Report for:

Example

November 2021

Version: 1.0

Prepared By: Extropy.IO
Email: info@extropy.io
Telephone: +44 1865261424

Table of Contents

1. Executive Summary

Extropy was contracted to conduct a code review and vulnerability assessment

1.1. Assessment Summary

Phase	Description	Critical	High	Medium	Low	Info	Total
1	Initial Audit	0	0	0	8	4	12

2. Using This Report

To facilitate the dissemination of the information within this report throughout your organisation, this document has been divided into the following clearly marked and separable sections.

Executive Summary	Management level, strategic overview of the assessment and the risks posed to the business
Technical Summary	An overview of the assessment from a more technical perspective, including a defined scope and any caveats which may apply
Technical Findings	Detailed discussion (including evidence and recommendations) for each individual security issue which was identified
Methodologies	Audit process and tools used

2.1. Disclaimer

The audit makes no statements or warranty about utility of the code, safety of the code, suitability of the business model, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bug free status. The audit documentation is for discussion purposes only

2.2. Client Confidentiality

This document contains Client Confidential information and may not be copied without written permission.

2.3. Proprietary Information

The content of this document should be considered proprietary information. Extropy gives permission to copy this report for the purposes of disseminating information within your organisation or any regulatory agency.

Document Version Control			
Data Classification	Client Confidential		
Client Name			
Document Title			
Author	Extropy Audit Team		

Document History			
Issue No.	**Issue Date**	**Issued By**	**Change Description**
1.0	19/11/2021	Laurence Kirk	Released to client

3. Technical Summary

3.1. Scope

This audit is of code at commit 027
and includes contracts

3.2. Design

The contracts implement an auction along with later validation of items.

4. Technical Findings

The remainder of this document is technical in nature and provides additional detail about the items already discussed, for the purposes of remediation and risk assessment.

.

5. Issues Found

5.1. Remove code duplication

Risk Rating	Low

Affects : ContractA

Description:

Function _getBid could be integrated as part of the cBid function in order to remove some code duplication. It seems unnecessary to have an internal function calling another internal function given that it is only ever called once by the same parent function, i.e. _getBid() is only ever called by cBid(). This way we could remove unnecessary code duplication, such as if (node != 0) that is repeated. The new merged function would look like the following:

```
function cBid() internal returns (uint64) {
    node = bidByBidder[bidder]; // bidder -> bid
    if (node != 0) {
        (Id, nT, pT) = _decode(node);
        bidders.remove(node);
        delete bidByBidder[_msgSender()];
        _refundBidder(_msgSender(), a * b);
        return bidderId;
    } else {
        return 0;
    }
}
```

Another note on coding, on line 123 delete bidByBidder[_msgSender()]; is perhaps not needed, considering it gets overridden anyway when adding a new bid in _addBid(), and is never used for any security check anyway such as require(bidByBidder[_msgSender()].

Improving even further, the calling function setBid() should check if the node exists first end then call cBid only if it does exist, e.g.

```
node = bidByBidder[bidder];
if (node != 0) {
        cBid();
} else {
        bidId = ++currentBatch.nextBidderId;
        bidderById[bidderId] = _msgSender();
}
```

this way we can also remove the (node != 0) check from the internal functions since if (bidId == 0) is handled by the else case, i.e. if node does not exist then bidId doesn't exist either.

5.2. Bidding Incentives

Risk Rating	Informational

Affects : ContractA

Description:

It seems that bids are encoded to a 128 bit uint and sorted by bid price per token (leftmost 64 bits) and number of tokens (rightmost 64 bits) to purchase in the bid.

```
uint256 position = bidders.getSlot(address(this), getValue(node));
bidders.insertBefore(position, node);
```

In theory, one could set a bid with price equal to someone else's but with a higher number of tokens, knowing that if the number of tokens left for sale is lower than the number of tokens in the offer, the difference will get refunded to the user anyway. This can be easily calculated in advance, the attacker needs to wait until the last moment before the auction expires (if tokens aren't sold out already) and place their bid, then call end() or wait until it gets called.

5.3. Unclear Error Message

Risk Rating	Low

Affects : ContractA

Description:

On line 252 in the _addBid() function.

```
uint256 node = _encodeBid(bidderId, numTokens, pricePerToken);
require(!bidders.nodeExists(node), "PANIC!! This should not happen!");
```

The error message does not give any extra information about the error or what is required'

It is not clear whether this require statement is even needed since "it should not happen!" in which case a we could use assert rather than require.
Additionally, since _addBid() can only be called by setBid(), and setBid() already calls cBid() deleting any existing bid for the bidder, therefore bidders.nodeExists(node) would return false anyway.

