thirdweb A-42

Security Audit

September 29th, 2022
Version 1.0.0

Presented by Zellic

Table of Contents

- Introduction
- Overall Assessment
- Specification
- Source Code
- Issue Descriptions and Recommendations
- Security Levels Reference
- Review Notes on Vesting20.sol and Sale.sol
- Disclaimer

Introduction

This document includes the results of the security audit for thirdweb's smart contract code as found in the section titled "Source Code". The security audit was performed by the Macro security team from August 22, 2022 to September 2, 2022.

The purpose of this audit is to review the source code of some thirdweb Vesting contracts and some Sale contracts, to identify any security issues, to verify that the source code and specification are satisfying the correctness and security of the contracts, and to provide recommendations.

Disclaimer: While Macro's review is comprehensive and has surfaced some changes that should be made to the source code, this audit should not solely be relied upon for security, as no single audit is guaranteed to catch all possible bugs.

Overall Assessment

The following is an aggregation of issues found by the Macro Audit team:

Severity	Found	Acknowledged	Won't Fix	Addressed
Critical	0	-	-	-
Medium	1	-	-	-
Low	1	-	-	-
Code Quality	2	-	-	-
Informational	1	-	-	-
Gas Optimizations	1	-	-	-

thirdweb was quick to respond to these issues.

Specification

Our understanding of the specification was based on the following sources:

- Discussions on Slack with the thirdweb team.
- The official Vesting, Marketplace documentation and more specifically provided documentation for contracts which were in scope of the performed audit.
- Tokenvesting.

Source Code

The following source code was reviewed during the audit:

- **Repository**: github.com/thirdweb
- **Commit Hash**: 1d31f2a2c1c5d7f8f7b6d9f9f6b2f7f2f1a7c7c8d0

Specifically, we audited the following contracts at commit 1d31f2a2c1c5d7f8f7b6d9f9f6b2f7f2f1a7c7c8d0:

Contract	SHA256
contracts/drop1155/Drop1155.sol	3d9f7c2f0c1f6c2c2f7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a
contracts/BaseRouter.sol	5d2a2d9f8e1d7c2f7a9f1c2f7d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f
contracts/BaseFactory.sol	2a1f7d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a
contracts/BaseFactoryFactory.sol	2a1f7d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a
contracts/extension/interfaces/IPlatformFee.sol	2a1f7d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a
contracts/extension/PlatformFee.sol	2a1f7d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a
contracts/extension/PlatformFee/PlatformFee.sol	2a1f7d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a
contracts/extension/interfaces/IPlatformFee.sol	2a1f7d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a
contracts/extension/PlatformFee.sol	2a1f7d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a
contracts/extension/PlatformFee/PlatformFee.sol	2a1f7d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a
contracts/extension/interfaces/IPlatformFee.sol	2a1f7d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a

We audited the following contracts as part of ThirdwebTOKEN contract audit:

Contract	SHA256
contracts/drop1155/Drop1155.sol	3d9f7c2f0c1f6c2c2f7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a
contracts/BaseRouter.sol	5d2a2d9f8e1d7c2f7a9f1c2f7d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f
contracts/BaseFactoryFactory.sol	2a1f7d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a
contracts/extension/interfaces/IPlatformFee.sol	2a1f7d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a
contracts/extension/PlatformFee.sol	2a1f7d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a
contracts/extension/PlatformFee/PlatformFee.sol	2a1f7d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a
contracts/extension/interfaces/IPlatformFee.sol	2a1f7d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a
contracts/extension/PlatformFee.sol	2a1f7d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a
contracts/extension/PlatformFee/PlatformFee.sol	2a1f7d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a
contracts/extension/interfaces/IPlatformFee.sol	2a1f7d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a
contracts/lib/TWStrings.sol	4f1a7d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a
contracts/lib/TWAddress.sol	7f1a9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a
contracts/lib/MerkleProof.sol	3d9f7c2f0c1f6c2c2f7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a
contracts/lib/TokenVesting.sol	9d2a2d9f8e1d7c2f7a9f1c2f7d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f

We audited the following contracts as part of ThirdwebTOKEN contract audits:

Contract	SHA256
contracts/drop1155/Drop1155.sol	3d9f7c2f0c1f6c2c2f7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a
contracts/BaseRouter.sol	5d2a2d9f8e1d7c2f7a9f1c2f7d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f
contracts/BaseFactoryFactory.sol	2a1f7d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a
contracts/extension/interfaces/IPlatformFee.sol	2a1f7d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a
contracts/extension/PlatformFee.sol	2a1f7d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a
contracts/extension/PlatformFee/PlatformFee.sol	2a1f7d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a
contracts/extension/interfaces/IPlatformFee.sol	2a1f7d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a
contracts/lib/TWStrings.sol	4f1a7d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a
contracts/lib/TWAddress.sol	7f1a9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a
contracts/lib/MerkleProof.sol	3d9f7c2f0c1f6c2c2f7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f2c2a
contracts/lib/TokenVesting.sol	9d2a2d9f8e1d7c2f7a9f1c2f7d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f
contracts/lib/TokenERC20Vesting.sol	1a2d7c2f9e1d7c2f7a9f1c2f7d9f1c2f2c2a1c7f8d9f1c2f2c2a1c7f8d9f1c2f

Note: This document contains an audit of the Vesting contracts listed above. Specifically, the audit portrays only to the contracts themselves, and does not pertain to any other programs or tools including deployment scripts.

Issue Descriptions and Recommendations

Here on an issue is going to, in an order down to rear after us:

CODEREVIEWQUALITY	Non-ruining vulnerability
	TokenERC20.changes price when quantity is 0
	Native tokens can get locked in the contract
	Use `_safeMint` instead of `_pMint`
	`initializeOwner` is not included
	`DistributionEmission` and `distributeAmount()` can serve as DoS
	Documentation discrepancy for price in TokenERC20.sol
	Inconsistency between Token contracts regarding pause/unpause functionality
	TokenERC20 can have empty URI
	Inconsistency among different contracts
	Useless parameters
	GAS OPTIMIZATIONS `bool` state value

Security Level Reference

We quantify issues in three parts:

- The high-impact/low-likelihood impact of the issue:
  - How bad things can get (for a vulnerability)
  - The significance of an improvement (for a code quality issue)
  - The amount of gas saved (for gas optimizations)
- The high/medium/low **likelihood** of the issue:
  - How likely is the issue to occur (for a vulnerability)
- The overall critical/high/medium/low **severity** of the issue.

This third part - the severity level - is a summary of how much consideration the Client should give to fixing the issue, based on the weight severity according to the table of guidelines below:

Severity	Description
0 - 4 Critical	We recommend the client **must** fix the issue, no matter what, because not fixing would mean **significant liability towards WILL be lost.**
5 - 9 High	We recommend the client **must** address the issue, no matter what, because not fixing would mean **significant liability towards will be lost.** There is a small chance that the issue is against the intended business logic
10 - 14 Medium	We recommend the client to **seriously consider** fixing the issue as not fixing it would be a bad thing that can be good enough to impact the project significantly, either not in an economical manner.
15 - 19 Low	The risk is small, unlikely, or may not relevant to the project in a meaningful way.
0 - 4 Code Quality	Whether or not the project wants to develop a fix is up to the goodwill and desire of the project.
0 - 4 Informational	We recommend the client could give you obvious risk, but fixing or not fixing is up to the goodwill of the project. Not fixing should always be intentional, or even certain aspects of protocol design or business model.
0 - 4 Gas Optimizations	Savings and things to keep in mind when operating the protocol, so monetary not security related.

The presented optimization/suggestions would save an unknown amount of gas/execution energy in exchange for, to the worst-case disappointment, a bit of maintenance.

Issue Details

M-01 `mintWithSignature()` **Front-running vulnerability**

Severity	Status	Impact	Likelihood
Medium	Found	High	High

In TokenERC20, TokenERC721 and TokenERC1155, `mintWithSignature()` allows the MintRequest's `to` address to be zero:

```
function mintWithSignature(MintRequest calldata _req, bytes calldata _signature
    address recoverAddr = _recoverAddress(_req, _signature);
    ...
}
```

The reference allows a minter to construct a mint signature such that the sender will receive the tokens, instead of needing to specify the recipient's address.

However, because the recipient is the zero address, an attacker can frontrun a `mintWithSignature()` transaction, receiving the tokens and leaving the victim with nothing.

If there is no price, the attacker gets the tokens for free. This is particularly bad when the tokens have a market price or an AMM, and the victim's owner wanted to grant someone else the tokens.

