Althea Liquid Infrastructure

Scope

The code under review can be found within the C4 Althea Liquid Infrastructure repository.

Summary

Findings

ID	Issues	Severity
H-01	Attacker can inflate holders array by 0 value burn() calls to DOS protocol functionality	High
L-01	Approved holder loses all revenue if their address is blacklisted for one of the distributable ERC20s	Low
L-02	DOS of protocol functionality if there are too many holders	Low
L-03	Function setThresholds() might DOS due to OOG error	Low
L-04	Function addManagedNFT() does not check if nftContract already exists	Low

Analysis Report

Findings

[H-01] Attacker can inflate holders array by 0 value burn() calls to DOS protocol functionality

Impact

Currently, on any mints, burns or transfers, the _beforeTokenTransfer() function hook is called. The _beforeTokenTransfer() function pushes the "to" address to the `holders` array if the balance of "to" is zero.

The issue is that an approved holder or an attacker (neither approved nor holding tokens) can make 0 value burn function calls using functions burn() and burnFrom() in ERC20Burnable to inflate the holders array with zero addresses. This would cause the mint, burn, transfer functionalities to permanently DOS and distributions to temporarily DOS.

Although the distribute() function can be called directly with numDistributions as a parameter, it would still require the caller to loop through all the 0 addresses in order to end the distribution and resume regular transfers, mints and burns.

Impacts:

1. Permanent DOS of mint(), burn() and transfer() functions in some cases (since _afterTokenTransfer() runs a loop upto the holders array length).
2. Delay in distributions to approved holders who were present in the holders array after the zero addresses.
3. More funds wasted for the team who need to pay for the gas to loop through all zero addresses. This impact increases every distribution phase since an attacker can keep on pushing more zero addresses as he wishes.
4. Delay of transfers, mints and burns (since distribution takes more longer than expected)

Note: This issue is also an attack idea suggested in the README here.

Proof of Concept

Here is the whole process:

The attack can be launched from functions burn() and burnFrom() in the ERC20Burnable contract. For simplicity, I'll be demonstrating the attack path through function burn(). The attacker could also use burnFrom() to somewhat disguise himself under another account's burn calls.

1. Malicious approved holder/attacker calls burn() present in ERC20Burnable with 0 amount as parameter.

```
File: ERC20Burnable.sol
19:     function burn(uint256 amount) public virtual {
20:         _burn(_msgSender(), amount);
21:     }
```

2. Function _burn() is internally called with attacker's address and amount 0.

- On line 277, the _beforeTokenTransfer() function hook is called with account ("from") = attacker's address, "to" = address(0) and amount = 0.

```
File: ERC20.sol
274:     function _burn(address account, uint256 amount) internal virtual
{
275:         require(account != address(0), "ERC20: burn from the zero
address");
276:
277:         _beforeTokenTransfer(account, address(0), amount);
278:
279:         uint256 accountBalance = _balances[account];
280:         require(accountBalance >= amount, "ERC20: burn amount exceeds
balance");
281:         unchecked {
282:             _balances[account] = accountBalance - amount;
283:         }
284:         _totalSupply -= amount;
285:
286:         emit Transfer(account, address(0), amount);
287:
288:         _afterTokenTransfer(account, address(0), amount);
289:     }
```

3. When function _beforeTokenTransfer() is called, the following happens:

- Before we look into the function body, let's look at the parameters: `from` = attacker's address, `to` = address(0), amount = 0
- On Line 138, let's assume we pass the check since the attacker makes the call when the distribution is not occurring.
- On Lines 139-141, we do not enter the if block since `to` = address(0)
- On Lines 145-146, we enter the if block to check if `MinDistributionPeriod` has passed. Once again, we assume the attacker makes the call before the next distribution period. **Note: Distribution periods will be occurring weekly/monthly, so there would be enough time to execute this attack.**
- On Line 148, we check the balance of the `to` address i.e. address(0). Since the balance is 0 (because during burns tokens are not actually sent to the zero address but deducted from the caller's balance), `exists` is set to false.
- On Line 149-150, we enter the if block since the condition evaluates to true, and it would push address(0) to the holders array.

```
File: LiquidInfrastructureERC20.sol
133:     function _beforeTokenTransfer(
134:         address from,
135:         address to,
136:         uint256 amount
```

```
137:     ) internal virtual override {
138:         require(!LockedForDistribution, "distribution in progress");
139:         if (!(to == address(0))) {
140:             require(
141:                 isApprovedHolder(to),
142:                 "receiver not approved to hold the token"
143:             );
144:         }
145:         if (from == address(0) || to == address(0)) {
146:             _beforeMintOrBurn();
147:         }
148:         bool exists = (this.balanceOf(to) != 0);
149:         if (!exists) {
150:             holders.push(to);
151:         }
152:     }
```

4. Following this, the remaining execution continues in the _burn function(). The attacker can continue this attack and every time address(0) would be pushed to the holders array since address(0) always has 0 balance.