Resolution
Remove the require statement if it is not needed.

5.4. Function setBid() does not check for the number of tokens

Risk Rating	Low

Affects : ContractA

Description:

There is no check to enforce that the number of tokens the user set in the bid is lower than the number of tokens in the “current batch”, such check would be like the one implemented for the purchase() function:

```
require(numTokens <= currentBatch.numTokensAuctioned, "Too many tokens for direct purchase");
```

Note: this is probably a false positive, if it is done on purpose in order to allow multiple bids to be set; the extra amounts of tokens in any bid will be refunded if tokens sell out.

5.5. Unusual inheritance hierarchy

Risk Rating	Low

Affects : ContractA

Description:

It is unusual for A.sol contract to inherit ContractA.sol and not the opposite. It would be expected that the NFT contract would be as simple as possible and more complex contracts such as governance, auction or marketplace contract to inherit or interface with the NFT contract.
Also, looking at the deployment script deployContractA.js we can see that ContractA is deployed separately before ContractA, in fact the address of ContractA is passed to ContractA in the constructor.

Resolution
By inspecting the code it seems that A only needs the interface in order to interact with ContractA, therefore the simplest fix would be to save the interface as its own file in the contracts folder and import that instead of the entire ContractA contract, in other words:

```
// import "./ContractA.sol";
import './INFA.sol';
```

Additionally, in ContractA, import the newly created interface and delete the one starting on line 10 in order to remove code duplication.

5.6. Function end() has no access control

Risk Rating	Low

Affects : ContractA

Description:

The end function can be called by any address. If this is not the desired behaviour, modifiers should be used to restrict access to the function.

5.7. Approval Race protection

Risk Rating	Low

Affects : **ERC20**

Description:

Some tokens, such as USDT do not allow approving an amount > 0 if there is an existing amount approved.

References : Approval Race

Recommendation :

There is little consensus on this issue, to address it, add a check for an existing approval and revert if not setting the approval to zero.

5.8. ContractA doesn't have a receive function.

Risk Rating	Low

Affects : ContractA

Description:

ContractA is intended to receive ether but doesn't have a receive function.

Resolution

If ether is accepted it is also good practice to have a maximum amount that can be accepted.

5.9. Unbounded loop

Risk Rating	Low

Affects : ContractA

Description:

The loop in function mintcould result in an out of gas error if array tokenIds has more than 535 elements.

5.10. ContractA bytecode exceeds maximum size

Risk Rating	Low

Affects : ContractA

Description:

This contract may be not be deployable on main net due to the limit introduced in Spurious Dragon.

Resolution

Consider enabling the optimizer (with a low “runs” value), turning off revert strings, or using libraries.

5.11. Storage optimisation

Risk Rating	Informational

Affects : ContractA

Also the packing in the struct data could be improved

```
struct data {
uint64 Id;
address bddress;
uint64 nT;
uint128 pT;
}
```

It would be better to pack the two uint64s (16 bytes) with the uint128 (16 bytes) to fill the 32 byte slot and have address in its own slot. The way the struct is written now it would take up x3 32byte slots. With the repacking suggested it would only take up x2 32byte slots in storage.

5.12. The contracts are not pausable

Risk Rating	Informational

Affects : Multiple contracts

Description:

It is good practice to allow contracts to be paused in the case of unexpected behaviour. This has to be weighed against issues of centralisation.

Resolution

Use the Open Zeppelin pausable library.

5.13. Unnecessary parameter in event

Risk Rating	Informational

Applies to ContractA

Description:

emit Mint(tokenId, uri);
This emit could be have tokenID field removed since tokenID is incorporated to the uri

6. ERC20 Checklist Items

The checklist detailed in section 8.2 was followed, in summary the results are

.

Issue	Status
Returns bool after transfer	No Issue
Prevent transferring tokens to the 0x0 address	No Issue
Prevent transferring tokens to the contract address	No Issue
Re entrant Calls	No Issue
Pausable	No Issue
Fee on transfer	No Issue
Balance Modifications Outside of Transfers	No Issue
Upgradable tokens	No Issue
Flash Mintable tokens	No Issue
Tokens with Blocklists	No Issue
Revert on Zero Value Transfers	See details
Low decimals	See details
High Decimals	No Issue
Decimals returns uint8	No Issue
No Revert on Failure	No Issue
Revert on Large Approvals & Transfers	No Issue
Code Injection Via Token Name	No Issue
Supply owned by single owner	See details
Approval Race protection	See details

7. Tool List

The following tools were used during the assessment:

Tools Used	Description	Resources
SWC Registry	Vulnerability database	https://swcregistry.io/

8. General Audit Goals

We audit the code in accordance with the following criteria:

Sound Architecture

This audit includes assessments of the overall architecture and design choices. Given the subjective nature of these assessments, it will be up to the development team to determine whether any changes should be made.