If a price is present, the attacker does have to pay it. But the situation is still problematic if the price was a discount, or if they are able to frontrun a large amount of transactions.

Note that this behavior also exists in Signature20.sol.

Consider forcing MintRequest's `to` address to always be defined.

M-02 TokenERC20.changes price when quantity is 0

Severity	Status	Impact	Likelihood
Medium	Found	Medium	Medium

In TokenERC20, `mintWithSignature()`'s function, when mint request is defined with `quantityLimit`, nothing is checked for the specified `_req.address` but:

- The price is not changed.
- The mint request is recorded as a success (`_mintWithSig.success = true`).

Although an account with a minter role would need to create this mint request, it's not uncommon for such roles to be huge capacities to run off their contracts.

Consider adding a check in the `_initializeMint` function to mint when quantity is 0:

```
_mintWithSig.success = true;
```

Also consider to add the above require statement to TokenERC721. Although this contract doesn't charge the price as quantity is considered in calculation, it still records the mint request as a success.

M-03 Native tokens can get locked in the contract

Severity	Status	Impact	Likelihood
Low Cases	Found	Medium	Low

In TokenERC20, TokenERC721 and TokenERC1155 mint requests can include a `price` and a `currency`, which can be either `NATIVE_TOKEN` or a ERC20 address. If the price is zero:

Suppose the currency specified in the request is some ERC20 token and the sender, `_req.recipient`, `_req.primarySaleRecipient`, and `_req.royaltyRecipient` are all the same address, the contract "refers (same)" to be minted tokens to the contract itself, instead of the recipient.

Consider adding a check in the `collectPrice()`'s function which prevents these edge cases by the contract when currency is not set to `NATIVE_TOKEN`:

```
if (_currency == Currency(address(0), NATIVE_TOKEN)
    && _currency.transferFrom(_from, _to, _amount, _data)
    // else
    ...
}
```

M-04 Use `_safeMint` instead of `_pMint`

Severity	Status	Impact	Likelihood
Coding Standards	Found	High	Medium

TokenERC721 and TokenERC1155 use `_pMint` instead of `_safeMint` to mint the actual tokens:

```
_pMint(_to, tokenIdMint);
```

This can lead to tokens being locked in those contracts with no recovery.

Consider using `_safeMint` instead of `_pMint` to check if a contract recipient can receive the tokens and avoid of accepting ERC721 tokens.

L-01 `initializeOwner()` **is not bounded**

Severity	Status	Impact	Likelihood
Input Ranges	Found	Low	Low

In TokenERC20, TokenERC721 and TokenERC1155, the `initialize` functions don't have a boundary check for `_initializeOwner`.

Also in the `setPlatformFeeInfo()`, function, there are no checks for `platformFeeBps` in `MAX_BPS` (10,000). If set to `MAX_BPS`, it's very close to 100percentage of the fee, but minting would fail because of underflow in case Thirdweb fee is applied. See following fee:

```
BuiltinTokenERC20.sol:_transferTokensOnClaim()
    primarySaleRecipient,
    _quantityToClaim,
    primarySaleRecipient,
    _currentClaimPhase.pricePerToken * _quantityToClaim
    );
```

Consider setting a reasonable upper limit for `platformFeeBps`.

L-02 `DistributionEmission` **is not bounded**

Severity	Status	Impact	Likelihood
Input Ranges	Found	Low	Low

In TokenERC20, TokenERC721 and TokenERC1155, the `initialize` functions don't have a boundary check for `_distributionEmission`.

Also in the `setDistributionEmission()`, function, and `setDistributionEmission()`, in all contracts, there are no checks for `_distributionEmission`. If set to `MAX_BPS` (10,000) or more, it will cause underflow in `_distributeAmount()`.

As a consequence, when using `NATIVE_TOKEN` as `_currency`, ether could be burned in the zero address.

In the case of an ERC20 transfer, the transaction would be reverted for `OpenZeppelin SafeERC20.sol`.

Consider adding checks to not allow address 0x0 for `primarySaleRecipient` and `platformFeeRecipient`.

L-03 Documentation discrepancy for price in TokenERC20.sol

Severity	Status	Impact	Likelihood
Code Quality	Found	Medium	Medium

Thirdweb's Signature20 documentation defines any of the mint request parameters as `pricePerToken`.

However, in TokenERC20.sol, the price specified in the mint request is considered the total price (based).

