ThirdWeb A-1

Security Audit

April 26th, 2022

Version 1.0.0

Prepared by

0xMacro.com

macro

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 Mar 14, 2022 to Apr 15, 2022.

The purpose of this audit is to review the source code of certain thirdweb Solidity contracts, and provide feedback on the design, architecture, and quality of the source code with an emphasis on validating the correctness and security of the software in its entirety.

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

We identified a few issues of low to high severity. thirdweb was quick to respond and fix these issues.

Specification

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

- Discussions on Slack with the thirdweb team.
- The official website, developer docs, and in-repo docs.

Source Code

The following source code was reviewed during the audit:

Repository	Commit
[Github]()	6b6aea60731fb4dcb78cae3fa3dd34782884bce7

Specifically, we audited the following contracts:

Contract	Sha256
contracts/drop/DropERC721.sol	7eb79f82b1e3dcdcc3d43900a397150d020ee98e2210aba80330fc50990cd43d
contracts/lib/MerkleProof.sol	cf3d021220b40ba34a503595000419df6576fabb4309dc3c265abe4ad21a25c8
contracts/lib/FeeType.sol	3d2ede585eb7e37872a0f3566a143f5b2aa586873160966d34c98963015f622d
contracts/interfaces/IWETH.sol	09e1104223d0b83a346c98102eafec96916c44f53c8c3eef13e1806149943bfb
contracts/lib/CurrencyTransferLib.sol	052c1c014b8169fdb02a9daa37b5edfbbbf9c883d89fcfe4ea3717810fecc76c
contracts/openzeppelin-presets/metatx/ERC2771ContextUpgradeable.sol	4ef0ce1601048c10a4b0fdc3247062be8f1a9ca0441c862ddfadc16251a31edb
contracts/interfaces/ITWFee.sol	4c57ef2e5572551ee29ec7ecfcb67932f152f7b0ffd1e5c84e0976f577eb43c5
contracts/interfaces/drop/IDropERC721.sol	32c1b993e77dc0a516a0a26eb076ce093d3f927f14d93e3f11112ae037e6e01b
contracts/interfaces/drop/IDropClaimCondition.sol	c00b24db96810b5e45a6e9147a3c02c8d0efb5360c7f8dbc9f2d2bff8f1ad52a
contracts/interfaces/IThirdwebOwnable.sol	616716b979cc688a58956278c7e28073e98e0eb0384435b5f3551adfcd27a6a0
contracts/interfaces/IThirdwebRoyalty.sol	2928dd51da718dc211340aac39231a6f6eea51cce2d2a1529f6f2058bd1e8939

Contract	Sha256
contracts/interfaces/IThirdwebPrimarySale.sol	78d189e4e669b38d60c15877ef5f24b0e7bad4be6f0e411ad840336d47c084fe
contracts/interfaces/IThirdwebPlatformFee.sol	d988667a8274e6b7c7b8ec0d9ea6821bef11c47dacf51c74e1b84d773518d309
contracts/interfaces/IThirdwebContract.sol	8fc9d29ddee99b052ccdc521c272ee4df8a7de0e1754bfcba397dc5cdfa18c72
contracts/marketplace/Marketplace.sol	7b7e50f1b4cbf1a14c0eba079ff4f8f9e7d302533023d8f2a5604e51143ec81d
contracts/interfaces/marketplace/IMarketplace.sol	8937ba859dc1d29332fec6ecfd4df5f2ad95150c2eb6ab7386ca4d627aad814e

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

Methodology

The audit was conducted in several steps.

First, we reviewed in detail all available documentation and specifications for the project, as described in the 'Specification' section above.

Second, we performed a thorough manual review of the code, checking that the code matched up with the specification, as well as the spirit of the contract (i.e. the intended behavior). During this manual review portion of the audit we primarily searched for security vulnerabilities, unwanted behavior vulnerabilities, and problems with systems of incentives.

Third, we performed the automated portion of the review consisting of measuring test coverage (while also assessing the quality of the test suite) and evaluating the results of various symbolic execution tools against the code.

Lastly, we performed a final line-by-line inspection of the code – including comments –in effort to find any minor issues with code quality, documentation, or best practices.