5. Functions mint(), burn() and transfer() would permanently DOS in some scenarios. This is because the _afterTokenTransfer() function hook runs a for loop upto the length of the holders array.

- On Line 178, it checks if the `to` address has 0 balance. If true, set `stillHolding` to false else true.
- Now let's look at each scenario of mint(), burn() and transfer().
- **In case of mint()**, `from` = address(0), `to` = some address. Since the `from` address is always address(0), which has 0 balance, `stillHolding` is set to false. Due to this, we enter the if block and the for loop would revert due to OOG exception.
- **In case of burn()**, `from` = some address, `to` = address(0). If the `from` address does not burn all its tokens, then it's `stillHolding` = true. This means we do not enter the if block and do not run the for loop. But if the `from` address tries to burn all tokens, it would revert since `stillHolding` = false due to it having no balance.
- **In case of transfer()**, `from` = user A, `to` = user B. If user A decides to transfer only some of it's tokens to B, then `stillHolding` = true since A still has a balance. This means we do not enter the if block and for loop. But if A at some point transfers it's remaining balance or whole balance to B or another address, it would revert since `stillHolding` = false due to it having no balance.

```
File: LiquidInfrastructureERC20.sol
173:     function _afterTokenTransfer(
174:         address from,
175:         address to,
176:         uint256 amount
177:     ) internal virtual override {
178:         bool stillHolding = (this.balanceOf(from) != 0);
179:         if (!stillHolding) {
180:             for (uint i = 0; i < holders.length; i++) {
181:                 if (holders[i] == from) {
182:                     // Remove the element at i by copying the last
```

```
one into its place and removing the last element
183:                     holders[i] = holders[holders.length - 1];
184:                     holders.pop();
185:                 }
186:             }
187:         }
188:     }
```

6. Function distribute() would DOS due to the for loop on Line 226 causing an OOG exception. The functions that would be permanently revert are distributeToAllHolders(), mintAndDistribute(), burnAndDistribute() and burnFromAndDistribute().

- On Line 210, the caller (usually protocol manager) can pass in `numDistributions` as a parameter to divide the array into batches. But this would waste considerable amount of funds behind gas fees during every distribution phase.
- Additionally, the caller (protocol manager) does not have an option to skip over the zero addresses since the distribution needs to end (LockedDistribution = false) in order to resume transfers, mints, burns and even withdrawFromManagedNFTs().

```
File: LiquidInfrastructureERC20.sol
210:     function distribute(uint256 numDistributions) public nonReentrant
{
211:         require(numDistributions > 0, "must process at least 1
distribution");
212:         if (!LockedForDistribution) {
213:             require(
214:                 _isPastMinDistributionPeriod(),
215:                 "MinDistributionPeriod not met"
216:             );
217:             _beginDistribution();
218:         }
219:
220:         uint256 limit = Math.min(
221:             nextDistributionRecipient + numDistributions,
222:             holders.length
223:         );
224:
225:         uint i;
226:         for (i = nextDistributionRecipient; i < limit; i++) {
227:             address recipient = holders[i];
228:             if (isApprovedHolder(recipient)) {
229:                 uint256[] memory receipts = new uint256[](
230:                     distributableERC20s.length
231:                 );
232:                 for (uint j = 0; j < distributableERC20s.length; j++)
{
233:                     IERC20 toDistribute =
IERC20(distributableERC20s[j]);
234:                     uint256 entitlement = erc20EntitlementPerUnit[j]
*
235:                         this.balanceOf(recipient);
```

```
236:                     if (toDistribute.transfer(recipient,
entitlement)) {
237:                         receipts[j] = entitlement;
238:                     }
239:                 }
240:
241:                 emit Distribution(recipient, distributableERC20s,
receipts);
242:             }
243:         }
244:         nextDistributionRecipient = i;
245:
246:         if (nextDistributionRecipient == holders.length) {
247:             _endDistribution();
248:         }
249:     }
```

Tools Used

Manual Review

Recommended Mitigation Steps

In function _beforeTokenTransfer(), check if the "to" address is the zero address. If true, do not push it. This would prevent the holders array from being inflated during burn() function calls.

There is nothing to worry about mints and transfers since ERC20Burnable disallows minting/transferring to the zero address.

[L-01] Approved holder loses all revenue if their address is blacklisted for one of the distributable ERC20s

Impact

The distribute() function is expected to distribute revenue to the approved holders in the distributable ERC20 tokens. Tokens like USDC, USDT implement blacklists that prevent blacklisted addresses from receiving/transferring any tokens.

The issue is that if an approved holder is blacklisted in one of the distributable ERC20 tokens, the holder's remaining revenue in other tokens (for which he is not blacklisted) is blocked as well.

Impacts:

1. Approved holder is blocked from receiving other revenue tokens for which he is not blacklisted.
2. Remaining approved holders in the holders array cannot receive their distribution, causing DOS.