This also creates an inconsistency with TokenERC721.sol, which uses `pricePerToken` as documentation.

Consider:

1. Updating TokenERC20.sol to use `pricePerToken` instead of `price` as accepting `mintWithSignature` function accordingly to include a `quantity` as parameter or
2. Document the differing behavior of TokenERC20.

L-04 Inconsistency between Token contracts regarding pause/unpause functionality

Severity	Status	Impact	Likelihood
Code Quality	Found	Medium	Medium

In TokenERC20 there is this possibility to pause/unpause token transfers, in contrary, TokenERC721 and TokenERC1155 don't provide this functionality.

Consider to add pause/unpause capability for token transfers in TokenERC721 and TokenERC1155.

L-05 TokenERC721 can have empty URI

Severity	Status	Impact	Likelihood
Code Quality	Found	Medium	Medium

In TokenERC1155, `lazyMint` function the minting results when the `_baseURI` parameter is `MintRequest`'s empty:

```
if (bytes(_baseURI).length == 0
    && _nextTokenId == _startId
    && _amount == 1
    && _data.length == 0
    && _to != address(0)) {
    return;
}
```

In contrast, TokenERC721 doesn't check for empty uri parameter.

Consider adding this check to TokenERC721 as well.

L-06 Inconsistency among different contracts

Severity	Status	Impact	Likelihood
Code Quality	Found	Low	Low

In TokenERC20 and TokenERC1155 define their own `NATIVE_TOKEN` constant:

```
address private constant NATIVE_TOKEN = address(0xeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee);
address private constant NATIVE_TOKEN = address(0);
```

However, TokenERC721 uses `DistributionTokenType.NATIVE_TOKEN`.

- The `collectPrice()`, function is written differently in TokenERC20 than TokenERC721 and TokenERC1155. Although the functionality is the same, the code is not as specific, problematic. In addition, TokenERC20's version can be tricked to determine `_currency` within `distributeAmount()` instead of `collectPrice()`.
- The `setPlatformFeeInfo()`, function is written differently in TokenERC20 than TokenERC721 and TokenERC1155. Although the functionality is the same, the code is not as specific, problematic. In addition, TokenERC20's version can be tricked to determine `platformFeeType` within `distributeAmount()` instead of `collectPrice()`.

and *TokenERC1155*.

- `_afterTokenTransfer()` is only overridden in *TokenERC20* without any changed behavior.
- `MAX_BPS` is declared as:
  - `uint128 internal constant` in *TokenERC20*.
  - `uint256 private constant` in *TokenERC721* and in *TokenERC1155*.
- `platformFeeBps` is declared as:
  - `uint128 internal` in *TokenERC20*.
  - `uint128 public` in *TokenERC721* and in *TokenERC1155*.
- *TokenERC721* and *TokenERC1155* both inherit from OpenZeppelin's `ReentrancyGuardUpgradeable` and call its `__ReentrancyGuard_init()`. However, *TokenERC20* does call this initializer, even though it inherits the same contract.

G-4 Calldata parameters

TOPIC	STATUS	GAS SAVINGS
Gas Optimization	Fixed	Low

In *TokenERC721* and *TokenERC1155* function `collectPrice` uses `memory` keyword for the `MintRequest` parameter. Consider using `calldata` allocation instead to save gas costs. In *TokenERC721* and *TokenERC1155* function `collectPrice` uses `memory` keyword for the `MintRequest` parameter. Consider using `calldata` allocation instead to save on gas cost.

I-1 `draft-EIP712Upgradeable.sol` draft status

TOPIC	IMPACT
Informational	Informational *

See OZ's statement regarding draft status here:
https://docs.openzeppelin.com/contracts/3.x/api/drafts

Review Notes on VoteERC20.sol and Split.sol

Scope: VoteERC20.sol and Split.sol

For these two contracts, we opted to do a review instead of a full audit, due there being no test suite and low documentation. However, we were happy to provide suggestions and considerations in this way.

VoteERC20.sol

(1) Both `proposalIndex` and `proposalId` are used

Resolution: Wont Do.

The two common options to store proposals on-chain in DAO proposal systems are:

1. Store all function calls and arguments on-chain.
2. Store a 32 byte hash of all function calls on-chain, and post the source of that hash off-chain (e.g. on a public forum).

The former leans more towards decentralization, as anyone can read on-chain data to verify the proposed function calls at any time.

