",
  });

  await hre.network.provider.request({
    method: "hardhat_impersonateAccount",
    params: [UPGRADER_ACCOUNT],
  });
  const signer = await ethers.provider.getSigner(UPGRADER_ACCOUNT)

  const UpgradedUSDGlobalIncomeCoin = await ethers.getContractFactory(
    upgradeContractName,
    signer
  );

  const upgraded = await upgrades.upgradeProxy(proxyAddress,
UpgradedUSDGlobalIncomeCoin)
  console.log(`upgraded: ${upgraded}`);
}

if (require.main === module) {
  dotenv.config();
  main()
    .then(() => process.exit(0))
    .catch((error) => {
      console.error(error);
      process.exit(1);
    });
}
```

In conjunction with this part of the review, AfterDark modified part of the existing test suite to target the upgraded contracts on the forked mainnet.

Fork Testing

To check that the upgrade went smoothly and that the existing test suite would apply to a more “live” version of the upgraded contract, AfterDark took several of the base foundry tests and modified them to apply generally to a contract that already had existing state changes. This was then used to target the newly deployed Glo Dollar V3 on the forked mainnet.

The below steps can be used to create a forked instance of Ethereum mainnet, upgrade the Glo Dollar contract to version three using the script from the section above, and then run the fork test suite:

```
npx hardhat node --fork $ETH_RPC_URL
npx hardhat run --network localhost scripts/forceImport.ts
npx hardhat run --network localhost scripts/prepareUpgrade2.ts
forge test --match-contract USDGLO_Fork --rpc-url http://127.0.0.1:8545
```

A snippet of one of the fork tests used can be seen below:

```
     /// forge-config: default.fuzz.runs = 256
    function testMint(address from, uint256 amount) public {
        amount = bound(amount, 0, MAX_ALLOWED_SUPPLY);
        uint256 beforeSupply = usdglo.totalSupply();
        uint256 beforeBalance = usdglo.balanceOf(from);
        vm.startPrank(minter);

        if (from == address(0)) {
            vm.expectRevert(bytes("ERC20: mint to the zero address"));

            usdglo.mint(from, amount);
        } else if (amount >= MAX_ALLOWED_SUPPLY - beforeSupply) {
            bytes4 selector = bytes4(keccak256("IsOverSupplyCap(uint256)"));
            vm.expectRevert(
                abi.encodeWithSelector(selector, usdglo.totalSupply() + amount)
            );

            usdglo.mint(from, amount);
        } else {
            usdglo.mint(from, amount);

            assertEq(usdglo.totalSupply(), beforeSupply + amount);
            assertEq(usdglo.balanceOf(from), beforeBalance + amount);
        }

        vm.stopPrank();
    }
```

All of the files needed to perform the upgrade and fork testing as outlined in the above sections will be provided along with this report.

3. Findings

3.1 Low Findings

L-01 Multiple Privileges Assigned to Single Role

Summary

There are multiple privileged roles within the protocol that assigned specific sets of capabilities. In accordance with the principle of least privilege, it is recommended that no single role be allowed to perform tasks beyond what is necessary. According to documentation, only the `MINTER_ROLE` should be allowed to mint `USDGLO` and burn `USDGLO`. The `DENYLISTER_ROLE`, however, is also able to burn a user's `USDGLO` by first adding them to the denylist.

Impact

Low: This issue does not pose a direct security risk, but it conflicts existing documentation and may reduce the robustness of the protocol to withstand operational security issues.

Exploitability

Medium: In order for this issue to be abused, a key must be compromised or an existing key holder must act against the protocol's interests. The exploitability is raised to medium due to another existing issue: **L-02 Single Threshold Multisig**.

Technical Details

The `DENYLISTER_ROLE` can add funds to a denylist by calling `denylist()`. From there, the `DENYLISTER_ROLE` can effectively burn those funds by calling `destroyDenylistedFunds()`.

Recommendation

Consider separating the privilege for burning and adding to a denylist. One way this could be achieved is by allowing the `MINTER_ROLE` to call `destroyDenylistedFunds()` instead of the `DENYLISTER_ROLE`.

Client Response

"Acknowledged, but as this is not an issue that introduces high risk we prefer to postpone making this change until our next smart contract upgrade & audit cycle."

L-02 Single Threshold Multisig

Summary

The Glo Foundation uses multisigs to manage several different privileged roles. During the audit, the multisigs that controlled the `MINTER_ROLE` and `DENYLISTER_ROLE` on Ethereum mainnet held a signing threshold of one. This significantly reduces the efficacy of the multisigs as there are three different owners that can unilaterally perform privileged operations on the Glo Dollar deployed to mainnet.

Impact

Low: This issue is not a vulnerability in and of itself, but it does both increase the centralization of the protocol and decrease its operational security.

Exploitability

Medium: The potential for misuse or damage done to a protocol via key compromise increases when any of several keys can perform privileged actions.

Technical Details

AfterDark auditors checked the threshold of each of the documented multisigs when reviewing the design and centralization of the protocol. It was discovered that all but two of the multisigs had a 2 of 3 signing requirement. These two instead had a 1 of 3 requirement.