Proof of Concept

1. Caller calls distribute() function.

2. Inside the distribute() function, this for loop is responsible for iterating over the holders array and the distributableERCs tokens array.

- On Line 227, let's say `recipient` is the holder blacklisted for one of the distributableERCs
- On Line 228, since `recipient` is approved, we enter the if block.
- Lines 232-238, run the for loop for distributing the ERC20 tokens to the `recipient`
- On Line 236, let's say the first token is the token the `recipient` is blacklisted for. Since the transfer() function is called on the token contract with the `recipient` as the parameter, the call would revert due to the blacklist.
- This would prevent the `recipient` from receiving the remaining revenue tokens (such as DAI, USDT, WETH etc) for which he is not blacklisted for. Additionally, this would also block the remaining approved holders from receiving their revenue.

```
File: LiquidInfrastructureERC20.sol
225:         uint i;
226:         for (i = nextDistributionRecipient; i < limit; i++) {
227:             address recipient = holders[i];
228:             if (isApprovedHolder(recipient)) {
229:                 uint256[] memory receipts = new uint256[](
230:                     distributableERC20s.length
231:                 );
232:                 for (uint j = 0; j < distributableERC20s.length; j++)
{
233:                     IERC20 toDistribute =
IERC20(distributableERC20s[j]);
234:                     uint256 entitlement = erc20EntitlementPerUnit[j]
*
235:                         this.balanceOf(recipient);
236:                     if (toDistribute.transfer(recipient,
entitlement)) {
237:                         receipts[j] = entitlement;
238:                     }
239:                 }
240:
241:                 emit Distribution(recipient, distributableERC20s,
receipts);
242:             }
243:         }
```

Tools Used

Manual Review

Recommended Mitigation Steps

Consider implementing a try-catch block when transferring tokens to users. This way when a call reverts due to the approved holders being blacklisted for one of the revenue tokens, the catch block would catch this and not revert. This would allow the approved holder to receive the remaining revenue tokens for which he is not blacklisted and allow the remaining approved holders to receive their revenue.

The remaining tokens left in the contract due to this could either be withdrawn by the owner of the contract through a withdraw() function implementation or could be left in the contract to be used for future distributions.

[L-02] DOS of protocol functionality if there are too many holders

Impact

If the holders array grows too long, it could DOS minting, burning and transferring of tokens in some scenarios. This is because the _afterTokenTransfer() function hook is called after each mint, burn and transfer, which runs a for loop upto the holders array length.

Impacts:

1. DOS of mint(), burn() and transfer() functionality.
2. There would also be a temporary DOS of distribute() function. But since it has the feature of batch distributions, it should work fine.

The OOG situation of distribute() is something that was expected as per the comment here and resolved through batch distributions but the mint(), burn() and transfer() function cases were not considered, where the issue could also arise.

Note: I've submitted another issue related to 0 value burn() calls that could inflate the holders array with zero addresses. Although the impacts are similar, the root cause of both issues is different. This issue talks about how the holders array grows big eventually over time through regular transfers/mints and more users being approved.

Proof of Concept

1. Currently, the mint(), burn() and transfer() functions call the _afterTokenTransfer() function hook once the state changes have been made.

- There are 3 scenarios to consider here: minting, burning and transfer of tokens
- **In case of mint()**, `from` = address(0), `to` = some address. Since the `from` address is always address(0), which has 0 balance, `stillHolding` is set to false. Due to this, we enter the if block and the for loop would revert due to OOG exception since it loops through the entire holders array.
- **In case of burn()**, `from` = some address, `to` = address(0). If the `from` address does not burn all its tokens, then it's `stillHolding` = true. This means we do not enter the if block and do not run the for loop. But if the `from` address tries to burn all tokens, we enter the for loop since `stillHolding` = false due to `from` having no balance, causing revert with OOG error.
- **In case of transfer()**, `from` = user A, `to` = user B. If user A decides to transfer only some of it's tokens to B, then `stillHolding` = true since A still has a balance. This means we do not enter the if block and for loop. But if A at some point transfers it's remaining balance or whole balance to B or another address, we enter the for loop since `stillHolding` = false due to `from` having no balance, causing revert with OOG error.

```
File: LiquidInfrastructureERC20.sol
167:     function _afterTokenTransfer(
168:         address from,
```

```
169:         address to,
170:         uint256 amount
171:     ) internal virtual override {
172:         bool stillHolding = (this.balanceOf(from) != 0);
173:         if (!stillHolding) {
174:             for (uint i = 0; i < holders.length; i++) {
175:                 if (holders[i] == from) {
176:                     // Remove the element at i by copying the last
one into its place and removing the last element
177:                     holders[i] = holders[holders.length - 1];
178:                     holders.pop();
179:                 }
180:             }
181:         }
182:     }
```

The more important cases to note here is the minting and transferring since burning would not be used that often. Through this, we can see how the functionalities could DOS if the holders array grows too long.