On the other hand, the latter saves on gas costs, as storing function calls on-chain can get expensive.

VoteERC20 takes the approach of both:

1. It uses Open Zeppelin's GovernorUpgradeable contract, which uses approach #2.
2. It extends `propose()` to store all calls in its own proposals variable (approach #1), including the 32 byte hash from the parent OZ contract.

Doing both is redundant, which will result in increased gas costs for no benefit. It may also cause confusion, as there are arguably two ids: the 32 byte hash, and the proposal index.

Consider taking one approach or another. If approach #1 is desired, consider using Compound's Governor Bravo contracts. Otherwise, it may be sufficient that the proposed function calls are emitted in an event.

Split.sol

(1) DoS vulnerability in `distribute` function

Resolution: Wont Do.

In *Split* contract, `distribute` function loops through each payee address and sends assets performing checks and reverting if any of these calls fails. If one of the payees reverts the transfer the entire `distribute` function will revert in every call. This can also be intentionally done by a smart contract address as a payee.

The vulnerability is not that bad, since anyone is able to call `release()` for their own account at any time. However, consider documenting the dangers 3rd party contracts or other thirdweb contracts relying on this function.

Also consider not reverting on failed transfers, if atomicity is not important.

(2) OpenZeppelin's version of PaymentSplitterUpgradeable.sol has been updated

Resolution: Updated preset version to reflect OZ changes.

Split contract inherits from `PaymentSplitterUpgradeable`. It is not referencing the OpenZeppelin version directly but instead is using a preset version:

```
import "./openzeppelin-presets/finance/PaymentSplitterUpgradeable.sol";
```

OpenZeppelin did some slight changes recently on this contract and has added `releasable` functions. See here: https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable/commit/b392c249e2c72434c438e0e495af1bacbc6cfd4f#diff-ec6a964c766447336ea6842ff3214ac85da5144ccf44d92e100d3d4acfd907c4

Consider to update the preset version to include those latest changes, as `releasable()` is a useful function.

(3) Inconsistent zero check

Resolution: The ThirdWeb fee was removed.

After calling `thirdwebFee.getFeeInfo()`, the Split contract checks to see if their values are not equal to zero. This seems to be inconsistent with the rest of the codebase.

If this check is important, consider reevaluating other parts of the codebase that do not have this check.

(4) Inaccurate comments

Resolution: Removed related comments.

On `distribute` function, *Split* contract has comments referring to an function named `_appPay` but it does not seem to exist in the codebase.

```
function distribute() public virtual {
    uint256 count = payeeCount();
    for (uint256 i = 0; i < count; i++) {
        ****// note: `_release` should not fail because payee always has shares
        _release(payable(payee(i)));
    }
}

function distribute(IERC20Upgradeable token) public virtual {
    uint256 count = payeeCount();
    for (uint256 i = 0; i < count; i++) {
        ****// note: `_release` should not fail because payee always has shares
        _release(token, payee(i));
    }
}
```

Disclaimer

Macro makes no warranties, either express, implied, statutory, or otherwise, with respect to the services or deliverables provided in this report, and Macro specifically disclaims all implied warranties of merchantability, fitness for a particular purpose, noninfringement and those arising from a course of dealing, usage or trade with respect thereto, and all such warranties are hereby excluded to the fullest extent permitted by law.

Macro will not be liable for any lost profits, business, contracts, revenue, goodwill, production, anticipated savings, loss of data, or costs of procurement of substitute goods or services or for any claim or demand by any other party. In no event will Macro be liable for consequential, incidental, special, indirect, or exemplary damages arising out of this agreement or any work statement, however caused and (to the fullest extent permitted by law) under any theory of liability (including negligence), even if Macro has been advised of the possibility of such damages.

The scope of this report and review is limited to a review of only the code presented by the Emergent team and only the source code Macro notes as being within the scope of Macro's review within this report. This report does not include an audit of the deployment scripts used to deploy the Solidity contracts in the repository corresponding to this audit. Specifically, for the avoidance of doubt, this report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. In this report you may through hypertext or other computer links, gain access to websites operated by persons other than Macro. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such websites' owners. You agree that Macro is not responsible for the content or operation of such websites, and that Macro shall have no liability to your or any other person or entity for the use of third party websites. Macro assumes no responsibility for the use of third party software and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.