```
cast call 0x1135A985908639b5a218B351d16A61e084CFF7a1 "getThreshold()" --rpc-url
$ETH_RPC_URL
0x0000000000000000000000000000000000000000000000000000000000000001

cast call 0x18216283a5045E8719D0F8B3617Ab6B14fC4d479 "getThreshold()" --rpc-url
$ETH_RPC_URL
0x0000000000000000000000000000000000000000000000000000000000000001
```

Recommendation

Consider increasing the threshold for these multisigs to a value higher than one.

Client Response

"The minter multisig will soon be changed to a 4/6 with 3 new members coming from our issuing partner. All other multisigs will become at least 2 of 3s, and we're working towards at least 3 of 5 within the next 12 months."

3.2 Informational Findings

I-01 Confusing Event Parameter Name

Summary

When a user has their denylisted funds destroyed, an event is emitted that specifies who called the function, which user had their funds destroyed, and how many tokens were destroyed. This event uses a slightly ambiguous name for the person who called the function.

Technical Details

When the internal `_destroyDenylistedFunds()` is called, it emits the `DestroyDenylistedFunds` event. The event parameters can be seen below:

```
address indexed denylister,
address indexed denylistee,
uint256 indexed amount
```

Even though the first parameter is the `msg.sender` who performs the destruction of the funds, the parameter shows them as the `denylister`. This can lead to confusion as it is unclear if the parameter represents the address who initially added the `denylistee` to the list or if it represents a different address that is specifically responsible for destroying the funds.

Recommendation

Consider using a less ambiguous name such as `destroyer` for the first parameter in the event.

Client Response

“Acknowledged, but as this is not an issue that introduces any risk we prefer to postpone making this change until our next smart contract upgrade & audit cycle.”

I-02 Unnamed Imports

Summary

Throughout the codebase, the imports did not explicitly name what was being imported from the other contracts and libraries.

Technical Details

Unnamed imports such as the ones seen below reduce readability of the codebase and may even lead to incorrect assumptions regarding which names are being used when dealing with longer inheritance chains and multi-contract files:

```
import "@openzeppelin/contracts-upgradeable/security/PausableUpgradeable.sol";
import
"@openzeppelin/contracts-upgradeable/access/AccessControlUpgradeable.sol";
import "@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol";
import "@openzeppelin/contracts-upgradeable/proxy/utils/UUPSUpgradeable.sol";
import "../v2/ERC20Upgradeable_V2.sol";
import "./ERC20PermitUpgradeable_V3.sol";
```

Recommendation

We recommend modifying the imports to explicitly show what is being imported from each file for example:

```
import {PausableUpgradeable} from
"@openzeppelin/contracts-upgradeable/security/PausableUpgradeable.sol";
```

Client Response

“Acknowledged, but as this is not an issue that introduces any risk we prefer to postpone naming the imports until our next smart contract upgrade & audit cycle.”

I-03 Outdated version of OpenZeppelin Library

Summary

An outdated version of the OpenZeppelin library was used by the project. While there were no associated vulnerabilities identified, it's best practice to use the latest versions of these libraries.

Technical Details

The project made use of an OpenZeppelin library and code from version 4.7.0. Certain updates available in newer versions, such as the finalization of `ERC20Permit` may benefit the codebase.

Recommendation

Consider updating to the current version of the OpenZeppelin library for both `contracts` and `contracts-upgradeable`.

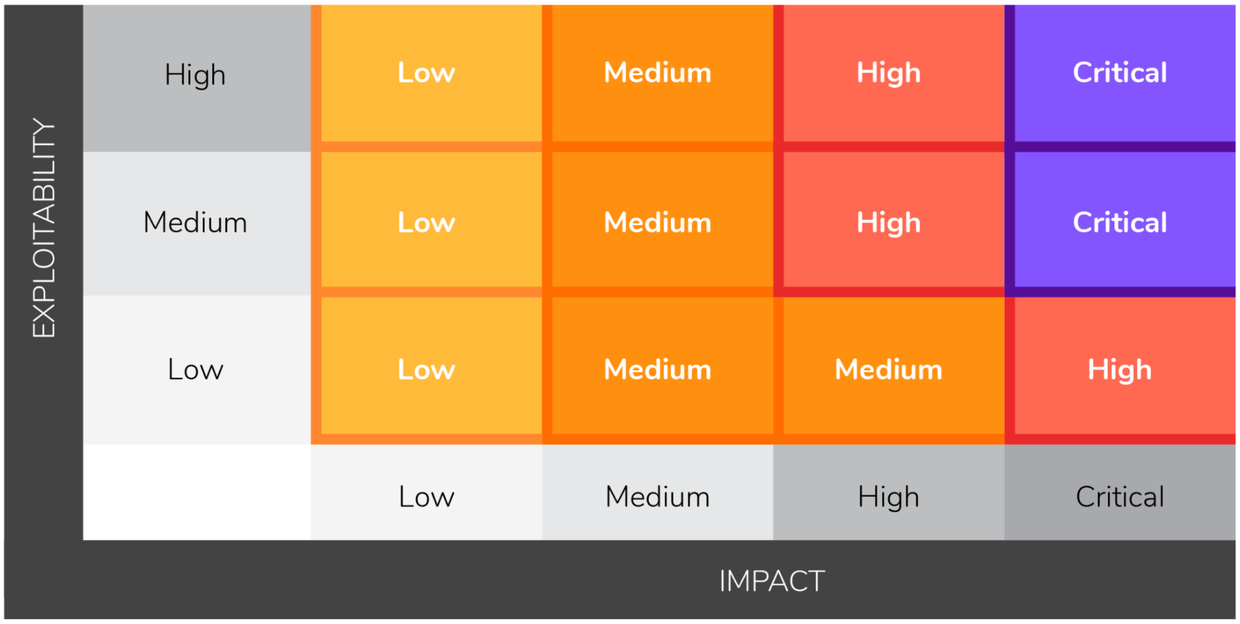
Client Response

"Acknowledged, but as this is not an issue that introduces any risk we prefer to postpone updating the libraries until our next smart contract upgrade & audit cycle."

4. Appendices

4.1 Appendix A: Finding Severity Evaluation Matrix

This appendix contains the evaluation matrix used to determine the severity of findings included in this report.



EXPLOITABILITY \ IMPACT	Low	Medium	High	Critical
High	Low	Medium	High	Critical
Medium	Low	Medium	High	Critical
Low	Low	Medium	Medium	High

See below for example lists of how impact and exploitability may be determined. Note that this list is not exhaustive and that ultimately the impact and exploitability assigned to a finding are at the discretion of the auditing team.

Example Impact Definitions

- **Critical Impact**: Significant loss of funds, prolonged denial of service
- **High Impact**: Loss of funds, denial of service
- **Medium Impac**t: Minimal loss of funds, significant potential integration issues
- **Low Impact**: Potential integration issues, best practices

Example Exploitability Definitions

- **High Exploitability**: Few or no preconditions are required to perform this exploit
- **Medium Exploitability**: Some preconditions are required to perform the exploit, but they can be reasonably met within the application’s lifetime
- **Low Exploitability**: Many preconditions are required to perform the exploit and/or the preconditions are nontrivial to meet

4.2 Appendix B: Project Contacts

NAME	ROLE	CONTACT
Roderic Deichler	CSO	roderic@afterdarklabs.xyz
Jackson Kelley	Auditor	@sjkelleyjr

Thank you for choosing AfterDark

We are a team of experts with extensive experience in **blockchain technology** and **cybersecurity.**

Our combination of technical knowledge and industry expertise allows us to provide clients with comprehensive security solutions for their blockchain applications. From **security advising** during ideation to **smart contract auditing** before deployment, we are dedicated to making our clients' applications safer every step of the way.

Let us help you build a **secure and trusted blockchain ecosystem.**

Visit us at **afterdarklabs.xyz**

AfterDark