Tools Used

Manual Review

Recommended Mitigation Steps

Instead of using an array, consider using a mapping that stores the address to bool status.

```
mapping(address holder => bool holdsOrNot) public holders;
```

This would also require making some changes to the distribution mechanism as a pull over push mechanism. The pulling/claiming of revenue would allow anyone to pass an address to the function, which would see if it exists in the holders mapping and transfer the revenue to that address. The function would also need to include the existing ApprovedHolder checks in addition to what I just mentioned.

Overall, this would solve both the permanent DOS problems during mints/burns/transfers and temporary DOS for distributions as well since it would remove the need to implement batch distributions.

[L-03] Function setThresholds() might DOS due to OOG error

The below check ensures that once the minimum distribution period passes, mints and burns are prevented ensuring that supply changes happen after any potential distributions. Consider also restricting transfers by adding this function call here.

This would ensure no major/minor transfers can occur before distributions occur. The liquid infrastructure would be used in several way externally and each design/model would require it's own restrictions.

For example consider a case where the approvedHolder is an address that provides some kind of service to its users. The users (non-technical) are not aware of the actual revenue they should be receiving and are instead fooled by the approvedHolder who informs them of lesser revenue. In this case, the approved

holder transfers the tokens to another address right before distribution occurs (but after pastMintDistribution time). He receives the revenue on the other approved address. When the distribution ends, he transfers only a % of tokens to the users, who believe him since the tokens were to be considered non-transferrable once a distribution period begins.

```
File: LiquidInfrastructureERC20.sol
151:     function _beforeMintOrBurn() internal view {
152:         require(
153:             !_isPastMinDistributionPeriod(),
154:             "must distribute before minting or burning"
155:         );
156:     }
```

[L-04] Function addManagedNFT() does not check if nftContract already exists

Consider implementing a mapping from nftContract to bool status that tracks if an nftContract has already been added. Then add a check in the function addManagedNFT() and state updates to true/false in addManagedNFT() and releaseManagedNFT().

This would avoid duplication of an nftContract that has already been added.

```
File: LiquidInfrastructureERC20.sol
396:     function addManagedNFT(address nftContract) public onlyOwner {
397:         LiquidInfrastructureNFT nft =
LiquidInfrastructureNFT(nftContract);
398:         address nftOwner = nft.ownerOf(nft.AccountId());
399:         require(
400:             nftOwner == address(this),
401:             "this contract does not own the new ManagedNFT"
402:         );
403:         ManagedNFTs.push(nftContract);
404:         emit AddManagedNFT(nftContract);
405:     }
```

Analysis Report

Preface

This audit report should be approached with the following points in mind:

1. The report does not include repetitive documentation that the team is already aware of.
2. The report is crafted towards providing the sponsors with value such as unknown edge case scenarios, faulty developer assumptions and unnoticed architecture-level weak spots.
3. If there exists repetitive documentation (mainly in Mechanism Review), it is to provide the judge with more context on a specific high-level or in-depth scenario for ease of understandability.

Approach taken in evaluating the codebase

Time spent on this audit: 14 hours

0-2 hours:

- Understood the codebase and architecture of the contracts
- Added inline bookmarks for gas optimizations and low-severity issues

2-5 hours:

- Adding possible HM severity issues
- Asking sponsors questions not included in documentation
- Examining possible edge cases/extreme values for input parameters

5-10 hours:

- Filtering out inline bookmarks
- Writing reports for HM severity issues
- Drawing mental models of processes

10-12 hours:

- Writing Gas and QA reports
- Searching for possible optimizations for some more time

12-14 hours:

- Writing Analysis Report
- Reviewing findings

Architecture recommendations

Protocol Structure

The protocol is structured in a simplistic manner. There are 3 contracts: LiquidInfrastructureERC20, LiquidInfrastructureNFT and OwnableApprovableERC721. The LiquidInfrastructureERC20 can own/manage multiple LiquidInfrastructureNFT contracts. The LiquidInfrastructureNFT contracts could be standalone as well depending on their use case.

A new model:

The protocol could be made more simplistic if the ERC1155 standard is used. In this case, the contract that inherits ERC1155 would hold the LiquidInfrastructureNFTs. The LiquidInfrastructureNFT would be represented using the tokenIds, which itself could be further fractionalizable due to tokenIds being semi-fungible. All functions related to LiquidInfrastructureNFT would be included in the contract that inherits the ERC1155 contract.

Where does the revenue come in from?

- The revenue in the original model first needs to be deposited to the LiquidInfrastructureNFT contract by the micro-tx module and then transferred to the LiquidInfrastructureERC20.

- This extra jump won't be required in case of the new model. The micro-tx module can look at the thresholds set by the owner of the tokenId using setThresholds() existing in the contract that inherits the ERC1155 contract.

How does revenue distribution occur?

- Since the revenue ERC20 tokens are deposited by the micro-tx module into the ERC1155 contract itself, the distribution would also occur from the distribute() function existing in the contract.