Smart Contract and Rust Best Practices

This audit will evaluate whether the codebase follows the current established best practices for smart contract development.

Code Correctness

This audit will evaluate whether the code does what it is intended to do.

Code Quality

This audit will evaluate whether the code has been written in a way that ensures readability and maintainability.

Security

This audit will look for any exploitable security vulnerabilities, or other potential threats to the users.

Although we have commented on the application design, issues of crypto-economics, game theory and suitability for business purposes as they relate to this project are beyond the scope of this audit.

8.1. ERC20 Specific Auditing

We additionally checked

- Compliance with ERC20 standard
- Token integration
- Token interaction
- Token implementation
- Best practices

8.2. ERC20 Checklist

- Returns bool after transfer(). Transfer and transferFrom return a boolean. Several tokens do not return a boolean on these functions. As a result, their calls in the contract might fail.
- Prevent transferring tokens to the 0x0 address. This may break systems that expect to be able to burn tokens by transfering them to address(0).
- Prevent transferring tokens to the contract address. Consider also preventing the transfer of tokens to the same address of the smart contract. An example of the potential for loss by leaving this open is the EOS token smart contract where more than 90,000 tokens are stuck at the contract address.
- `Pausable` tokens; the token is not pausable. Malicious or compromised owners can trap contracts relying on pausable tokens. Identify pauseable code by hand. Some tokens can be paused by an admin (e.g. `BNB`, `ZIL`). Similary to the blocklist issue above, an admin controlled pause feature opens users of the token to risk from a malicious or compromised token owner. *example*: Pausable.sol An admin controlled pause feature opens users of the token to risk from a malicious or compromised token owner.
- Reentrant Calls (ERC777). The token is not an ERC777 token and has no external function call in transfer and transferFrom. External calls in the transfer functions can lead to reentrancies.
- Fee on transfer. Transfer and transferFrom should not take a fee. Deflationary tokens can lead to unexpected behavior. Writeups of balancer being exploited: Writeup 1, Writeup 2. Some tokens take a transfer fee (e.g. `STA`, `PAXG`), some do not currently charge a fee but may do so in the future (e.g. `USDT`, `USDC`). The `STA` transfer fee was used to drain $500k from several balancer pools (more details). *example*: TransferFee.sol

- Balance Modifications Outside of Transfers (rebasing / airdrops). Potential interest earned from the token is taken into account. Some tokens distribute interest to token holders. This interest might be trapped in the contract if not taken into account.
- Upgradable tokens. A change to the token semantics can break any smart contract that depends on the past behaviour. Some tokens (e.g. `USDC`, `USDT`) are upgradable, allowing the token owners to make arbitrary modifications to the logic of the token at any point in time. A change to the token semantics can break any smart contract that depends on the past behaviour. Developers integrating with upgradable tokens should consider introducing logic that will freeze interactions with the token in question if an upgrade is detected. (e.g. the `TUSD` adapter used by MakerDAO). *example*: Upgradable.sol
- Flash Mintable tokens. Some tokens (e.g. `DAI`) allow for so called "flash minting", which allows tokens to be minted for the duration of one transaction only, provided they are returned to the token contract by the end of the transaction. This is similar to a flash loan, but does not require the tokens that are to be lent to exist before the start of the transaction. A token that can be flash minted could potentially have a total supply of max `uint256`. Documentation for the MakerDAO flash mint module can be found here. Flash minting can lead to substantial swings in the balance and the total supply, which neccessitate strict and comprehensive overflow checks in the operation of the token.
- Tokens with Blocklists. Malicious or compromised token owners can trap funds in a contract by adding the contract address to the blocklist.
- Approval Race Protections. Some tokens like USDT do not allow approving an amount M > 0 when an existing amount N > 0 is already approved. This PR shows some in the wild problems caused by this issue. To prevent the issue discussed here, Clients SHOULD make sure to create user interfaces in such a way that they set the allowance first to 0 before setting it to another value for the same spender. THOUGH The contract itself shouldn't enforce it, to allow backwards compatibility with contracts deployed before. Example here.
- Revert on Zero Value Transfers. Some tokens (e.g. LEND) revert when transfering a zero value amount.
    - If applicable, override _beforeTokenTransfer hook, including a check that prevents transfering zero value amounts. See the [docs]
- Low decimals. This may result in larger than expected precision loss. Some tokens have low decimals (e.g. `USDC` has 6). Even more extreme, some tokens like Gemini USD only have 2 decimals. This may result in larger than expected precision loss. *example*: LowDecimals.sol
- High decimals. This may trigger unexpected reverts due to overflow, posing a liveness risk to the contract. Some tokens have more than 18 decimals (e.g. `YAM-V2` has 24). This may trigger unexpected reverts due to overflow, posing a liveness risk to the contract. *example*: HighDecimals.sol
- Decimals returns a uint8. Several tokens incorrectly return a uint256. If this is the case, ensure the value returned is below 255.