Issues Descriptions and Recommendations

Severity Level Reference

Level	Description
High	The issue poses existential risk to the project, and the issue identified could lead to massive financial or reputational repercussions. We highly recommend fixing the reported issue. If you have already deployed, you should upgrade or redeploy your contracts.
Medium	The potential risk is large, but there is some ambiguity surrounding whether or not the issue would practically manifest. We recommend considering a fix for the reported issue.
Low	The risk is small, unlikely, or not relevant to the project in a meaningful way. Whether or not the project wants to develop a fix is up to the goals and needs of the project.
Code Quality	The issue identified does not pose any obvious risk, but fixing it would improve overall code quality, conform to recommended best practices, and perhaps lead to fewer development issues in the future.
Gas Optimizations	The presented optimization suggestion would save an amount of gas significant enough, in our opinion, to be worth the development cost of implementing it.

[H-01] Marketplace bidder not refunded in edge case

~~HIGH~~ *Fixed by d847e4665cd11eef0b4eb843acb9c7f8a22e3f10*

Marketplace.sol's `handleBid` function escrows the highest bid. When a new bid arrives, the previous highest bid is refunded to its respective bidder.

However, when the **final** bid is a **buyout** bid, the previous highest bid does not get refunded. Refund logic is only applied for non-buyout bids.

Consider moving the previous bid refund logic to a code path that applies to all bids.

[M-01] Unsafe ERC-20 transfers

~~MEDIUM~~ *Fixed by fd82e10e81e7cfe26ed2f1c3415ff78d18618de2*

CurrencyTransferLib's safeTransferERC20, safeTransferERC20WithBalanceCheck, and safeTransferNativeTokenWithWrapper each use a normal function call to initiate a transfer. However, they only check if the function call returns `true` (lines 74, 94, 116).

Although this is correct according to the ERC-20 spec, unfortunately not all ERC-20 contracts behave this way. Many will instead have no return data, which will cause the current `require` statements to prevent those contracts from being used as currency. See this link for more context.

Consider using OpenZeppelin's SafeERC20 library, which considers this scenario, to make thirdweb's contracts more broadly compatible with the ecosystem.

[M-02] Claim condition race condition

~~MEDIUM~~ *Fixed by 221a4a321fe72527088b67f10e5ab17c12f206c1*

DropERC721's `setClaimConditions` receives an array of claim conditions. This array is then assigned to the `phases` of `claimCondition` (the contract's claim condition list).

Each claim condition in `_phases` has a value for `supplyClaimed`, which becomes the assigned value for the new array of phases. However, this exposes a race condition: If the admin intends for a condition's `supplyClaimed` to remain unchanged, a `claim` transaction that occurs just before theirs will cause a discrepancy.

Illustration of race condition:

- Alice the admin sets a claim condition with supply `0` and max supply `10`.
- Others mint until `supplyClaimed` is `4`.
- Alice, wanting to limit the max supply to `8`, sends a transaction with `supplyClaimed` as `4` and `maxClaimableSupply` as `10`.
- While this transaction is in the mempool, Bob also sends a claim transaction.
- Bob's transaction happens to be ordered before Alice's transaction.
- In Bob's transaction, supplyClaimed gets set to `5`, but Alice's transaction sets it back to `4` (!)

Consider, when **updating** conditions, assigning each **updated** phase's `supplyClaimed` to its respective **previous** version's value, so that the caller does not have to manually manage this value.

[M-03] Certain ERC-20 incompatibility

~~MEDIUM~~ *Fixed by 704a2bf0a2a93a6b1600068b765dafaf96d4c28f*

CurrencyTransferLib's `safeTransferERC20WithBalanceCheck` requires that the full amount sent is the amount received. This is incompatible with tax-on-transfer tokens (e.g. PAX Gold), and rebasing tokens.

Consider removing this check to make the marketplace compatible with more currencies. Note that removing this check is only safe if you switch to using OpenZeppelin's SafeERC20 library or similar.

[M-04] Lack of expiration for offers

~~MEDIUM~~ *Fixed by 8825283ed412a7526d3897c193ec89d8f1fd49b3*

Marketplace.sol's offers do not have an expiration date. Consider adding expiration to protect offers from old, forgotten offers that later become bad for the offeror and good for the lister due to market forces.

[L-01] Listing startTime validation

~~LOW~~ *Fixed by b487bfac9acf5346bf41a5614759c2b807415f0c*

In Markeplace.sol’s `createListing()`, `_params.startTime` is valid even if it is far behind block.timestamp. This could result in an unwanted listing if the user accidentally chooses the wrong date, e.g. next week but in a previous year, causing the listing to become active immediately instead of in the future.