What will the ERC20 token holders hold?

- The original model required users to hold only LiquidInfrastructureERC20 tokens to earn revenue.
- The new model can allow users to hold onto any tokens to earn revenue. The protocol can decide which tokens to consider. The team can also just use one standalone ERC20 contract token (as done in LiquidInfrastructureERC20) to achieve the same purpose as the original model.

This model would provide the same features/benefits as the original model, with an add-on of the tokenId/LiquidInfrastructureNFT being fractionalizable.

What's unique?

1. Representation of RWA - Real world assets are tokenized using NFTs, which can provide revenue in multiple tokens to infrastructure holders. This incentivization opens up a new realm of earning opportunities to individuals who provide services on Althea and support the network by holding infrastructure tokens.
2. DEXes for infrastructure token holders - The team/protocol managers are expected to build a DEX, which would allow approved holders to swap and LP their tokens to one another. This creates a like minded community of holders supporting Althea by creating a market for them.
3. Regular holders - Although the infrastructure model seems to be inclining towards RWAs, there is still a place for regular holders/investors to profit of the revenue just by holding the infrastructure tokens in a decentralized manner.

What ideas can be incorporated?

1. Creating a DAO for infrastructure token holders on Althea. This DAO would vote on proposals such as possible revenue token additions they might want to earn in and removal of any existing revenue tokens or maybe even adding/releasing NFTs that are not generating any revenue.
2. Providing revenue back to the NFT that provides highest revenue. The NFT when released would be sent to the original owner who created the NFT as a token of appreciation.
3. Revenue could be redirected towards a treasury as a small fee for the infra service the althea team provides. This could be used by the team to support projects on althea and provide them with grants.

Centralization risks

The biggest centralization vector in the codebase is the owner. The owner has the following privileges currently:

1. Mint tokens to any address
2. Approve/Disapprove any holders
3. Add/remove managed NFT contracts

4. Update the revenue ERC20 tokens

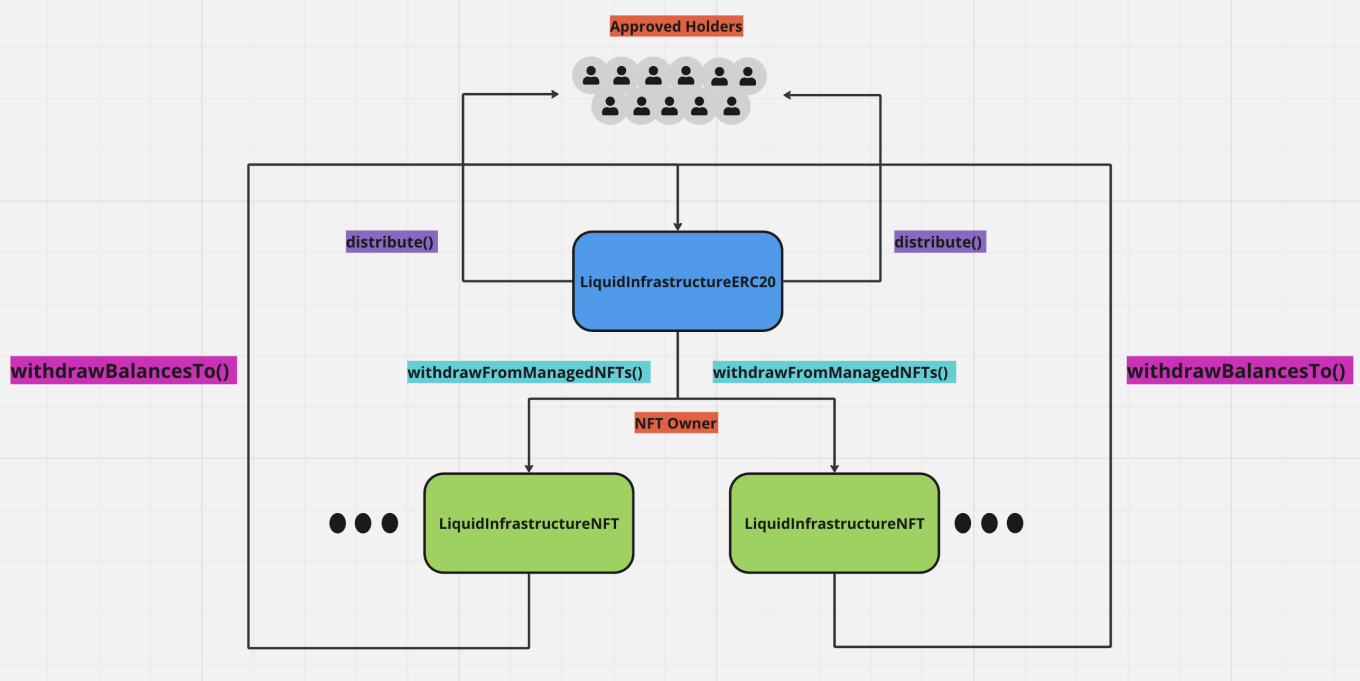
Consider using a multi-sig for the owner role. If possible, consider rotating the owner role to new multi-sigs every 6 months for additional security.