- transferFrom with src == msg.sender. (e.g. OpenZeppelin, Uniswap-v2) will attempt to decrease the caller's allowance from the sender in transferFrom even if the caller and the sender are the same address, giving transfer(dst, amt) and transferFrom(address(this), dst, amt) a different semantics in this case. Some token implementations (e.g. `DSToken`) will not attempt to decrease the caller's allowance if the sender is the same as the caller. This gives `transferFrom` the same semantics as `transfer` in this case. Other implementations (e.g. OpenZeppelin, Uniswap-v2) will attempt to decrease the caller's allowance from the sender in `transferFrom` even if the caller and the sender are the same address, giving `transfer(dst, amt)` and `transferFrom(address(this), dst, amt)` a different semantics in this case.
- Non string metadata. Some tokens (e.g. MKR) have metadata fields (name / symbol) encoded as bytes32 instead of the string prescribed by the ERC20 specification.
- Revert on Approval To Zero Address. Some tokens (e.g. OpenZeppelin) will revert if trying to approve the zero address to spend tokens (i.e. a call to `approve(address(0), amt)`). Integrators may need to add special cases to handle this logic if working with such a token. *example*: ApprovalToZero.sol. Some tokens (e.g. `LEND`) revert when transfering a zero value amount. *example*: RevertZero.sol
- No Revert on Failure. Contracts should revert on failure and not just return false. Some tokens do not revert on failure, but instead return `false` (e.g. ZRX). While this is technicaly compliant with the ERC20 standard, it goes against common solidity coding practices and may be overlooked by developers who forget to wrap their calls to `transfer` in a `require`. *example*: NoRevert.sol.
- Revert on Large Approvals & Transfers. Some tokens (e.g. `UNI`, `COMP`) revert if the value passed to `approve` or `transfer` is larger than `uint96`. Both of the above tokens have special case logic in `approve` that sets `allowance` to `type(uint96).max` if the approval amount is `uint256(-1)`, which may cause issues with systems that expect the value passed to `approve` to be reflected in the `allowances` mapping. *example*: Uint96.sol.
- Code Injection Via Token Name. Some malicious tokens have been observed to include malicious javascript in their `name` attribute, allowing attackers to extract private keys from users who choose to interact with these tokens via vulnerable frontends. This has been used to exploit etherdelta users in the wild (reference).
- The contract avoids unneeded complexity. The token should be a simple contract; a token with complex code requires a higher standard of review. Use Slither's human-summary printer to identify complex code.
- The contract has only a few non–token-related functions. Non–token-related functions increase the likelihood of an issue in the contract. Use Slither's contract-summary printer to broadly review the code used in the contract.
- The token only has one address. Some proxied tokens have multiple addresses. For example TUSD has two addresses. Tokens with multiple entry points for balance updates can break internal bookkeeping based on

the address (e.g. balances[token_address][msg.sender] might not reflect the actual balance).

- No user owns most of the supply. If a few users own most of the tokens, they can influence operations based on the token's repartition.
- The token is not upgradeable. Upgradeable contracts might change their rules over time. Use Slither's human-summary printer to determine if the contract is upgradeable.
- The owner has limited minting capabilities. Malicious or compromised owners can abuse minting capabilities. Use Slither's human-summary printer to review minting capabilities, and consider manually reviewing the code.
- The owner cannot blacklist the contract. Malicious or compromised owners can trap contracts relying on tokens with a blacklist. Identify blacklisting features by hand.
- The name, decimals, and symbol functions are present if used. These functions are optional in the ERC20 standard and might not be present.