Consider validating that `_params.startTime` is “close enough” to block.timestamp when set in the past, perhaps at most one hour ago. This also helps protect against certain signature phishing attacks – important since Marketplace implements ERC-2771 – since old signatures to create listings will eventually expire.

[L-02] Dangerous offer currency validation

~~LOW~~ *Fixed by 960bcc6c02ac411cf0893f6ef00e8556ba4b89a6*

Marketplace.sol’s `offer` function has the following code:

```
// A bid to an auction must be made in the auction's desired currency.
newOffer.currency = targetListing.currency;
```

This line allows offers that specify the wrong currency to be accepted. In general, it could lead to unexpected results from the user’s perspective. In the worst case, the user sends ETH with this transaction (with the wrong `0xEee...` value as currency), and then, due to this line, also sends tokens from the *correct* currency, effectively paying the bid price twice, and not getting refunded for the extra ETH.

Consider `require`’ing these addresses to be the same, instead of reassigning to the correct value.

[L-03] ETH lost in listing offer edge case

~~LOW~~ *Fixed by e430b8aae7299d231d1bb8709736cf2f28c7c887*

Marketplace.sol's `offer` function accepts `NATIVE_TOKEN` as its `_currency` parameter to support handling native ETH. However, its prerequisites differ between its two use cases:

1. When handling a bid on an auction, it **auto-wraps** ETH sent via **msg.value**.
2. When handling an offer on a listing, msg.value is **ignored**, while WETH must be transfer-approved to the contract instead.

Because of the latter, any ETH sent with an offer to a listing will be lost to the sender.

Although this has low likelihood (the sender must have already transfer-approved Marketplace on WETH), consider adding a `require` so that msg.value must equal to zero when making an offer to a listing.

[L-04] platformFeeBps range not checked in initialize

~~LOW~~ *Fixed by e430b8aae7299d231d1bb8709736cf2f28c7c887*

In Marketplace.sol's initialize function, `_platformFeeBps` is not validated to be under 10,000 like it is in `setPlatformFeeInfo`.

Consider adding this check to guarantee consistent behavior across all usage.

[Q-01] Quantity per transaction limit UX

~~CODE QUALITY~~ *Fixed by ef37414aedf883e09b800f48e745571bb9355eaa*

DropERC721's `claim()` function validates `_quantity` twice; once against `currentClaimPhase.quantityLimitPerTransaction`, and once against `_proofMaxQuantityPerTransaction`.

The former is general and the latter is specific. However, the code is written such that the general takes precedence over the specific. While this isn't strictly wrong, it may be undesirable if someone, having a merkle proof that permits minting a quantity **greater** than the general limit, is unable to claim at all.

Consider giving `_proofMaxQuantityPerTransaction` precedence, to make the behavior more intuitive for end project owners.

[G-01] Variable packing improvement

A main advantage of Solidity's variable packing – say, two variables packed – is only accessing a single storage slot in a function call instead of two (cold SLOADs have cost of 2100 gas, whereas a warm SLOAD only costs 100 gas).

`royaltyBps` and `platformFeeBps` appear to have been defined adjacently in an attempt to take advantage of variable packing. However, the two are never accessed in the same functional call, so this cost savings never occurs.

Consider making these two variables `uint16`s and arranging them as follows:

```
address private platformFeeRecipient;
uint16 private platformFeeBps;

address private royaltyRecipient;
uint16 private royaltyBps;
```

[G-02] Modifier refactor

Solidity modifiers "copy/paste" your code across multiple functions. Take the following example:

```
modifier onlyListingCreator(uint256 _listingId) {
    require(listings[_listingId].tokenOwner == _msgSender(), "!OWNER");
    _;
}

function foo() external onlyListingCreator {}
function bar() external onlyListingCreator {}
```

In this case, `foo` and `bar` will **each** contain the full code of the `require()` statement.

Consider using a helper function to reduce code duplication and contract code size:

```
modifier onlyListingCreator(uint256 _listingId) {
    _onlyListingCreator(_listingId);
    _;
}

function _onlyListingCreator(uint256 _listingId) internal {
  require(listings[_listingId].tokenOwner == _msgSender(), "!OWNER");
}

function foo() external onlyListingCreator {}
function bar() external onlyListingCreator {}
```

This can become more important as you upgrade your contracts with more code.

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.