Resources used to gain deeper context on the codebase

1. Althea documentation
2. Cosmos documentation

Mechanism Review

High-level System Overview



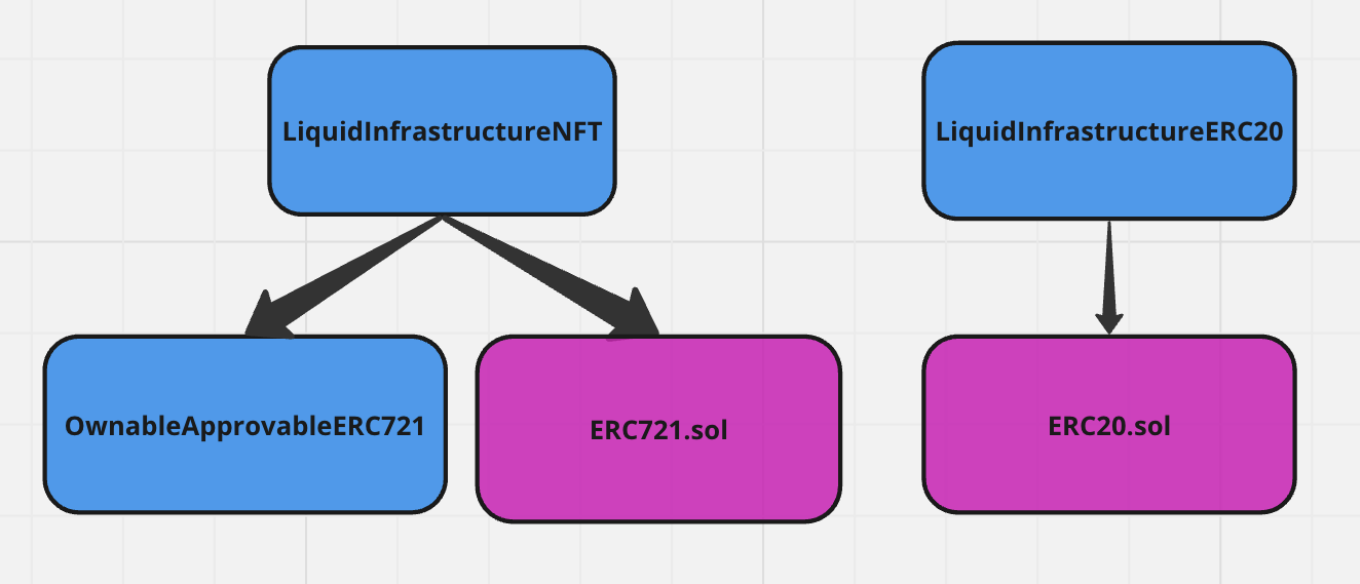
Chains supported

- Althea L1
- Gnosis (might be deployed as per sponsor)
- EVM chains (by other teams/maintainers)

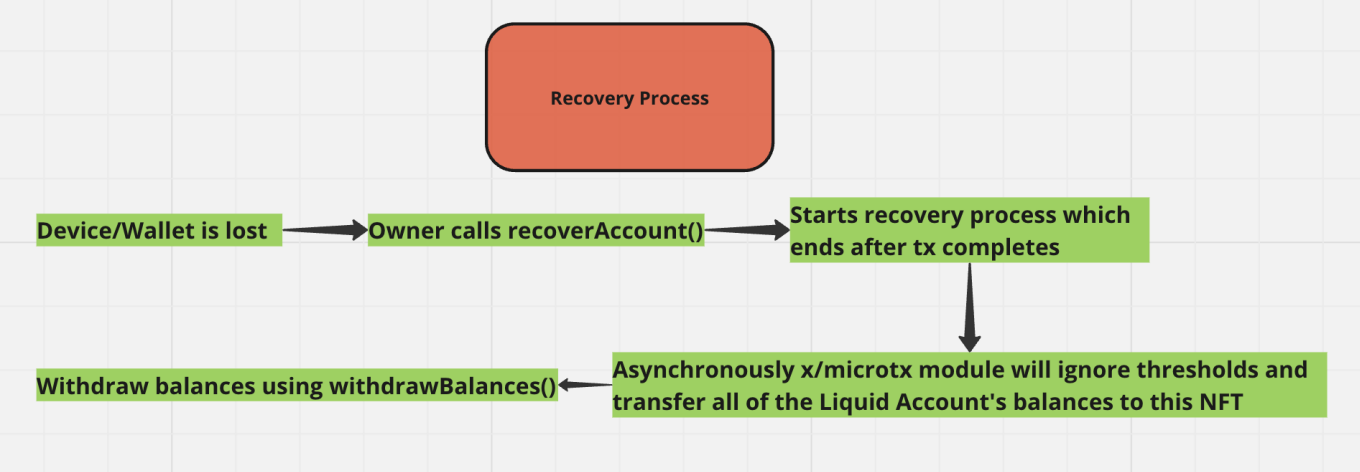
Mental models

Inheritance structure

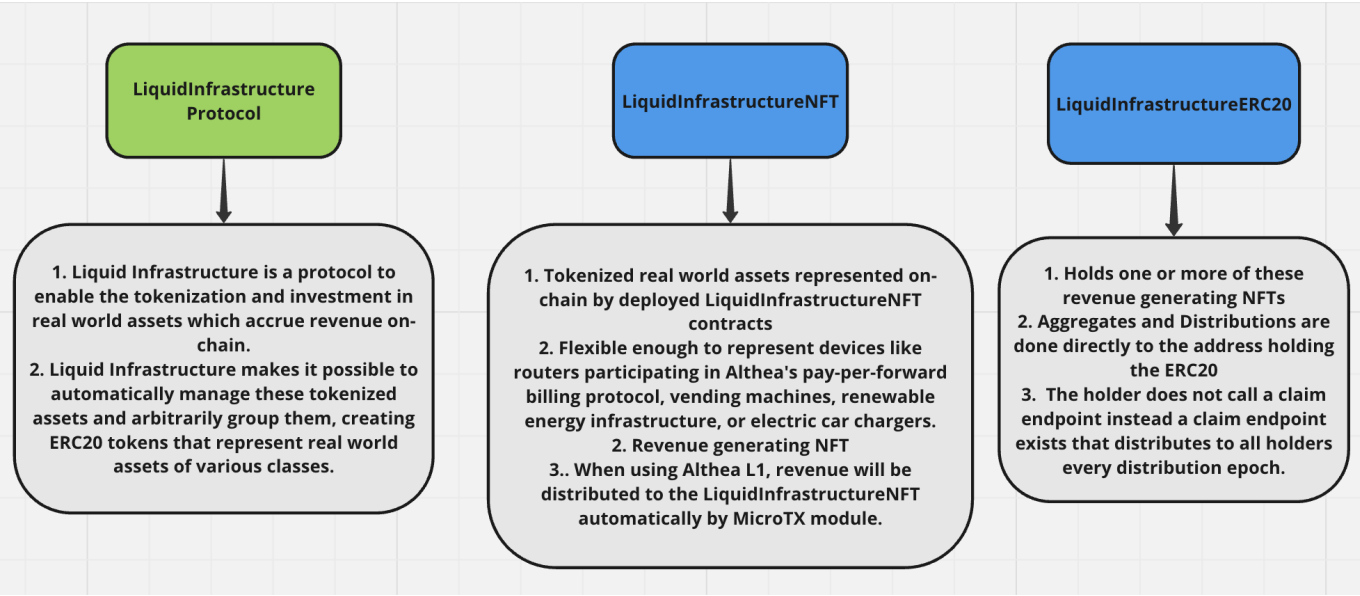
Contracts marked other than blue are out of scope.



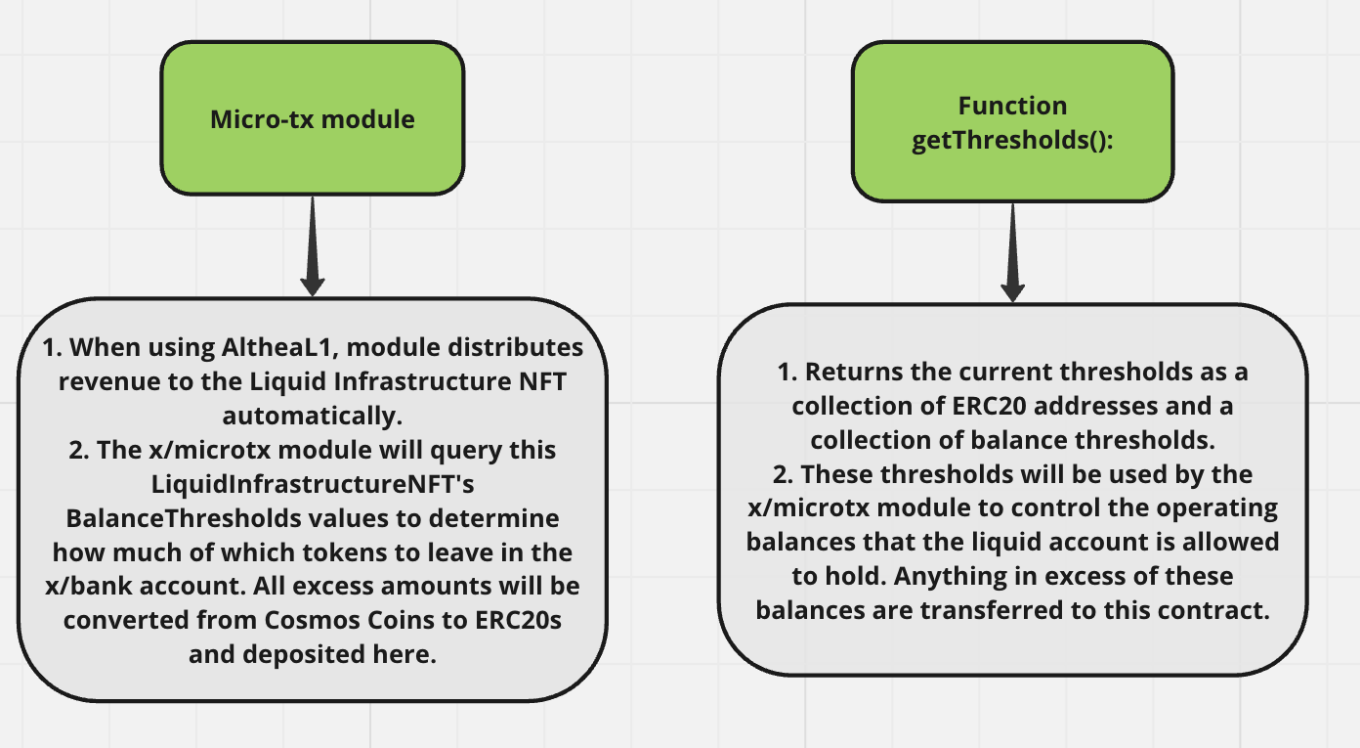
Recovery mechanism



Features of Protocol/Contracts



Features of Micro-tx module



Systemic Risks/Architecture-level weak spots and how they can be mitigated

1. Contracts would not work on all chains due to tokens like USDT, USDC, DAI, WETH etc having different signatures and decimals on different chains. Before an Althea maintainer deploys contract on the other chain, check if the tokens in scope use the correct function signatures. For example, USDT does not return bool in its transfer() function. Another example, USDT, USDC uses 6 decimals while DAI uses 18 decimals.
2. Address poisoning - Currently the protocol allows 0 value transfers. This opens up doors to address poisoning attacks, which could cause users to send tokens to incorrect addresses.
3. Zero value burn calls allow inflating the holders array to cause permanent OOG errors. This attack can be launched by a non-approved holder as well and requires urgent action. To avoid this, do not push the 0 address to the holders arrays as well on burns.
4. Consider using mappings over arrays - Currently there are numerous instances that could cause the protocol to undergo frequent OOG errors. The managedNFTs, erc20Entitlements, distributableERC20s, holders, thresholdERC20s arrays are at the risk of causing OOG errors when used in for loops.
5. Consider locking the setter function for distributable erc20 tokens during distribution period. This would ensure the owner cannot mess around with any revenue tokens for some of the users mid distributions.

Issues surfaced from Attack Ideas in README

1. Attack Idea - Errors in the distribution to holders, such as the ability to take rewards entitled to another holder

- Although other holders cannot steal rewards, there are two cases related to reward distributions.
- In case one of the holders is blacklisted for just one of the revenue tokens, they are devoid of receiving the remaining revenue tokens for which they are not blacklisted. This also prevents the remaining holders from receiving their revenue since the distribute() function reverts.

- The balance/supply calculation is flawed since if balance < supply or balance uses decimals less than 18 (such as USDT, USDC), the entitlement would round down to 0, causing holders to receive no payments for that distribution period. Those tokens remain in the contract and are subject to the same issue in future distribution periods unless the balance >= supply.

2. Attack Idea - DoS attacks against the LiquidInfrastructureERC20 are of significant concern, particularly if there is a way to permanently trigger out of gas errors

- There are 2 scenarios that lead to mints, burns and transfers being permanently DOSed.
- In case 1, an attacker (approved or non-approved holder) can make 0 value burn calls to inflate the holders array. This would cause OOG in the for loop of _afterTokenTransfer() function.
- In case 2, if the holders array grows too large, it would cause the OOG in the for loop of _afterTokenTransfer() function.

3. Attack Idea - Acquiring the ERC20 (and therefore rewards) without approval is another significant concern

- There are no issues regarding this.

Some questions I asked myself and other FAQ:

1. What are Liquid accounts?

- Once Althea-L1 is deployed, real-world devices will have accounts on the chain. They will use these accounts to send and receive ERC20 tokens for a service, most likely because they are part of the Althea internet service protocol. Each device account can opt-in to becoming a Liquid account - they will have a LiquidInfrastructureNFT deployed and will automatically forward balances beyond a configured threshold to the NFT. Althea-L1 the chain will manage the flow of stablecoins from the device accounts to the LiquidInfrastructureNFT.

2. Is it possible for any functionality to DOS?

- Yes, mints/burns/transfers are possible to DOS using 0 value burn calls

3. Are the contracts ERC20 and ERC721 compliant?

- Yes, in most cases. There are some cases such as blocking of transfers during distribution that break the spec somewhat.